LECTURE NOTES

EC3352-DIGITAL SYSTEM DESIGN

II YEAR – III SEMESTER – R2021

UNIT- I BASIC CONCEPTS

INTRODUCTION:

In 1854, George Boole, an English mathematician, proposed algebra for symbolically representing problems in logic so that they may be analyzed mathematically. The mathematical systems founded upon the work of Boole are called ***Boolean algebra*** in his honor.

The application of a Boolean algebra to certain engineering problems was introduced in 1938 by C.E. Shannon.

For the formal definition of Boolean algebra, we shall employ the postulates formulated by E.V. Huntington in 1904.

Fundamental postulates of Boolean algebra:

The postulates of a mathematical system forms the basic assumption from which it is possible to deduce the theorems, laws and properties of the system.

The most common postulates used to formulate various structures are—

i) **Closure**:

A set S is closed w.r.t. a binary operator, if for every pair of elements of S, the binary operator specifies a rule for obtaining a unique element of S.

The result of each operation with operator (+) or (.) is either 1 or 0 and 1, 0 ϵB.

ii) **Identity element**:

A set S is said to have an identity element w.r.t a binary operation * on S, if there exists an element e ϵ S with the property,

$$e * x = x * e = x$$

Eg: 0+ 0 = 0 0+1=1+0=1 a) **x+ 0= x**

1.1=1 1.0=0.1=1 b) **x. 1 = x**

iii) **Commutative law**:

A binary operator * on a set S is said to be commutative if,

$$x * y = y * x$$ for all x, y ϵ S

Eg: 0+ 1 = 1+ 0 = 1 a) **x+ y= y+ x**

0 . 1 = 1 . 0 = 0 b) **x. y= y. x**

iv) **Distributive law**:

If * and • are two binary operation on a set S, • is said to be distributive over + whenever,

x . (y+ z) = (x. y) + (x. z)

Similarly, + is said to be distributive over • whenever,

x + (y. z) = (x+ y). (x+ z)

v) **Inverse**:

A set S having the identity element e, w.r.t. binary operator * is said to have an inverse, whenever for every x Є S, there exists an element x' Є S such that,

x. x' Є e

a) x+ x' = 1, since 0 + 0' = 0+ 1 and 1+ 1' = 1+ 0 = 1

b) x. x' = 1, since 0 . 0' = 0. 1 and 1. 1' = 1. 0 = 0

Summary:

Postulates of Boolean algebra:

POSTULATES	(a)	(b)
Postulate 2 (Identity)	**x + 0 = x**	**x . 1 = x**
Postulate 3 (Commutative)	**x+ y = y+ x**	**x . y = y. x**
Postulate 4 (Distributive)	**x (y+ z) = xy+ xz**	**x+ yz = (x+ y). (x+ z)**
Postulate 5 (Inverse)	**x+x' = 1**	**x. x' = 0**

Basic theorem and properties of Boolean algebra:

Basic Theorems:

The theorems, like the postulates are listed in pairs; each relation is the dual of the one paired with it. The postulates are basic axioms of the algebraic structure and need no proof. The theorems must be proven from the postulates. The proofs of the theorems with one variable are presented below. At the right is listed the number of the postulate that justifies each step of the proof.

1) a) **x+ x = x**

x+ x = (x+ x) . 1----------	by postulate 2(b) [x. 1 = x]
= (x+ x). (x+ x')----------	5(a) [x+ x' = 1]
= x+ xx'----------	4(b) [x+yz = (x+y)(x+z)]
= x+ 0----------	5(b) [x. x' = 0]
= x----------	2(a) [x+0 = x]

b) x. x = x

x. x = (x. x) + 0----------	by postulate 2(a) [x+ 0 = x]
= (x. x) + (x. x')----------	5(b) [x. x' = 0]
= x (x+ x')----------	4(a) [x (y+z) = (xy)+ (xz)]
= x (1)----------	5(a) [x+ x' = 1]
= x----------	2(b) [x.1 = x]

2) a) **x+ 1 = 1**

x+ 1 = 1 . (x+ 1)----------	by postulate 2(b) [x. 1 = x]
= (x+ x'). (x+ 1)----------	5(a) [x+ x' = 1]
= x+ x'.1----------	4(b) [x+yz = (x+y)(x+z)]
= x+ x'----------	2(b) [x. 1 = x]
= 1----------	5(a) [x+ x'= 1]

b) **x .0 = 0**

3) (x')' = x

From postulate 5, we have x+ x' = 1 and x. x' = 0, which defines the complement of x. The complement of x' is x and is also (x')'.
Therefore, since the complement is unique,

(x')' = x.

4) <u>Absorption Theorem</u>:

a) x+ xy = x

x+ xy = x. 1 + xy---------- by postulate 2(b) [x. 1 = x]
= x (1+ y)---------- 4(a) [x (y+z) = (xy)+ (xz)]
= x (1) ----------by theorem 2(a) [x+ 1 = x]
= x ----------by postulate 2(a) [x. 1 = x]

b) x. (x+ y) = x

x. (x+ y) = x. x+ x. y---------- 4(a) [x (y+z) = (xy)+ (xz)]
= x + x.y ----------by theorem 1(b) [x. x = x]
= x ----------by theorem 4(a) [x+ xy = x]

c) x+ x'y = x+ y

x+ x'y = x+ xy+ x'y---------- by theorem 4(a) [x+ xy = x]
= x+ y (x+ x') ----------by postulate 4(a) [x (y+z) = (xy)+ (xz)]
= x+ y (1)---------- 5(a) [x+ x' = 1]
= x+ y---------- 2(b) [x. 1= x]

d) x. (x'+y) = xy

x. (x'+y) = x.x'+ xy---------- by postulate 4(a) [x (y+z) = (xy)+ (xz)]
= 0+ xy---------- 5(b) [x. x' = 0]
= xy.---------- 2(a) [x+ 0= x]

Properties of Boolean algebra:

1. Commutative property:

Boolean addition is commutative, given by

x+ y = y+ x

According to this property, the order of the OR operation conducted on the variables makes no difference.

Boolean algebra is also commutative over multiplication given by,

x. y = y. x

This means that the order of the AND operation conducted on the variables makes no difference.

2. Associative property:

The associative property of addition is given by,

A+ (B+ C) = (A+B) + C

The OR operation of several variables results in the same, regardless of the grouping of the variables.

The associative law of multiplication is given by,

A. (B. C) = (A.B) . C

It makes no difference in what order the variables are grouped during the AND operation of several variables.

3. Distributive property:

The Boolean addition is distributive over Boolean multiplication, given by

A+ BC = (A+B) (A+C)

The Boolean addition is distributive over Boolean addition, given by

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

4. Duality:

It states that every algebraic expression deducible from the postulates of Boolean algebra remains valid if the operators and identity elements are interchanged.

If the dual of an algebraic expression is desired, we simply interchange OR and AND operators and replace 1's by 0's and 0's by 1's.

x+ x' = 1 is **x. x' = 0**

Duality is a very important property of Boolean algebra.

Summary:

Theorems of Boolean algebra:

	THEOREMS	(a)	(b)
1	Idempotent	**x + x = x**	**x . x = x**
		x + 1 = 1	**x . 0 = 0**
2	Involution	**(x')' = x**	
3	Absorption	**x+ xy = x**	**x (x+ y) = x**
		x+ x'y = x+ y	**x. (x'+ y)= xy**
4	Associative	**x+(y+ z)= (x+ y)+ z**	**x (yz) = (xy) z**
5	DeMorgan's Theorem	**(x+ y)'= x'. y'**	**(x. y)'= x'+ y'**

DeMorgan's Theorems:

Two theorems that are an important part of Boolean algebra were proposed by DeMorgan.

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

(AB)' = A'+ B'

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

(A+ B)' = A'. B'

Consensus Theorem:

In simplification of Boolean expression, an expression of the form **AB+ A'C+ BC**, the term BC is redundant and can be eliminated to form the equivalent expression AB+ A'C. The theorem used for this simplification is known as consensus theorem and is stated as,

AB+ A'C+ BC = AB+ A'C

The dual form of consensus theorem is stated as,

(A+B) (A'+C) (B+C) = (A+B) (A'+C)

BOOLEAN FUNCTIONS:

Minimization of Boolean Expressions:

The Boolean expressions can be simplified by applying properties, laws and theorems of Boolean algebra.

Simplify the following Boolean functions to a minimum number of literals:

1. x (x'+y)

= xx'+ xy **[x. x'= 0]**
= 0 + xy **[x+ 0 = x]**
= xy.

2. x+ x'y

= x + xy + x'y **[x+ xy= x]**
= x+ y (x+x')
= x+ y (1) **[x+ x' = 1]**
= x+ y.

3. (x+ y) (x+ y')

= x.x+ xy'+ xy+ yy'
= x+ xy'+ xy+ 0 **[x. x= 0]; [y. y'= 0]**
= x (1+ y'+ y)
= x (1) **[1+y= 1]**
= x.

4. xy + x'z + yz.

= xy + x'z + yz(x+ x') **[x+ x'= 1]**
= xy + x'z + xyz + x'yz

Re-arranging,

= xy + xyz + x'z +x'yz
= xy (1+ z) + x'z (1+y) **[1+y= 1]**
= xy+ x'z.

5. xy+ yz+ y'z

= xy+ z (y+ y')
= xy+ z (1) **[y+ y' = 1]**
= xy+ z.

6. (x+ y) (x'+ z) (y+ z)

= (x+ y) (x'+ z) **[** dual form of consensus theorem**,**
(A+ B) (A'+ C) (B+ C) = (A+ B) (A'+ C)]

7. x'y+ xy+ x'y'

= y (x'+ x) + x'y' **[x (y+ z) = xy+ xz]**
= y (1) + x'y' **[x+ x' = 1]**
= y+ x'y' **[x+ x'y' = x+ y']**
= y+ x'.

8. x+ xy'+ x'y

= x (1+ y')+ x'y
= x (1) + x'y **[1+ x = 1]**
= x+ x'y **[x+ x'y = x+ y]**
= x+ y.

9. AB + (AC)' + AB'C (AB + C)

= AB + (AC)' + AAB'BC + AB'CC
= AB + (AC)' + 0+ AB'CC **[B.B' = 0]**
= AB + (AC)' + AB'C **[C.C = 1]**
= AB + A' + C' +AB'C **[(AC)' = A' + C']**
= AB+A'+C'+AB' **[C' + AB'C = C' + AB']**
= A' + B+ C'+ AB' **[A'+AB=A'+B]**

Re- arranging,

=A'+AB'+B+C' **[A'+AB=A'+B]**
=A'+B'+B+C' **[B'+B=1]**
= A'+1+C' **[A+ 1=1]**

= 1

10. (x'+ y) (x+ y)

= x'.x+ x'y+ yx+ y.y
= 0+ x'y+ xy+ y **[x.x'= 0]; [x. x= x]**
= y (x'+ x+ 1)
= y(1) **[1+ x = 1]**
= y.

11. xy+ xyz+ xy (w+ z)

= xy (1+ z+ w+ z)

= xy (1) [1+ x = 1]

= xy.

12. xy+ xyz+ xyz'+ x'yz

= xy (1+ z+ z')+ x'yz

= xy (1) + x'yz [1+ x = 1]

= xy+ x'yz

= y (x+ x'z) [x+ x'y = x+ y]

= y (x+ z).

13. xyz+ xy'z+ xyz'

= xy (z+ z') + xy'z

= xy+ xy'z [x+ x'= 1]

= x(y+ y'z) [x+ x'y = x+ y]

= x(y+ z)

14. x'y'z'+ x'yz'+ xy'z'+ xyz'

= x'z' (y'+ y)+ xz' (y'+ y)

= x'z'+ xz' [x+ x'= 1]

= z' (x'+ x)

= z' [x+ x'= 1]

15. w'xyz'+ xyz'+ xy'z'+ xy'z

= xyz' (w'+ 1) + xy'z'+ xy'z

= xyz'+ xy'z'+ xy'z [1+ x = 1]

= xz' (y+ y') + xy'z

= xz'+ xy'z [x+ x'= 1]

= x (z'+ y'z)

= x (z'+ y'). [x'+ xy' = x'+ y']

16. w'xy'z+ w'xyz+ wxz

= w'xz (y'+ y)+ wxz

= w'xz (1)+ wxz [x+ x'= 1]

= w'xz+ wxz

= xz (w'+ w)

= xz. [x+ x'= 1]

17. x'y'z'+ x'y'z+ x'yz'+ x'yz+ xy'z'

= x'y' (z'+z) + x'y (z'+z)+ xy'z'

= x' y' (1) + x'y (1)+ xy'z' [x+ x'= 1]

= x'y' + x'y + xy'z'

= x'(y'+y) + xy'z'

= x' (1) + xy'z' **[x+ x'= 1]**

= x' + xy'z'

= x'+ y'z'. **[x'+ xy' = x'+ y']**

18. w'y (w'xz)' + w'xy'z' + wx'y

= w'y (w''+ x'+ z') + w'xy'z' + wx'y

= w'y (w+ x'+ z') + w'xy'z' + wx'y **[x'' = x]**

= w'yw+ w'y x'+ w'y z' + w'xy'z' + wx'y

= 0 + w'x'y+ w'y z' + w'xy'z' + wx'y **[x. x'= 0]**

Re-arranging,

= w'x'y+ wx'y + w'y z' + w'xy'z'

= x'y (w'+ w) + w'z' (y+ xy')

= x'y (1) + w'z' (y+ xy') **[x+ x'= 1]**

= x'y+ w'z' (y+x) **[x+ x'y = x+ y]**

19. xy+ x (y+ z) + y (y+ z)

= xy+ xy+ xz+ yy+ yz

= xy+ xz+ y+ yz **[x+ x= x]; [x. x= x]**

= xy+ xz+ y **[x+ xy= x]**

= y+ xz **[x+ xy= x]**

20. [xy' (z+ wy) + x'y'] z

= [xy'z+ xy'wy+ x'y'] z

= [xy'z+ 0+ x'y'] z **[x. x'= 0]**

= xy'z. z+ x'y'z

= xy'z+ x'y'z **[x. x= x]**

= y'z (x+ x')

= y'z (1) **[x+ x'= 1]**

= y'z.

21. x'yz+ xy'z'+ x'y'z'+ xy'z+ xyz

= yz (x'+x) + xy'z'+ x'y'z'+ xy'z

= yz (1) + y'z' (x+ x') + xy'z **[x+ x'= 1]**

= yz+ y'z' (1) + xy'z **[x+ x'= 1]**

= yz+ y'z'+ xy'z

= yz+ y' (z'+ xz)

= yz+ y' (z'+ x) **[x'+ xy = x'+ y]**

= yz+ y'z'+ xy'

22. [(xy)'+ x'+ xy]'

= [x'+ y'+ x'+ xy]'

= [x'+ y'+ xy]' **[x+ x= x]**

= [x'+ y'+ x]' [x'+ xy = x'+ y]
= [y'+ 1]' [x+ x'= 1]
= [1]' [1+ x = 1]

= 0.

23. [xy+ xz]'+ x'y'z

= (xy)'. (xz)'+ x'y'z
= (x'+ y'). (x'+ z')+ x'y'z
= x'x'+ x'z'+ x'y'+ y'z'+ x'y'z

= x'+ x'z'+ x'y'+ y'z'+ x'y'z[x+ x= x]
= x'+ x'z'+ x'y'+ y' [z'+ x'z]
= x'+ x'z'+ x'y'+ y' [z'+ x'] [x'+ xy = x'+ y]
= x'+ x'y'+ y' [z'+ x'] [x+ xy = x]
= x'+ x'y'+ y'z'+ x'y'

= x'+ y'z'+ x'y' [x+ xy = x]
= x'+ y'z'. [x+ xy = x]

24. xy+ xy'(x'z')'

= xy+ xy' (x''+ z'')
= xy+ xy' (x+ z) [x'' = x]
= xy+ xy'x+ xy'z
= xy+ xy'+ xy'z [x. x= x]
= xy+ xy' [1+ z]
= xy+ xy' [1] [1+ x = 1]
= xy+ xy'
= x(y+ y')

= x [1] [x+ x'= 1]

= x.

25. [(xy'+ xyz)'+ x (y+ xy')]'

= [x(y'+yz)'+ x (y+ xy')]'
= [x(y'+z)'+ x (y+ x)]' [x'+ xy = x'+ y]; [x+ x'y = x+ y]
= [x(y'+z)'+ xy+ x.x)]'
= [(xy'+xz)'+ xy+ x)]' [x. x= x]
= [(xy'+xz)'+ x)]' [x+ xy = x]
= [(xy')'. (xz)'+ x]'
= [(x'+y''). (x'+z')+ x]'
= [(x'+y). (x'+z')+ x]' [x'' = x]
= [(x'+ yz')+ x]' [(x+ y) (x+ z)= x+ yz]
= [x'+ yz'+ x]'
= [1+ yz']' [x+ x'= 1]
= [1]' [1+ x = 1]

= 0.

26. [(xy+ z') ((x+ y)'+z)]'

= [(xy+ z') ((x'. y')+z)]'
= [xy. x'y'+ xy. z+ z'. x'y'+ z'. z]'
= [0+ xyz+ x'y'z'+ 0]' **[x. x'= 0]**
= [xyz+ x'y'z']'
= (xyz)'. (x'y'z')'
= (x'+ y'+ z'). (x''+ y''+ z'')
= (x'+ y'+ z'). (x+ y+ z). **[x'' = x]**

27. (x+ y) (x'z'+ z) (y'+ xz)'

= (x+ y) (x'z'+ z) (y''. (xz)')
= (x+ y) (x'+ z) (y. (xz)') **[x+ x'y = x+ y]; [x'' = x]**
= (x+ y) (x'+ z) (y. (x'+z'))
= (x.x'+ xz+ x'y+ yz) (x'y+ yz')
= (0+ xz+ x'y+ yz) (x'y+ yz')
= (xz+ x'y+ yz) (x'y+ yz')
= xz. x'y+ xz. yz'+ x'y. x'y+ x'y. yz'+ yz. x'y+ yz. yz'
= 0+ 0+ x'y+ x'yz'+ x'yz+ 0 **[x. x'= 0]; [x. x= x]**
= x'y+ x'yz'+ x'yz
= x'y (1+ z'+ z)
= x'y (1) **[1+ x = 1]**
= x'y.

28. Y= ∑m (1, 3, 5, 7)

= x'y'z+ x'yz+ xy'z+ xyz
= x'z(y'+y) + xz(y'+y)
= x'z (1)+ xz (1) **[x+ x'= 1]**
= x'z+ xz
= z(x'+ x)
= z (1) **[x+ x'= 1]**
= z.

COMPLEMENT OF A FUNCTION:

The complement of a function F is F' and is obtained from an interchange of 0's for 1's and 1's for 0's in the value of F. The complement of a function may be derived algebraically through DeMorgan's theorem.

DeMorgan's theorems for any number of variables resemble in form the two-variable case and can be derived by successive substitutions similar to the method used in the preceding derivation. These theorems can be generalized as –

(A+B+C+D+…+F)'=A'B'C'D'…F'

(A B C D … F)' = A'+B'+ C'+ D'+ … +F'.

Find the complement of the following functions,

1. **F= x'yz'+ x'y'z**

   F'= (x'yz'+ x'y'z)'
   = (x"+ y'+ z") . (x"+ y"+z')
   = (x+ y'+ z). (x+ y+ z').

2. **F= (xy + y'z + xz) x.**

   F' = [(xy + y'z + xz) x]'
   = (xy + y'z + xz)' + x'
   = [(xy)' . (y'z)'. (xz)'] + x'
   = [(x'+y'). (y+z'). (x'+z')] + x'
   = [(x'y+ x'z'+ 0+ y'z') (x'+z')] + x'
   = x'x'y+ x'x'z'+ x'y'z'+ x'yz'+ x'z'z'+ y'z'z'+ x'
   = x'y+ x'z'+ x'y'z'+ x'yz'+ x'z'+ y'z'+ x' [x+ x = x], [x. x = x]
   = x'y+ x'z'+ x'z' (y'+ y) + y'z'+ x' [x+ x'= 1]
   = x'y+ x'z'+ x'z' (1) + y'z'+ x'
   = x'y+ x'z'+ y'z'+ x'
   = x'y+ x'+ x'z'+ y'z'

   = x'(y+1) + x'z+ y'z' [y+1= 1]
   = x' (1+z) + y'z' [y+1= 1]
   = x'+ y'z'

3. **F= x (y'z'+ yz)**

F'= [x (y'z'+ yz)]'

= x'+ (y'z'+ yz)'

= x'+ (y'z')'. (yz)'

= x'+ (y"+ z") . (y'+ z')

= x'+ (y+ z) . (y'+ z').

4. **F= xy'+ x'y**

F'= (xy'+ x'y)'

= (xy')'. (x'y)'

= (x'+y) (x+y')

= x'x+ x'y'+ yx+ yy'

= x'y'+ xy.

5. **f = wx'y + xy'+ wxz**

f' = (wx'y + xy'+ wxz)'

= (wx'y)' (xy')' (wxz)'

= (w'+x+ y') (x'+ y) (w'+ x'+ z')

= (w'x'+ w'y+ xx'+ xy+ x'y'+ yy') (w'+ x'+ z')

= (w'x'+ w'y+ xy+ x'y') (w'+ x'+ z')

= w'x'. w'+ w'y. w'+ xy. w'+ x'y'. w'+ w'x'. x'+w'y. x'+ xy. x'+ x'y'. x'+
w'x'. z'+ w'y. z'+ xy. z'+ x'y'.z'

= w'x'+ w'y+ w'xy+ w'x'y'+ w'x'+ w'x'y+ 0 + x'y'+ w'x'z'+ w'yz'+ xyz'+ x'y'z'

= w'x'+ w'y+ w'xy+ w'x'y'+ w'x'y+ x'y'+ w'x'z'+ w'yz'+ xyz'+ x'y'z'

= w'x'(1+ y'+ y+ z')+ w'y(1+ x+ z')+ x'y'(1+ z')+ xyz'

= w'x'(1)+ w'y(1)+ x'y'(1)+ xyz'

= w'x'+ w'y+ x'y'+ xyz'

CANONICAL AND STANDARD FORMS:

Minterms and Maxterms:

A binary variable may appear either in its normal form (x) or in its complement form (x'). Now either two binary variables x and y combined with an AND operation. Since each variable may appear in either form, there are four possible combinations:

x'y', x'y, xy' and xy

Each of these four AND terms is called a '***minterm***'.

In a similar fashion, when two binary variables x and y combined with an OR operation, there are four possible combinations:

x'+ y', x'+ y, x+ y' and x+ y

Each of these four OR terms is called a '***maxterm***'.

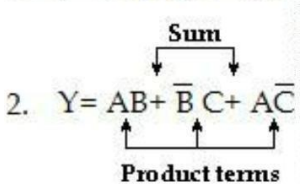
The minterms and maxterms of a 3- variable function can be represented as in table below.

Variables			Minterms	Maxterms
x	**y**	**Z**	**m_i**	**M_i**
0	0	0	$x'y'z' = m_0$	$x+ y+ z= M_0$
0	0	1	$x'y'z = m_1$	$x+ y+ z'= M_1$
0	1	0	$x'yz' = m_2$	$x+ y'+ z= M_2$
0	1	1	$x'yz = m_3$	$x+ y'+ z'= M_3$
1	0	0	$xy'z' = m_4$	$x'+ y+ z= M_4$
1	0	1	$xy'z = m_5$	$x'+ y+ z'= M_5$
1	1	0	$xyz' = m_6$	$x'+ y'+ z= M_6$
1	1	1	$xyz = m_7$	$x'+ y'+ z'= M_7$

Sum of Minterm: (Sum of Products)

The logical sum of two or more logical product terms is called sum of products expression. It is logically an OR operation of AND operated variables such as:

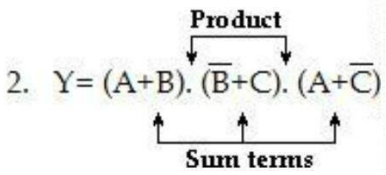
1. $Y= AB+ BC+ AC$



Sum of Maxterm: (Product of Sums)

A product of sums expression is a logical product of two or more logical sum terms. It is basically an AND operation of OR operated variables such as,

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



Canonical Sum of product expression:

If each term in SOP form contains all the literals then the SOP is known as standard (or) canonical SOP form. Each individual term in standard SOP form is called minterm canonical form.

F (A, B, C) = AB'C+ ABC+ ABC'

Steps to convert general SOP to standard SOP form:

1. Find the missing literals in each product term if any.
2. AND each product term having missing literals by ORing the literal and its complement.
3. Expand the term by applying distributive law and reorder the literals in the product term.
4. Reduce the expression by omitting repeated product terms if any.

Obtain the canonical SOP form of the function:

1. Y(A, B) = A+ B
   = A. (B+ B')+ B (A+ A')
   = AB+ AB'+ AB+ A'B
   = AB+ AB'+ A'B.
2. Y (A, B, C) = A+ ABC
   = A. (B+ B'). (C+ C')+ ABC
   = (AB+ AB'). (C+ C')+ ABC
   = ABC+ ABC'+ AB'C+ AB'C'+ ABC
   = ABC+ ABC'+ AB'C+ AB'C'
   = m_7+ m_6+ m_5+ m_4

$= \sum m\ (4, 5, 6, 7).$

3. Y(A,B,C)=A+BC

$= A.\ (B+ B').\ (C+ C')+(A+ A').\ BC$

$= (AB+ AB').\ (C+ C')+ ABC+ A'BC$

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

$= ABC+ ABC'+ AB'C+ AB'C'+ A'BC$

$= m_7+ m_6+ m_5+ m_4+ m_3$

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

4. Y (A, B, C) = AC+ AB+ BC

$= AC\ (B+ B')+ AB\ (C+ C')+ BC\ (A+ A')$

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

$= ABC+ AB'C+ ABC'+ A'BC$

$= \sum m\ (3, 5, 6, 7).$

5. Y (A, B, C, D) = AB+ ACD

$= AB\ (C+ C')\ (D+ D') + ACD\ (B+ B')$

$= (ABC+ ABC')\ (D+ D') + ABCD+ AB'CD$

$= \underline{ABCD}+ ABCD'+ ABC'D+ ABC'D'+ \underline{ABCD}+ AB'CD$

$= ABCD+ ABCD'+ ABC'D+ ABC'D'+ AB'CD.$

Canonical Product of sum expression:

If each term in POS form contains all literals then the POS is known as standard (or) Canonical POS form. Each individual term in standard POS form is called Maxterm canonical form.

- F (A, B, C) = (A+ B+ C). (A+ B'+ C). (A+ B+ C')
- F (x, y, z) = (x+ y'+ z'). (x'+ y+ z). (x+ y+ z)

Steps to convert general POS to standard POS form:

1. Find the missing literals in each sum term if any.
2. OR each sum term having missing literals by ANDing the literal and its complement.
3. Expand the term by applying distributive law and reorder the literals in the sum term.
4. Reduce the expression by omitting repeated sum terms if any.

Obtain the canonical POS expression of the functions:

1. Y= A+ B'C
   = (A+ B') (A+ C) [**A+ BC = (A+B) (A+C)**]
   = (A+ B'+ C.C') (A+ C+ B.B')
   = (A+ B'+C) (A+ B'+C') (A+ B+ C) (A+ B'+ C)
   = (A+ B'+C). (A+ B'+C'). (A+ B+ C)
   = M_2. M_3. M_0
   = ∏M (0, 2, 3)

2. Y= (A+B) (B+C) (A+C)
   = (A+B+ C.C') (B+ C+ A.A') (A+C+B.B')
   = (A+B+C) (A+B+C') (A+B+C) (A'+B+C) (A+B+C) (A+B'+C)
   = (A+B+C) (A+B+C') (A'+B+C) (A+B'+C)
   = M_0. M_1. M_4. M_2
   = ∏M (0, 1, 2, 4)

3. Y= A. (B+ C+ A)
   = (A+ B.B'+ C.C'). (A+ B+ C)
   = (A+B+C) (A+B+C') (A+B'+C) (A+ B'+C') (A+B+C)
   = (A+B+C) (A+B+C') (A+B'+C) (A+ B'+C')
   = M_0. M_1. M_2. M_3
   = ∏M (0, 1, 2, 3)

4. Y= (A+B') (B+C) (A+C')
   = (A+B'+C.C') (B+C+ A.A') (A+C'+ B.B')
   = (A+B'+C) (A+B'+C') (A+B+C) (A'+B+C) (A+B+C') (A+B'+C')
   = (A+B'+C) (A+B'+C') (A+B+C) (A'+B+C) (A+B+C')
   = M_2. M_3. M_0. M_4. M_1
   = ∏M (0, 1, 2, 3, 4)

5. Y= xy+ x'z
   = (xy+ x') (xy+ z)Using distributive law, convert the function into OR terms.
   = (x+x') (y+x') (x+z) (y+z) [x+ x'=1]
   = (x'+y) (x+z) (y+z)
   = (x'+y+ z.z') (x+z+y.y') (y+z+ x.x')

   = (x'+ y+ z) (x'+ y+ z') **(x+ y+ z)** (x+ y'+ z) **(x+ y+ z)** (x'+ y+ z)
   = (x'+ y+ z) (x'+ y+ z') (x+ y+ z) (x+ y'+ z)

$= M_4 . M_5 . M_0 . M_2$

$= \prod M\ (0, 2, 4, 5).$

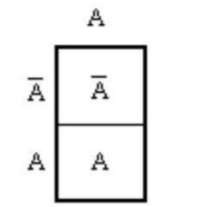
KARNAUGH MAP MINIMIZATION:

The simplification of the functions using Boolean laws and theorems becomes complex with the increase in the number of variables and terms. The map method, first proposed by Veitch and slightly improvised by Karnaugh, provides a simple, straightforward procedure for the simplification of Boolean functions. The method is called **Veitch diagram** or **Karnaugh map**, which may be regarded as a pictorial representation of a truth table.

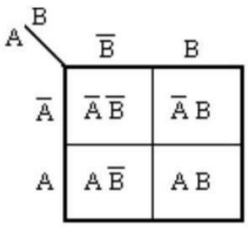
The Karnaugh map technique provides a systematic method for simplifying and manipulation of Boolean expressions. **A** K-map is a diagram made up of squares, with each square representing one minterm of the function that is to be minimized. For *n* variables on a Karnaugh map there are 2^n numbers of squares. Each square or cell represents one of the minterms. It can be drawn directly from either minterm (sum-of- products) or maxterm (product-of-sums) Boolean expressions.

Two- Variable, Three Variable and Four Variable Maps

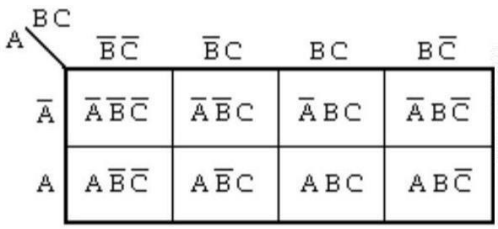
Karnaugh maps can be used for expressions with two, three, four and five variables. The number of cells in a Karnaugh map is equal to the total number of possible input variable combinations as is the number of rows in a truth table. For three variables, the number of cells is $2_3 = 8$. For four variables, the number of cells is $2_4 = 16$.



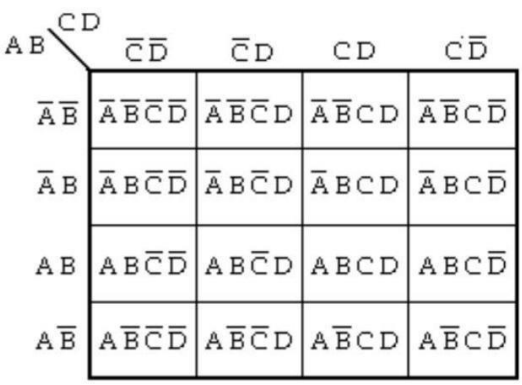
1 -Variable map



2 -Variable map



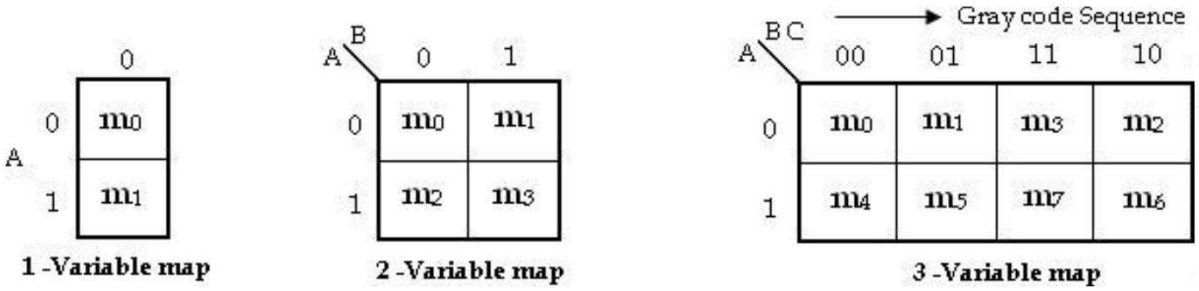
3 -Variable map

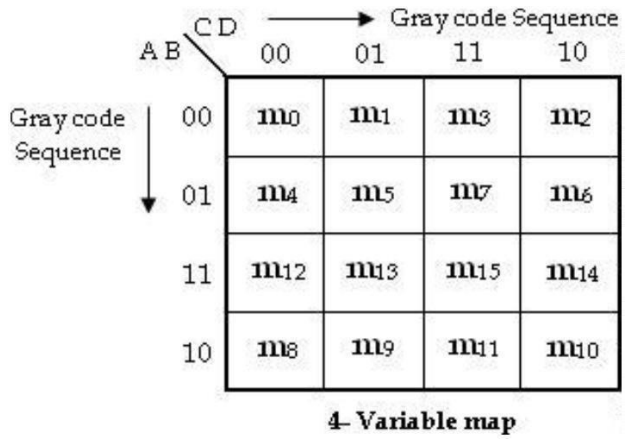


4 -Variable map

Product terms are assigned to the cells of a K-map by labeling each row and each column of a map with a variable, with its complement or with a combination of variables & complements. The below figure shows the way to label the rows & columns of a 1, 2, 3 and 4- variable maps and the product terms corresponding to each cell.

It is important to note that when we move from one cell to the next along any row or from one cell to the next along any column, one and only one variable in the product term changes (to a complement or to an uncomplemented form). Irrespective of number of variables the labels along each row and column must conform to a single change. Hence gray code is used to label the rows and columns of K-map as shown ow.



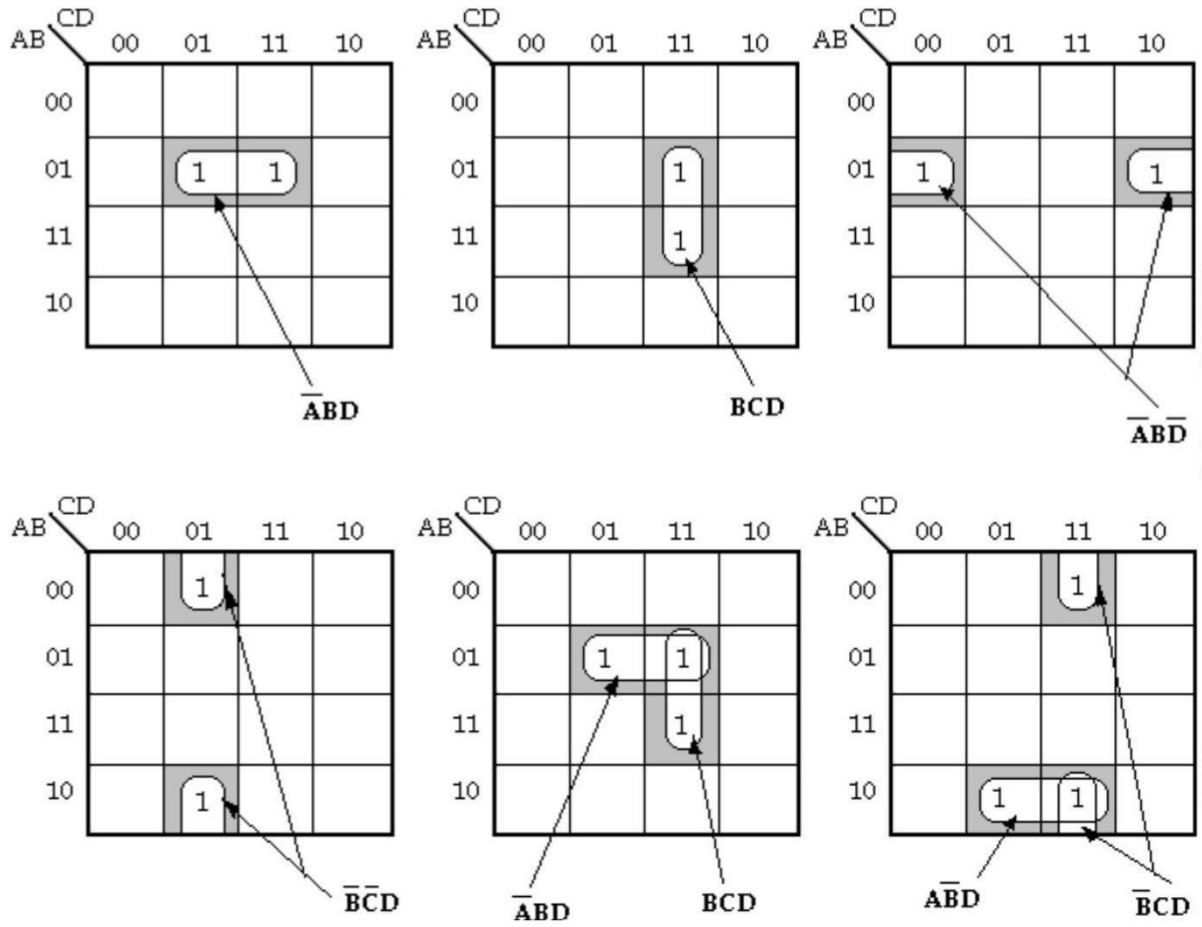


Grouping cells for Simplification:

The grouping is nothing but combining terms in adjacent cells. The simplification is achieved by grouping adjacent 1's or 0's in groups of 2i, where i = 1, 2, …, n and n is the number of variables. When adjacent 1's are grouped then we get result in the sum of product form; otherwise we get result in the product of sum form.

Grouping Two Adjacent 1's: (Pair)

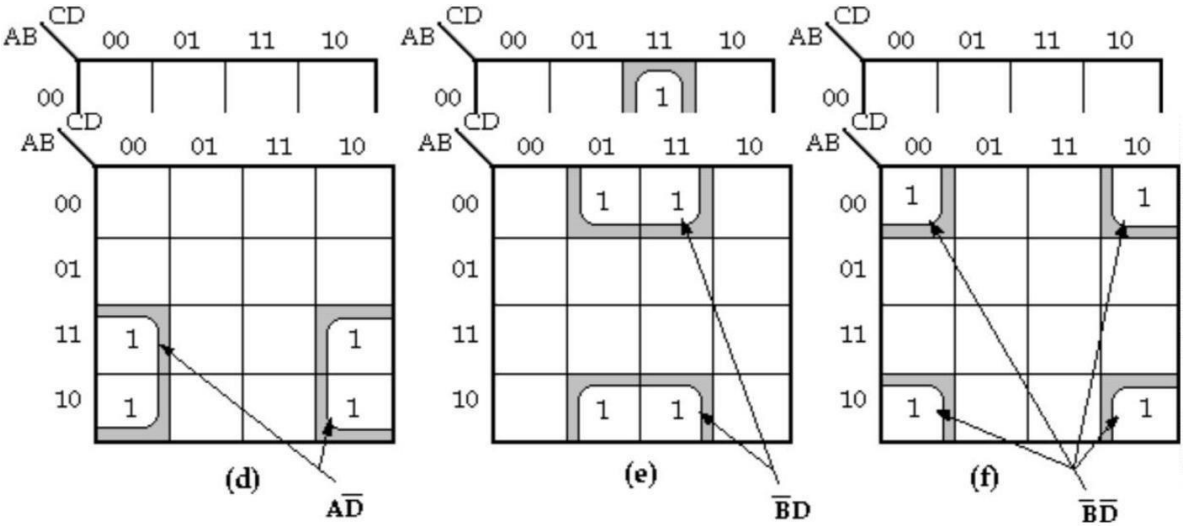
In a Karnaugh map we can group two adjacent 1's. The resultant group is called Pair.



Examples of Pairs

Grouping Four Adjacent 1's: (Quad)

In a Karnaugh map we can group four adjacent 1's. The resultant group is called Quad. Fig (a) shows the four 1's are horizontally adjacent and Fig (b) shows they are vertically adjacent. Fig (c) contains four 1's in a square, and they are considered adjacent to each other.

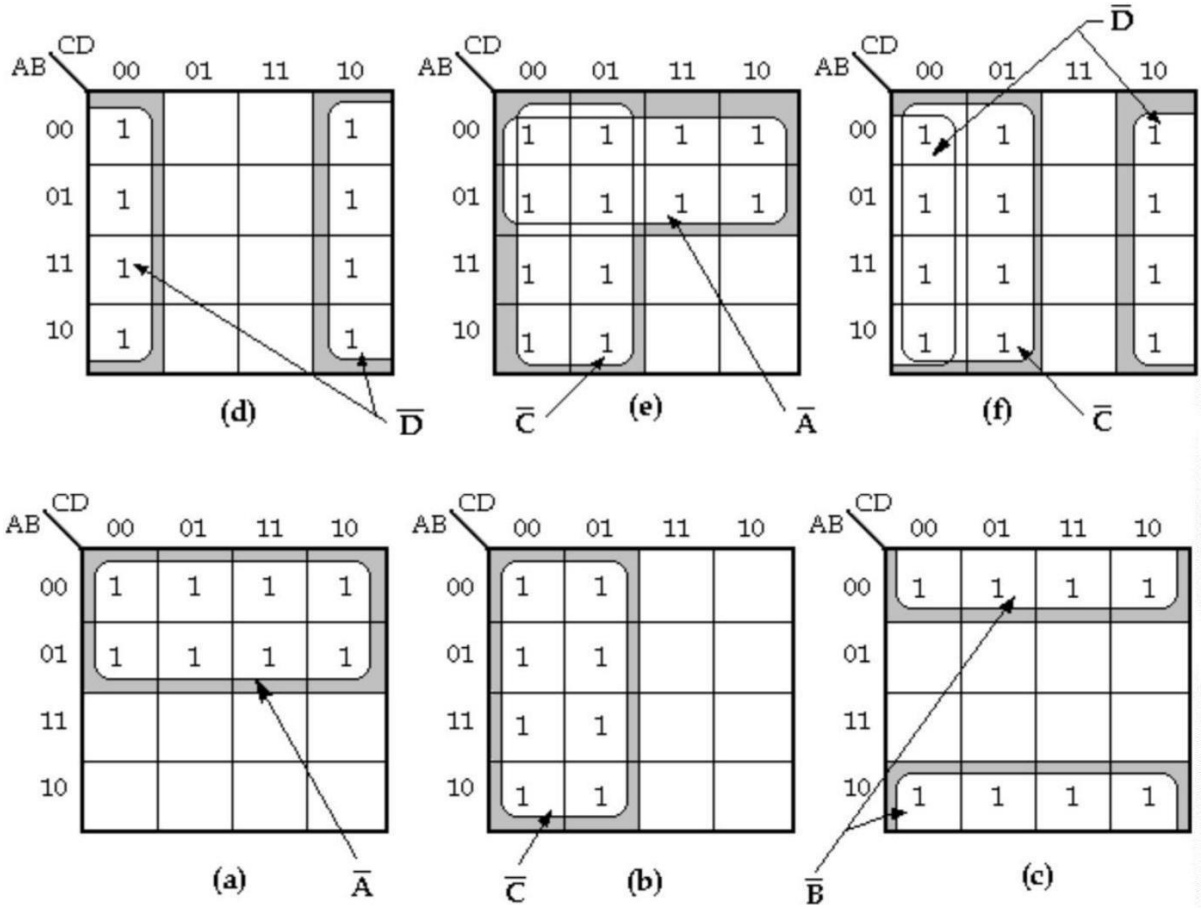

AB
CD
00
01
11
10
1
(d)
$A\overline{D}$
(e)
$\overline{B}D$
(f)
$\overline{B}\overline{D}$

Examples of Quads

The four 1's in fig (d) and fig (e) are also adjacent, as are those in fig (f) because, the top and bottom rows are considered to be adjacent to each other and the leftmost and rightmost columns are also adjacent to each other.

Grouping Eight Adjacent 1's: (Octet)

In a Karnaugh map we can group eight adjacent 1's. The resultant group is called Octet.



Simplification of Sum of Products Expressions: (Minimal Sums)

The generalized procedure to simplify Boolean expressions as follows:

1. Plot the K-map and place 1's in those cells corresponding to the 1's in the sum of product expression. Place 0's in the other cells.
2. Check the K-map for adjacent 1's and encircle those 1's which are not adjacent to any other 1's. These are called **isolated 1's**.
3. Check for those 1's which are adjacent to only one other 1 and encircle such **pairs**.

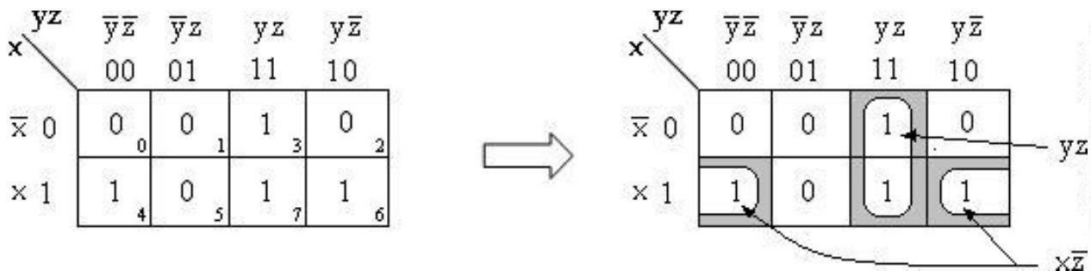
4. Check for **quads** and **octets** of adjacent 1's even if it contains some 1's that have already been encircled. While doing this make sure that there are minimum number of groups.
5. Combine any pairs necessary to include any 1's that have not yet been grouped.
6. Form the simplified expression by summing product terms of all the groups.

Three- Variable Map:

1. Simplify the Boolean expression,

F(x, y, z) = ∑m (3, 4, 6, 7).

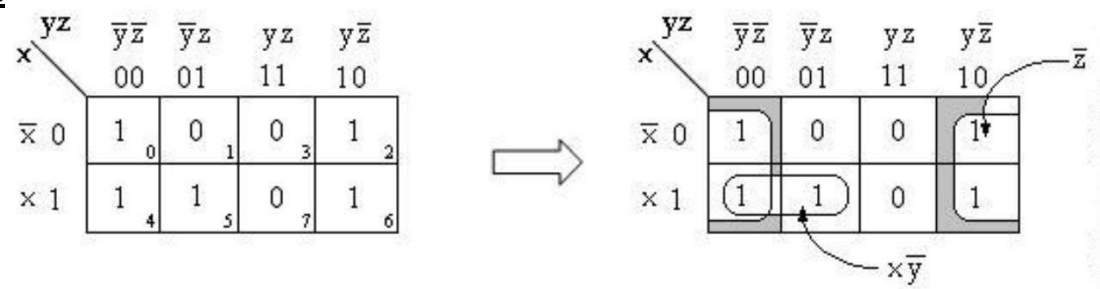
Soln:



F = yz+ xz'

2. **F(x, y, z) = ∑m (0, 2, 4, 5, 6).**

Soln:



F = z'+ xy'

3. F=A'C+A'B+AB'C+BC

Soln:

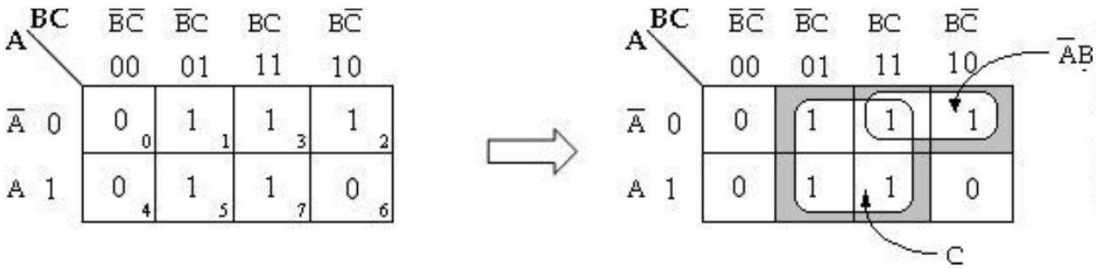
= A'C (B+ B') + A'B (C+ C') + AB'C + BC (A+ A')

= A'BC+ A'B'C + A'BC + A'BC' + AB'C + ABC + A'BC

= A'BC+ A'B'C + A'BC' + AB'C + ABC

= m_3+ m_1+ m_2+ m_5+ m_7

= ∑ m (1, 2, 3, 5, 7)



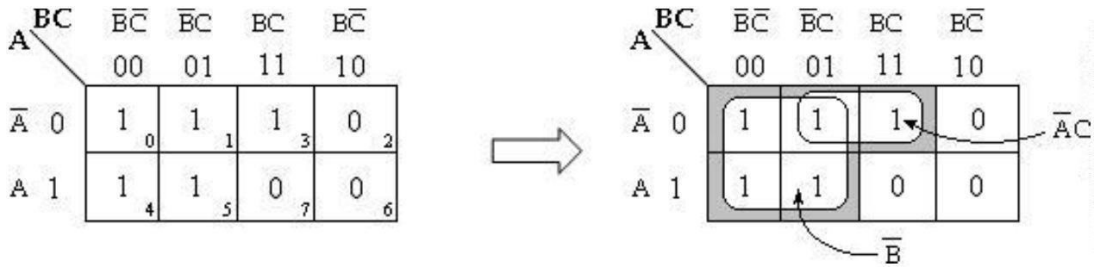
F=C+A'B

4. AB'C + A'B'C + A'BC + AB'C' + A'B'C'

Soln:

= m5 + m1 + m3 + m4 + m0

= ∑ m (0, 1, 3, 4, 5)



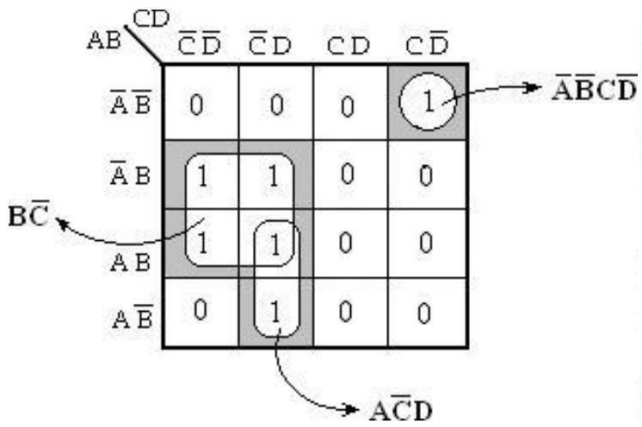
F=A'C+B'

Four - Variable Map:

1. Simplify the Boolean expression,

Y = A'BC'D' + A'BC'D + ABC'D' + ABC'D + AB'C'D + A'B'CD'

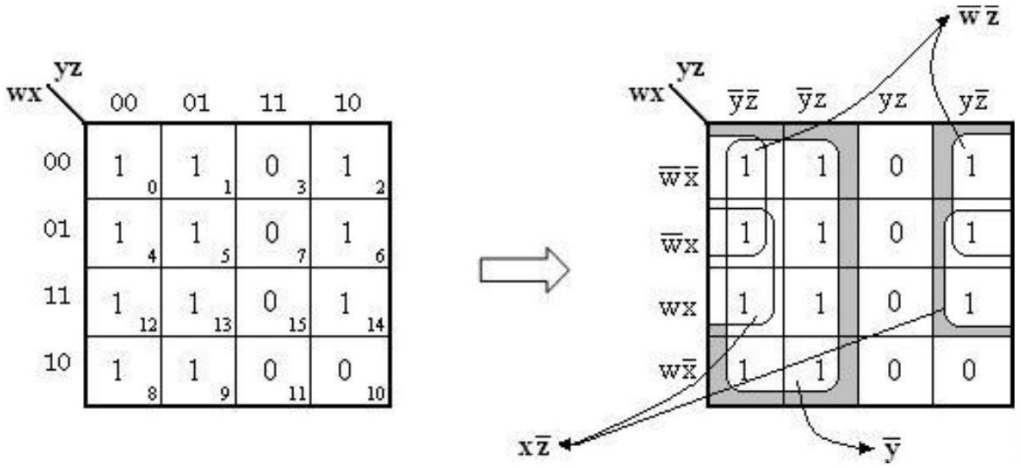
Soln:



Therefore, Y= A'B'CD'+ AC'D+ BC'

2. **F (w, x, y, z) = ∑ m(0, 1, 2, 4, 5, 6, 8, 9, 12, 13, 14)**

Soln:



Therefore,

F= y'+ w'z'+ xz'

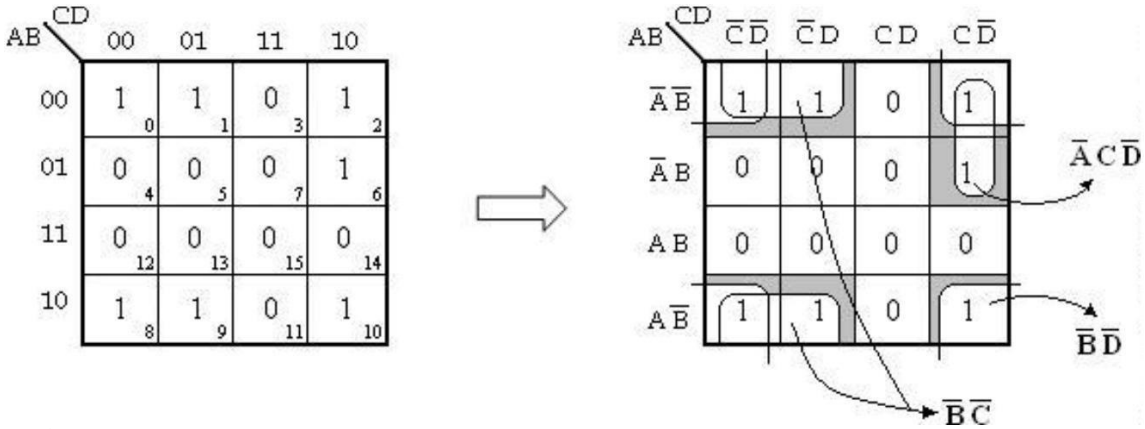
3. **F= A'B'C'+ B'CD'+ A'BCD'+ AB'C'**

= A'B'C' (D+ D') + B'CD' (A+ A') + A'BCD'+ AB'C' (D+ D')

= A'B'C'D+ A'B'C'D'+ AB'CD'+ A'B'CD'+ A'BCD'+ AB'C'D+ AB'C'D'

= m_1+ m_0+ m_{10}+ m_2+ m_6+ m_9+ m_8

= **∑ m (0, 1, 2, 6, 8, 9, 10)**



Therefore,

F= B'D'+ B'C'+ A'CD'.

4. Y= ABCD+ AB'C'D'+ AB'C+ AB

= ABCD+ AB'C'D'+ AB'C (D+D')+ AB (C+C') (D+D')

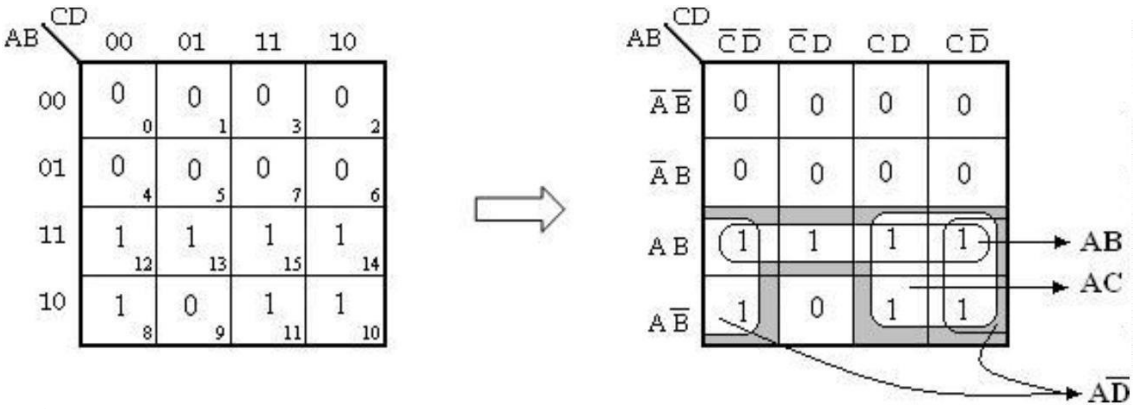
= ABCD+ AB'C'D'+ AB'CD+ AB'CD'+ (ABC+ ABC') (D+ D')

= ABCD+ AB'C'D'+ AB'CD+ AB'CD'+ ABCD+ ABCD'+ ABC'D+ ABC'D'

= ABCD+ AB'C'D'+ AB'CD+ AB'CD'+ ABCD'+ ABC'D+ ABC'D'

= $m_{15}+ m_8+ m_{11}+ m_{10}+ m_{14}+ m_{13}+ m_{12}$

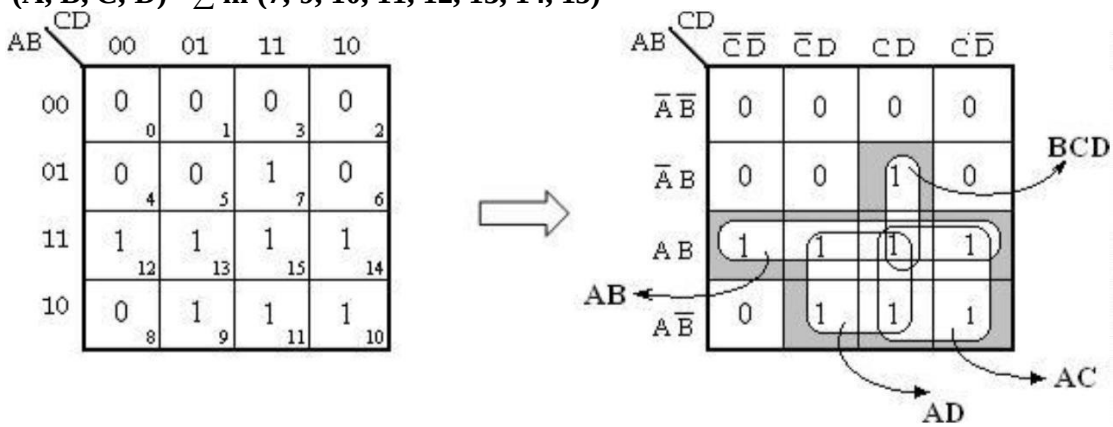
= **$\sum$ m (8, 10, 11, 12, 13, 14, 15)**



Therefore,

Y= AB+ AC+ AD'.

5. **Y (A, B, C, D)= $\sum$ m (7, 9, 10, 11, 12, 13, 14, 15)**



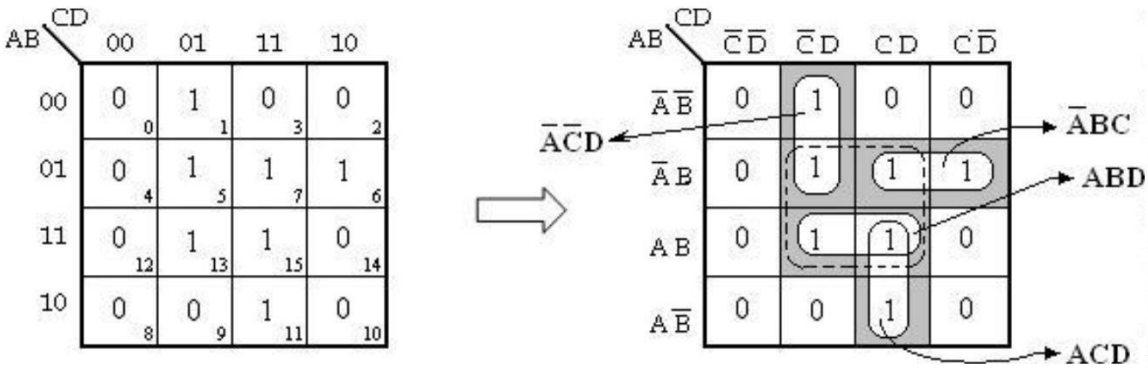
Therefore,

Y= AB+ AC+ AD+BCD.

6. Y= A'B'C'D+ A'BC'D+ A'BCD+ A'BCD'+ ABC'D+ ABCD+ AB'CD

= m_1+ m_5+ m_7+ m_6+ m_{13}+ m_{15}+ m_{11}

= $\sum$ **m (1, 5, 6, 7, 11, 13, 15)**

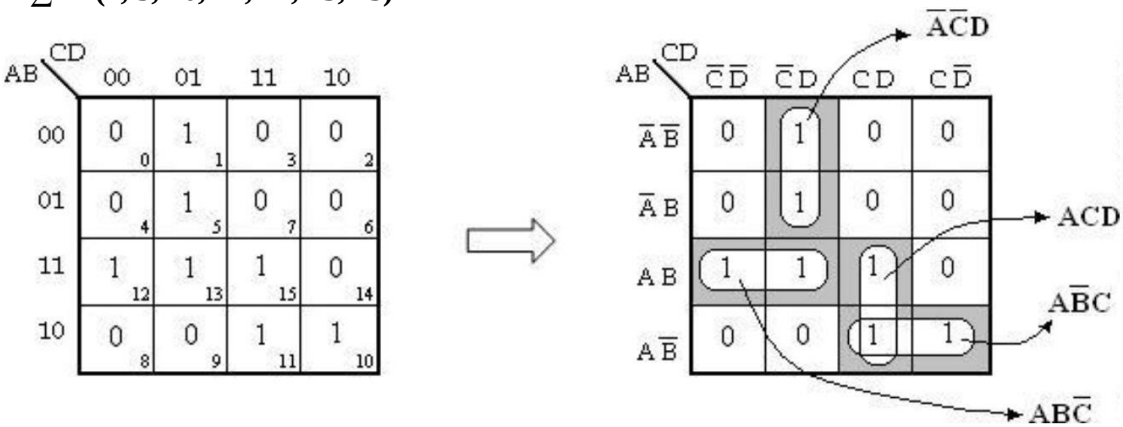


In the above K-map, the cells 5, 7, 13 and 15 can be grouped to form a quad as indicated by the dotted lines. In order to group the remaining 1's, four pairs have to be formed. However, all the four 1's covered by the quad are also covered by the pairs. So, the quad in the above k-map is redundant.

Therefore, the simplified expression will be,

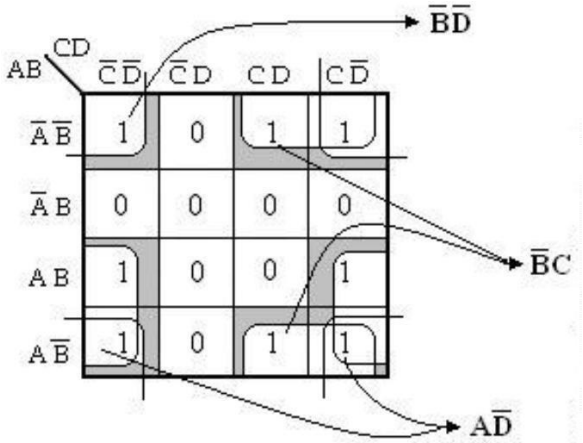
Y = A'C'D+ A'BC+ ABD+ ACD.

7. **Y= $\sum$ m (1, 5, 10, 11, 12, 13, 15)**



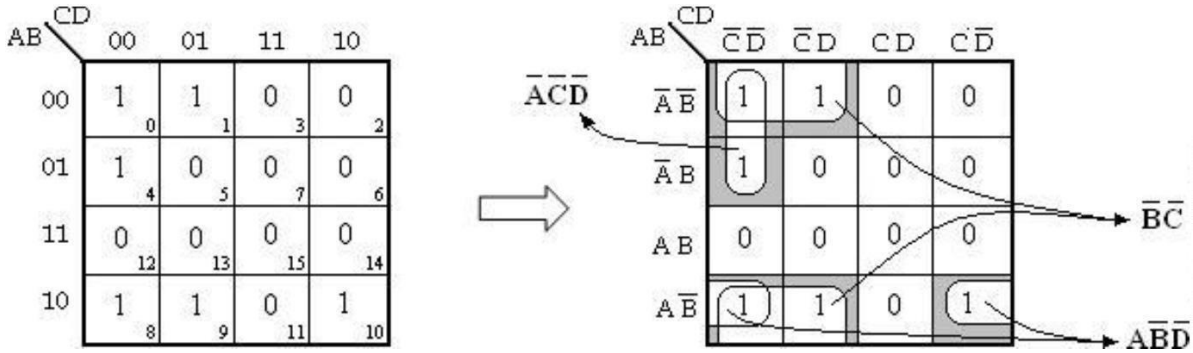
Therefore, **Y= A'C'D+ ABC'+ ACD+ AB'C.**

8. **Y= A'B'CD'+ ABCD'+ AB'CD'+ AB'CD+ AB'C'D'+ ABC'D'+ A'B'CD+ A'B'C'D'**



Therefore, **Y= AD'+ B'C+ B'D'**

9. **F (A, B, C, D) = ∑ m (0, 1, 4, 8, 9, 10)**



Therefore, **F= A'C'D'+ AB'D'+ B'C'.**

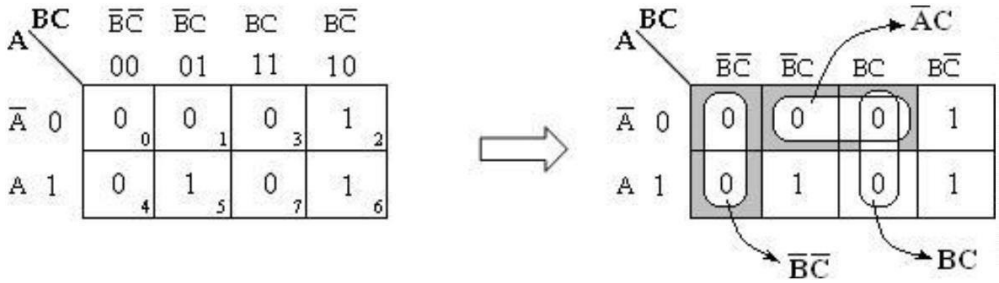
Simplification of Sum of Products Expressions: (Minimal Sums)

1. **Y= (A+ B+ C') (A+ B'+ C') (A'+ B'+ C') (A'+ B+ C) (A+ B+ C)**

= $M_1 . M_3 . M_7 . M_4 . M_0$

= ∏ M (0, 1, 3, 4, 7)

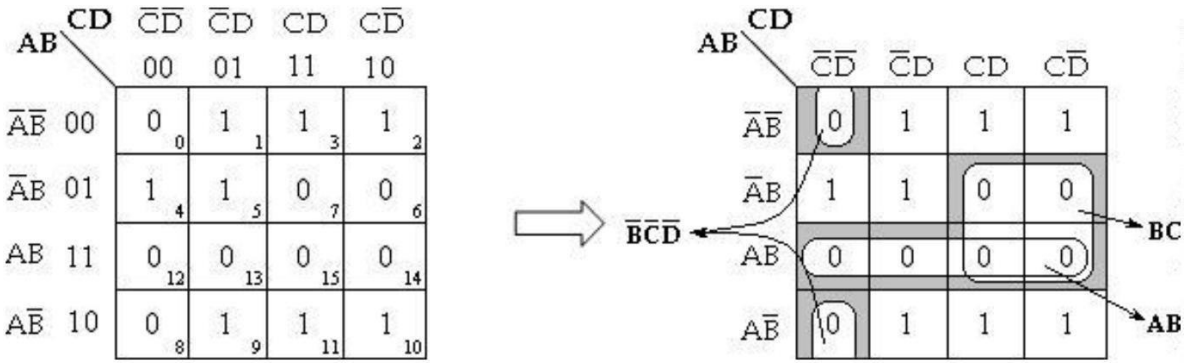
= ∑ m (2, 5, 6)



Y' = B'C'+ A'C+ BC.

Y= Y" = (B'C'+ A'C+ BC)'
= (B'C')'. (A'C)'. (BC)'
= (B"+ C"). (A"+C'). (B'+ C')
Y = (B+ C). (A+C'). (B'+ C')

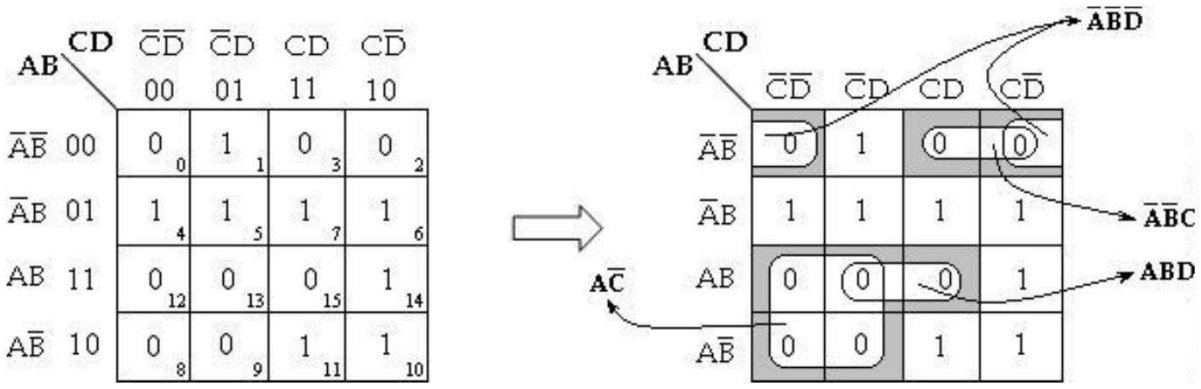
2. **Y= (A'+ B'+ C+ D) (A'+ B'+ C'+ D) (A'+ B'+ C'+ D') (A'+ B+ C+ D) (A+ B'+ C'+ D) (A+ B'+ C'+ D') (A+ B+ C+ D) (A'+ B'+ C+ D')**
= $M_{12}. M_{14}. M_{15}. M_8. M_6. M_7. M_0. M_{13}$
= ∏M (0, 6, 7, 8, 12, 13, 14, 15)



Y' = B'C'D'+ AB+ BC

Y= Y" = (B'C'D'+ AB+ BC)'
= (B'C'D')'. (AB)'. (BC)'
= (B"+ C"+D"). (A'+B'). (B'+ C')
= (B+ C+ D). (A'+ B'). (B'+ C')
Therefore, **Y= (B+ C+ D). (A'+ B'). (B'+ C')**

3. **F(A, B, C, D)**= ∏M (0, 2, 3, 8, 9, 12, 13, 14, 15)

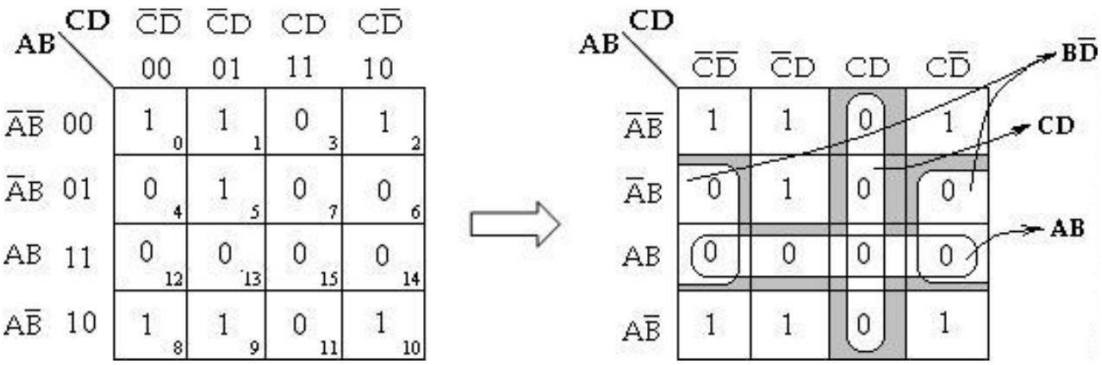


Y' = A'B'D'+ A'B'C+ ABD+ AC'

Y= Y" = (A'B'D'+ A'B'C+ ABD+ AC')'
= (A'B'D')'. (A'B'C)'. (ABD)'. (AC')'
= (A"+ B"+ D"). (A"+ B"+C'). (A'+ B'+ D'). (A'+ C")
= **(A+ B+ D). (A+ B+ C'). (A'+ B'+ D'). (A'+ C)**

Therefore, **Y= (A+ B+ D). (A+ B+ C'). (A'+ B'+ D'). (A'+ C)**

4. **F(A, B, C, D)**= ∑m (0, 1, 2, 5, 8, 9, 10)
= ∏M (3, 4, 6, 7, 11, 12, 13, 14, 15)



Y' = BD'+ CD+ AB

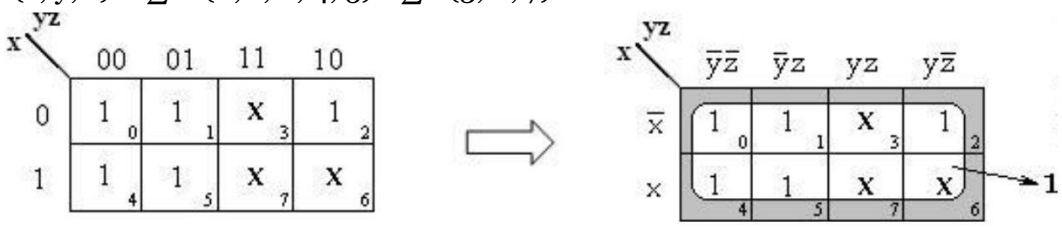
Y= Y" = (BD'+ CD+ AB)'
= (BD')'. (CD)'. (AB)'
= (B'+ D"). (C'+ D'). (A'+ B')
= **(B'+ D). (C'+ D'). (A'+ B')**

Therefore, **Y= (B'+ D). (C'+ D'). (A'+ B')**

Don't care Conditions:

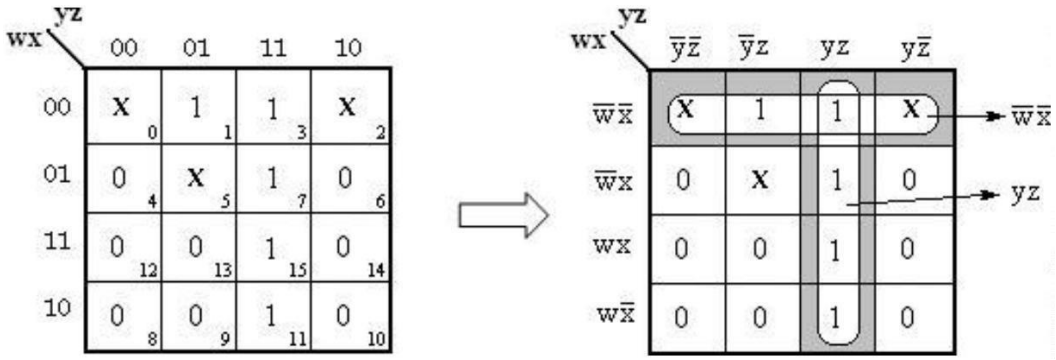
A don't care minterm is a combination of variables whose logical value is not specified. When choosing adjacent squares to simplify the function in a map, the don't care minterms may be assumed to be either 0 or 1. When simplifying the function, we can choose to include each don't care minterm with either the 1's or the 0's, depending on which combination gives the simplest expression.

1. F (x, y, z) = ∑m (0, 1, 2, 4, 5)+ ∑d (3, 6, 7)



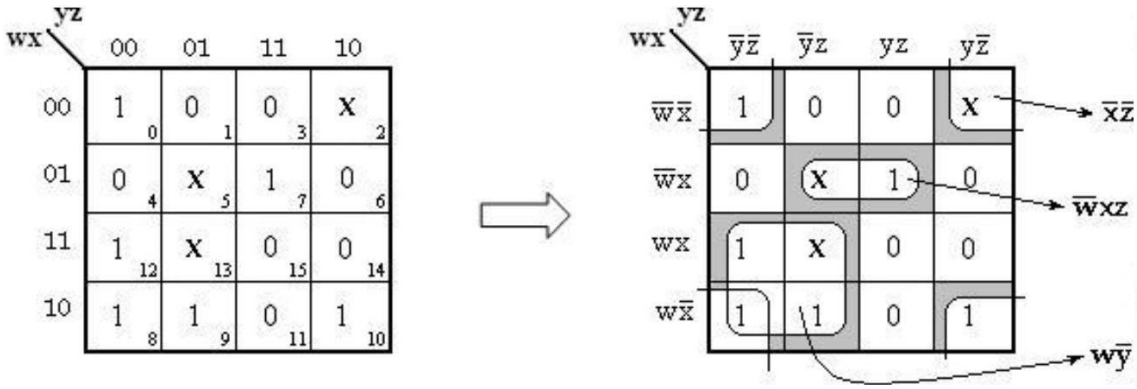
F (x, y, z) = 1

2. F (w, x, y, z) = ∑m (1, 3, 7, 11, 15)+ ∑d (0, 2, 5)



F (w, x, y, z) = w'x'+ yz

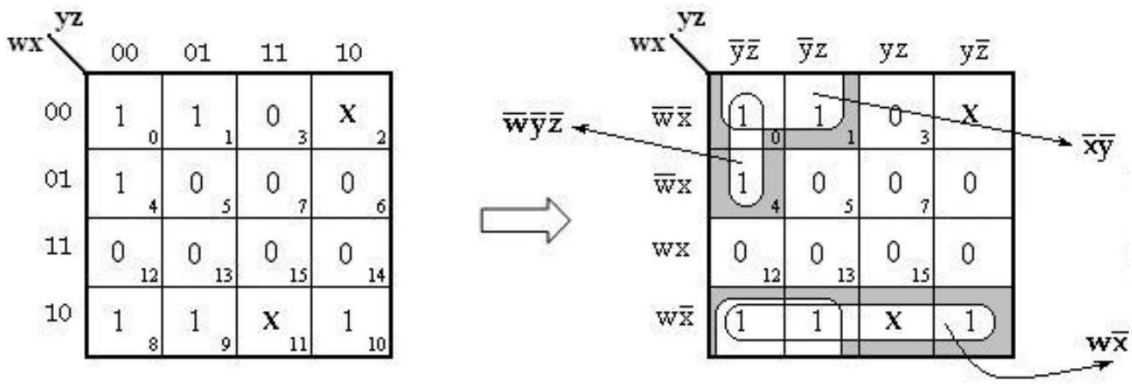
3. F (w, x, y, z) = ∑m (0, 7, 8, 9, 10, 12)+ ∑d (2, 5, 13)



F (w, x, y, z) = w'xz+ wy'+ x'z'.

4. F (w, x, y, z) = ∑m (0, 1, 4, 8, 9, 10)+ ∑d (2, 11)

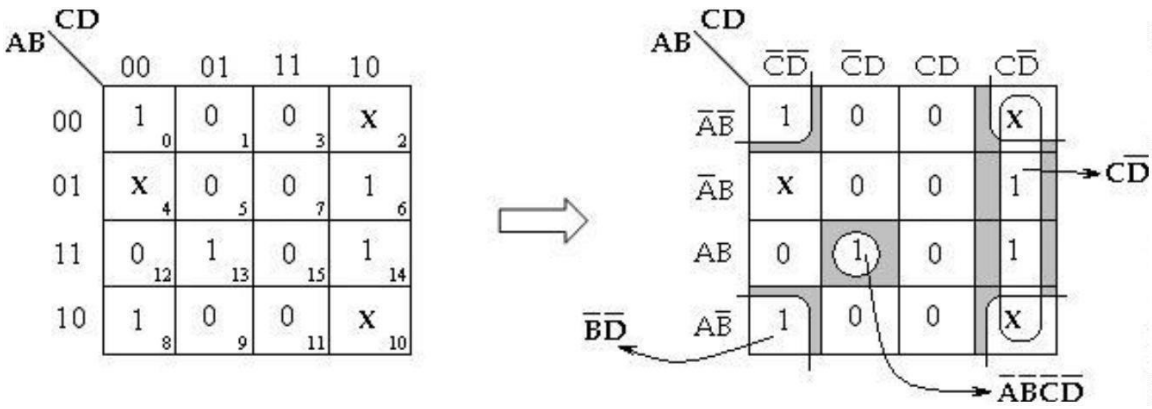
Soln:



F (w, x, y, z) = wx'+ x'y'+ w'y'z'.

5. **F(A, B, C, D)** = ∑m (0, 6, 8, 13, 14)+ ∑d (2, 4, 10)

Soln:



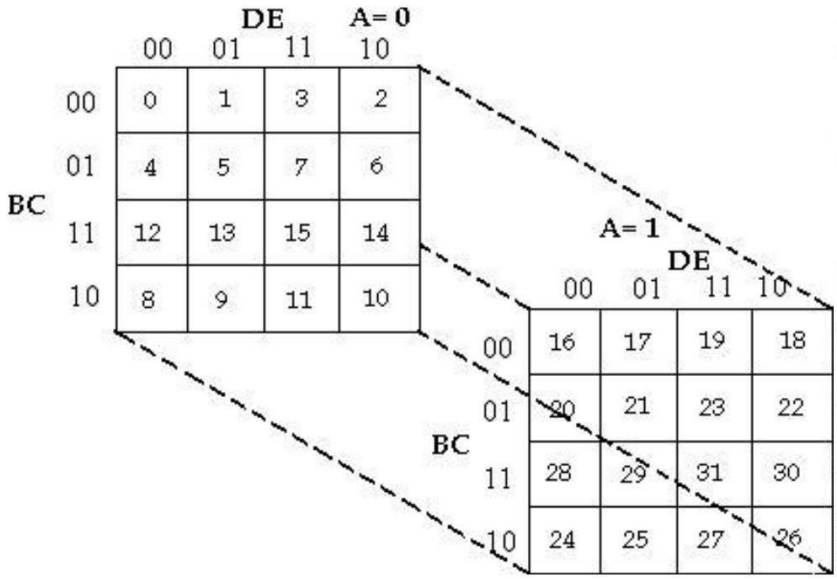
F(A, B, C, D) = CD'+ B'D'+ A'B'C'D'.

Five- Variable Maps:

A 5- variable K- map requires 25= 32 cells, but adjacent cells are difficult to identify on a single 32-cell map. Therefore, two 16 cell K-maps are used.

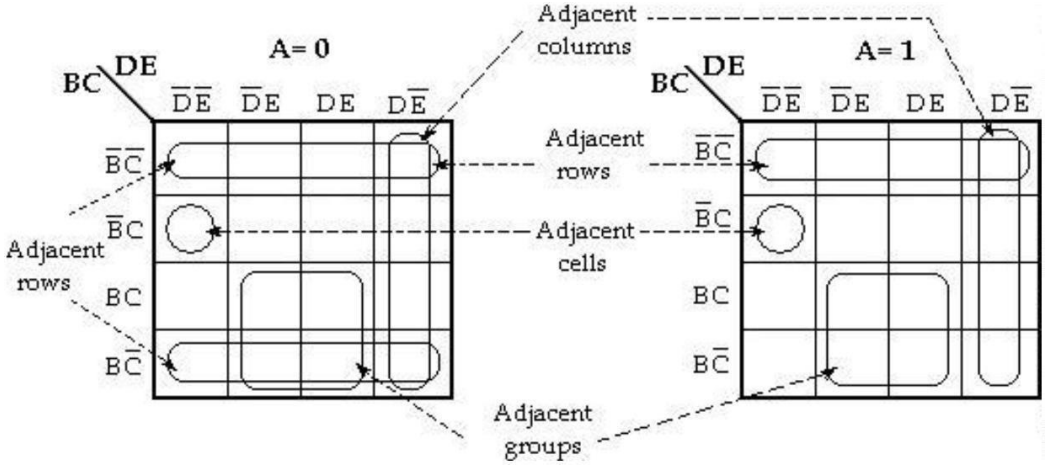
If the variables are A, B, C, D and E, two identical 16- cell maps containing B, C, D and E can be constructed. One map is used for A and other for A'.

In order to identify the adjacent grouping in the 5- variable map, we must imagine the two maps superimposed on one another ie., every cell in one map is adjacent to the corresponding cell in the other map, because only one variable changes between such corresponding cells.



Five- Variable Karnaugh map (Layer Structure)

Thus, every row on one map is adjacent to the corresponding row (the one occupying the same position) on the other map, as are corresponding columns. Also, the rightmost and leftmost columns within each 16- cell map are adjacent, just as they are in any 16- cell map, as are the top and bottom rows.



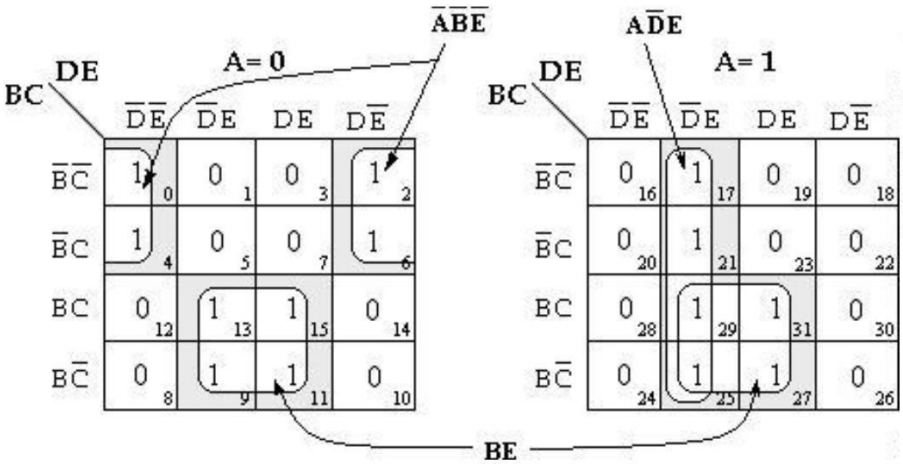
Typical subcubes on a five-variable map

However, the rightmost column of the map is not adjacent to the leftmost column of the other map.

1. Simplify the Boolean function

F (A, B, C, D, E) = ∑m (0, 2, 4, 6, 9, 11, 13, 15, 17, 21, 25, 27, 29, 31)

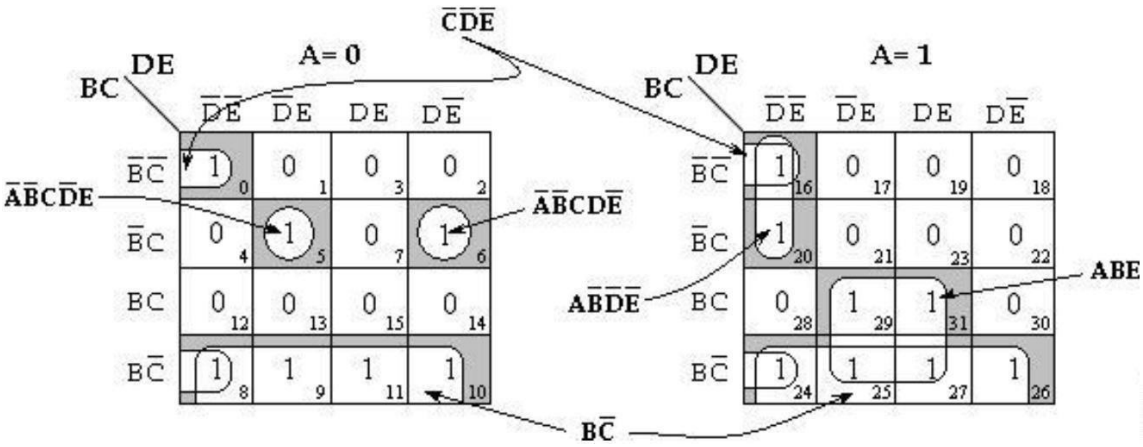
Soln:



F (A, B, C, D, E) = A'B'E'+ BE+ AD'E

2. F (A, B, C, D, E) = ∑m (0, 5, 6, 8, 9, 10, 11, 16, 20, 24, 25, 26, 27, 29, 31)

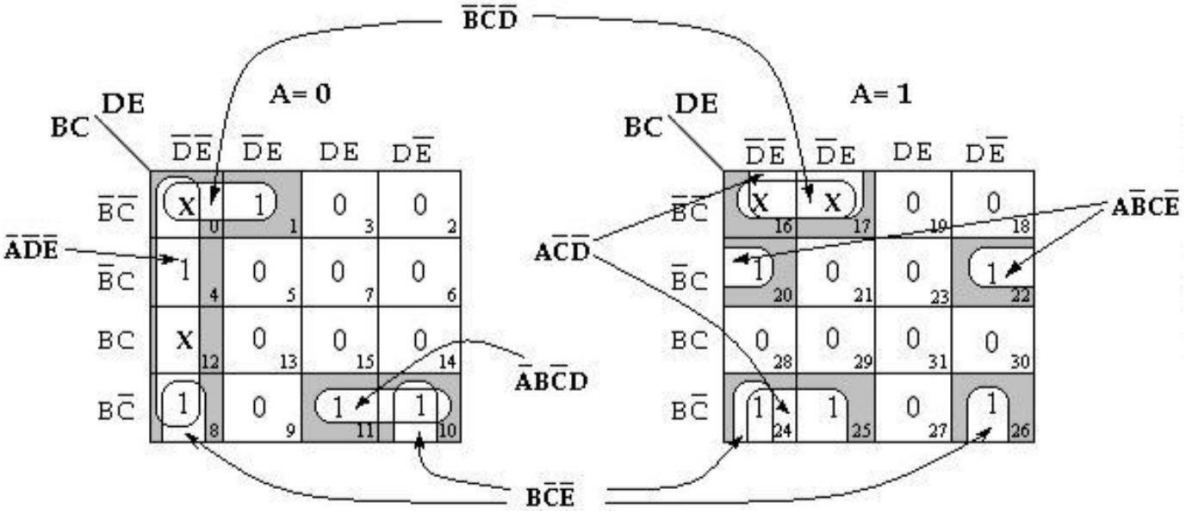
Soln:



F (A, B, C, D, E) = C'D'E'+ A'B'CD'E+ A'B'CDE'+ AB'D'E'+ ABE+ BC'

3. F (A, B, C, D, E) = ∑m (1, 4, 8, 10, 11, 20, 22, 24, 25, 26)+∑d (0, 12, 16, 17)

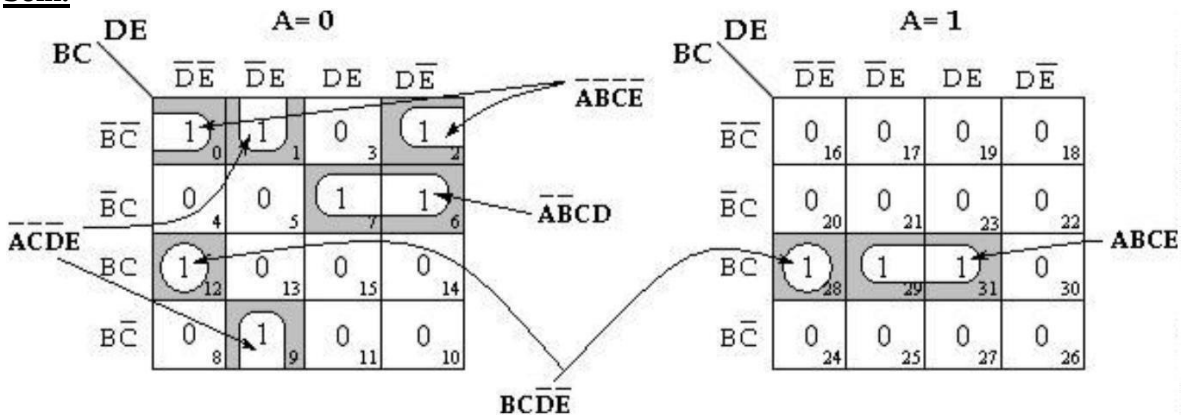
Soln:



F (A, B, C, D, E) = B'C'D'+ A'D'E'+ BC'E'+ A'BC'D+ AC'D'+ AB'CE'

4. F (A, B, C, D, E) = ∑m (0, 1, 2, 6, 7, 9, 12, 28, 29, 31)

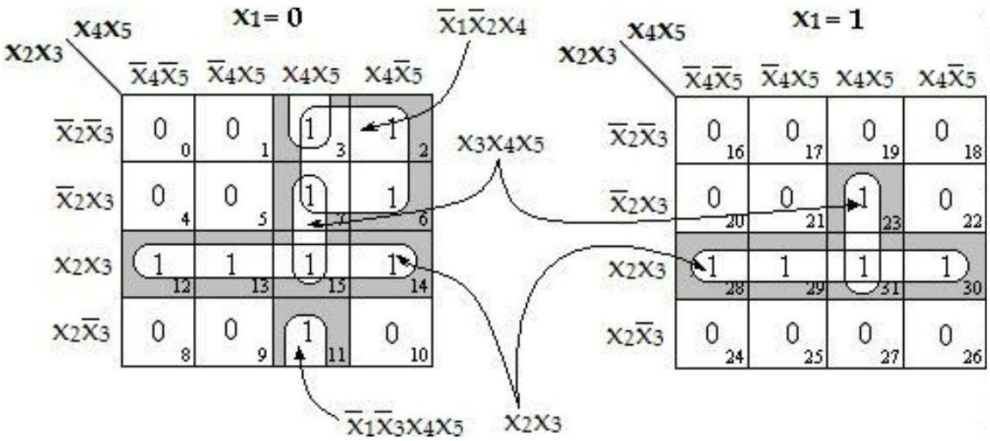
Soln:



F (A, B, C, D, E) = BCD'E'+ ABCE+ A'B'C'E'+ A'C'D'E+ A'B'CD

5. F (x_1, x_2, x_3, x_4, x_5) = ∑m (2, 3, 6, 7, 11, 12, 13, 14, 15, 23, 28, 29, 30, 31)

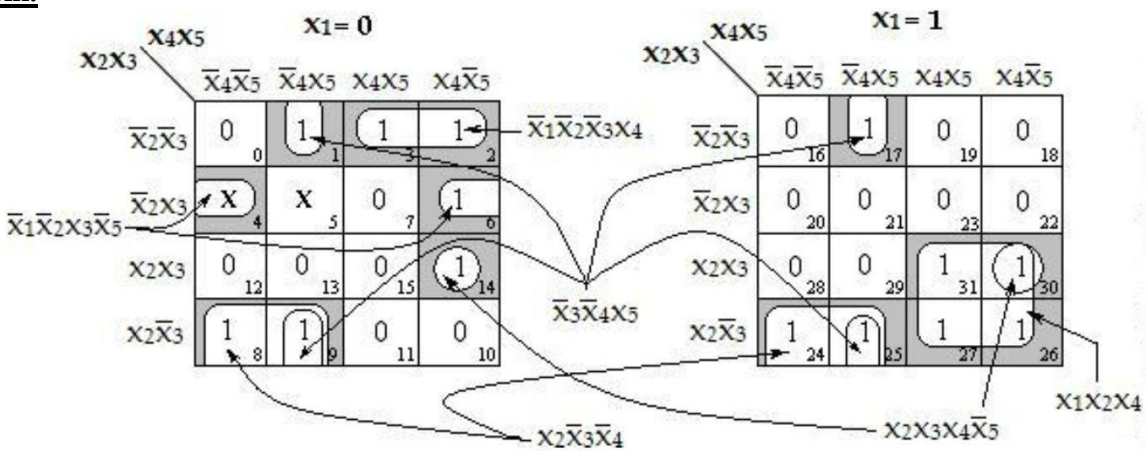
Soln:



F (x_1, x_2, x_3, x_4, x_5) = x_2x_3+ $x_3x_4x_5$+ $x_1'x_2'x_4$+ $x_1'x_3'x_4x_5$

6. F (x_1, x_2, x_3, x_4, x_5) = ∑m (1, 2, 3, 6, 8, 9, 14, 17, 24, 25, 26, 27, 30, 31)+ ∑d (4, 5)

Soln:



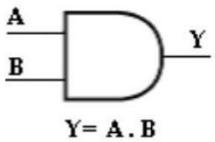
F (x_1, x_2, x_3, x_4, x_5) = $x_2x_3'x_4' + x_2x_3x_4x_5' + x_3'x_4'x_5 + x_1x_2x_4 + x_1'x_2'x_3x_5' + x_1'x_2'x_3'x_4$

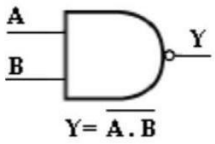
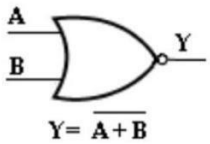
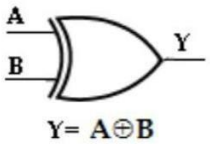
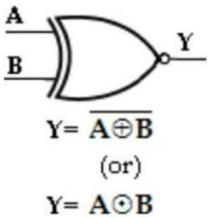
LOGIC GATES

BASIC LOGIC GATES:

Logic gates are electronic circuits that can be used to implement the most elementary logic expressions, also known as Boolean expressions. The logic gate is the most basic building block of combinational logic.

There are three basic logic gates, namely the OR gate, the AND gate and the NOT gate. Other logic gates that are derived from these basic gates are the NAND gate, the NOR gate, the EXCLUSIVE- OR gate and the EXCLUSIVE-NOR gate.

GATE	SYMBOL	OPERATION	TRUTH TABLE
NOT (7404)		NOT gate (Invertion), produces an inverted output pulse for a given input pulse.	A | Y= $\bar{A}$ 0 | 1 1 | 0
AND (7408)		AND gate performs logical **multiplication**. The output is HIGH only when all the inputs are HIGH. When any of the inputs are low, the output is LOW.	A | B | Y= A . B 0 | 0 | 0 0 | 1 | 0 1 | 0 | 0 1 | 1 | 1

OR (7432)		OR gate performs logical **addition**. It produces a HIGH on the output when any of the inputs are HIGH. The output is LOW only when all inputs are LOW.	A B Y= A+B 0 0 0 0 1 1 1 0 1 1 1 1
NAND (7400)		It is a universal gate. When any of the inputs are LOW, the output will be HIGH. LOW output occurs only when all inputs are HIGH.	A B Y= $\overline{A.B}$ 0 0 1 0 1 1 1 0 1 1 1 0
NOR (7402)		It is a universal gate. LOW output occurs when any of its input is HIGH. When all its inputs are LOW, the output is HIGH.	A B Y= $\overline{A+B}$ 0 0 1 0 1 0 1 0 0 1 1 0
EX- OR (7486)		The output is HIGH only when odd number of inputs is HIGH.	A B Y= A⊕B 0 0 0 0 1 1 1 0 1 1 1 0
EX- NOR		The output is HIGH only when even number of inputs is HIGH. Or when all inputs are zeros.	A B Y= A⊙B 0 0 1 0 1 0 1 0 0 1 1 1

UNIVERSAL GATES:

The NAND and NOR gates are known as universal gates, since any logic function can be implemented using NAND or NOR gates. This is illustrated in the following sections.

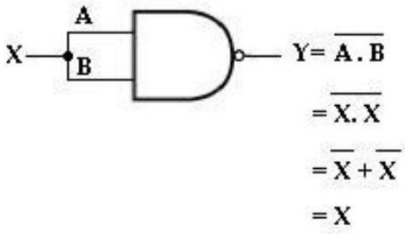
a) **NAND Gate:**

The NAND gate can be used to generate the NOT function, the AND function,

the OR function and the NOR function.

i) NOT function:

By connecting all the inputs together and creating a single common input.



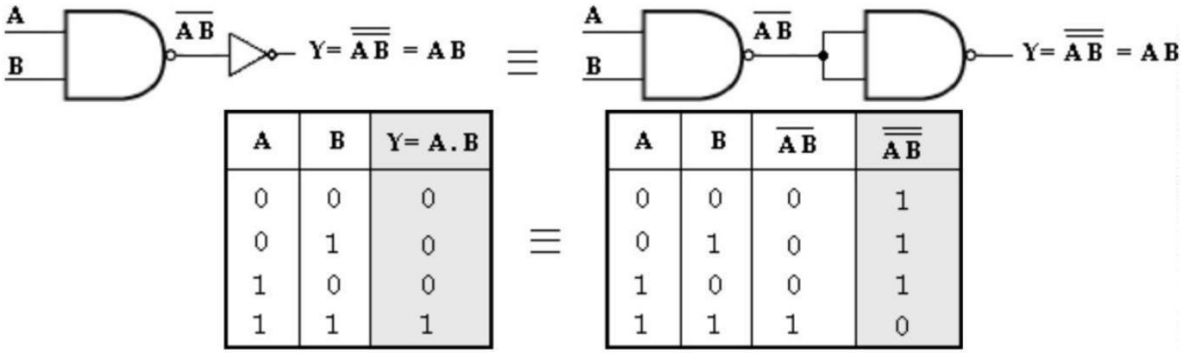
	A	B	Y= $\overline{A.B}$	
X= 0	0	0	1	Y= 1
	0	1	1	
	1	0	1	
X= 1	1	1	0	Y= 0

NOT function using NAND gate

ii) AND function:

By simply inverting output of the NAND gate. i.e.,

$\overline{\overline{AB}} = AB$



A	B	Y= A . B
0	0	0
0	1	0
1	0	0
1	1	1

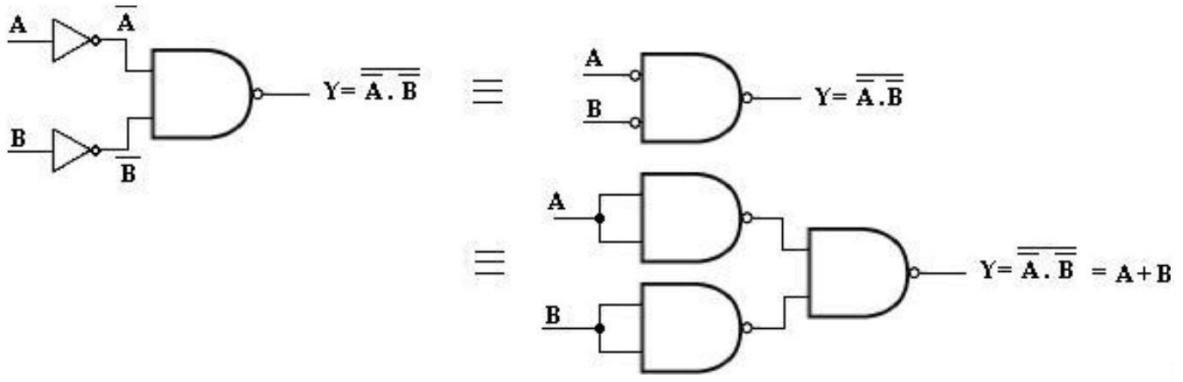
≡

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

AND function using NAND gates

iii) OR function:

By simply inverting inputs of the NAND gate. i.e.,



OR function using NAND gates

Bubble at the input of NAND gate indicates inverted input.

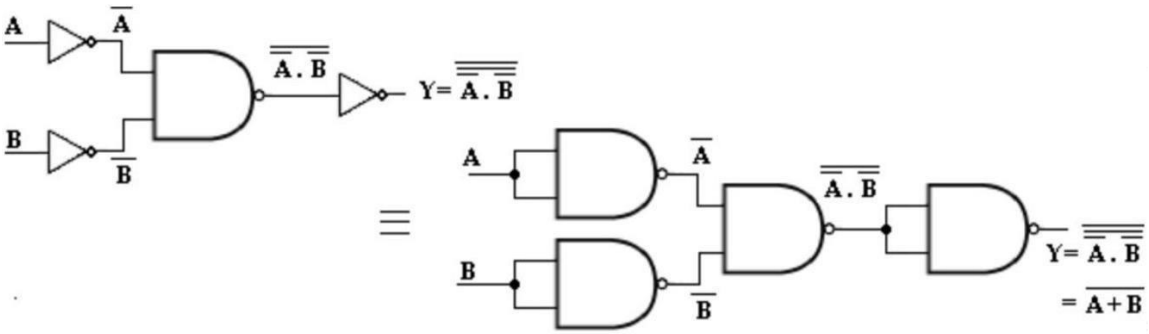
A	B	Y= A+B
0	0	0
0	1	1
1	0	1
1	1	1

≡

A	B	$\overline{A}.\overline{B}$	$\overline{\overline{A}.\overline{B}}$
0	0	1	0
0	1	0	1
1	0	0	1
1	1	0	1

iv) NOR function**:**

By inverting inputs and outputs of the NAND gate.



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

≡

A	B	$\overline{A}.\overline{B}$	$\overline{\overline{A}.\overline{B}}$	$\overline{\overline{\overline{A}.\overline{B}}}$
0	0	1	0	1
0	1	0	1	0
1	0	0	1	0
1	1	0	1	0

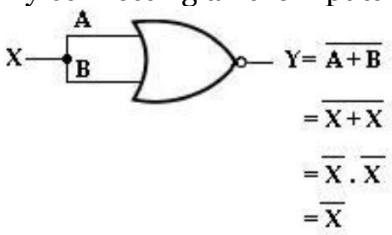
NOR function using NAND gates

b) **NOR Gate**:

Similar to NAND gate, the NOR gate is also a universal gate, since it can be used to generate the NOT, AND, OR and NAND functions.

i) NOT function:

By connecting all the inputs together and creating a single common input.



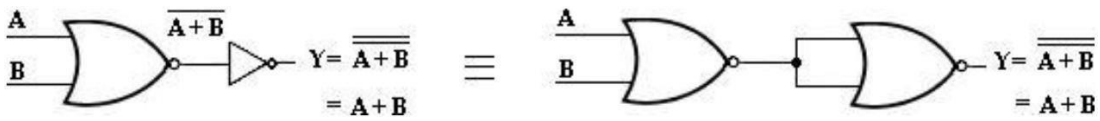
	A	B	Y= $\overline{A+B}$	
X= 0	0	0	1	Y= 1
	0	1	0	
	1	0	0	
X= 1	1	1	0	Y= 0

NOT function using NOR gates

ii) OR function:

By simply inverting output of the NOR gate. i.e.,

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



OR function using NOR gates

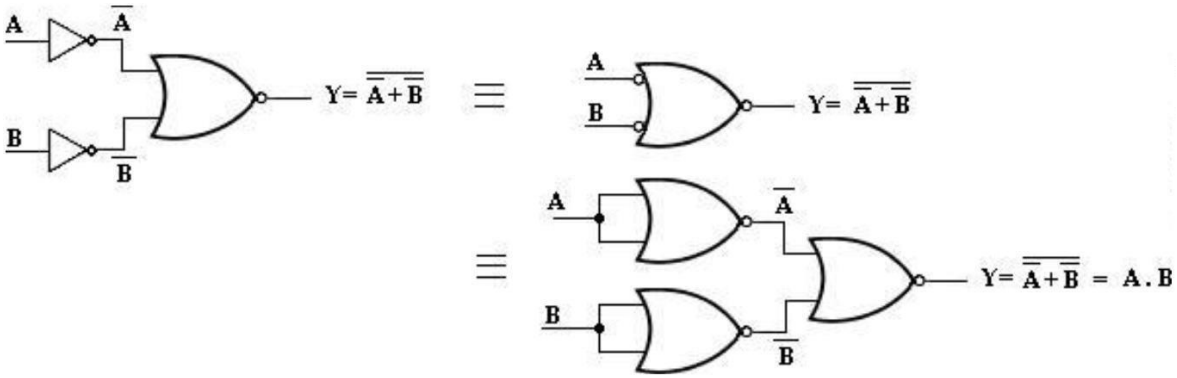
A	B	Y= A+B
0	0	0
0	1	1
1	0	1
1	1	1

≡

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

iii) AND function:

By simply inverting inputs of the NOR gate. i.e.,



AND function using NOR gates

Bubble at the input of NOR gate indicates inverted input.

A	B	Y= A.B
0	0	0
0	1	0
1	0	0
1	1	1

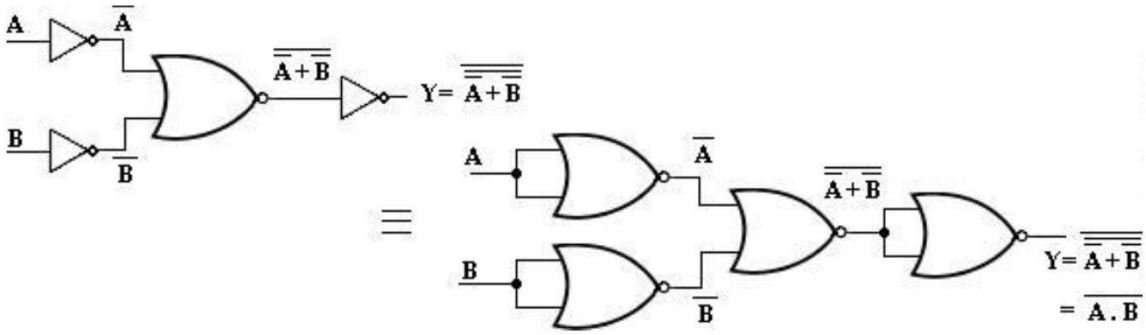
≡

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

Truth table

iv) NAND Function:

By inverting inputs and outputs of the NOR gate.



NAND function using NOR gates

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

≡

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

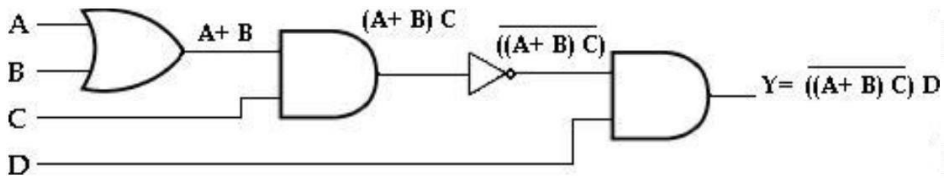
Conversion of AND/OR/NOT to NAND/NOR:

1. Draw AND/OR logic.
2. If NAND hardware has been chosen, add bubbles on the output of each AND gate and bubbles on input side to all OR gates.
   If NOR hardware has been chosen, add bubbles on the output of each OR gate and bubbles on input side to all AND gates.
3. Add or subtract an inverter on each line that received a bubble in step 2.
4. Replace bubbled OR by NAND and bubbled AND by NOR.
5. Eliminate double inversions.

1. Implement Boolean expression using NAND gates:

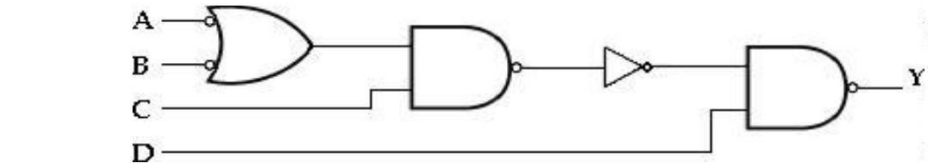
$\overline{((A+\overline{B})\,C)}\,D$

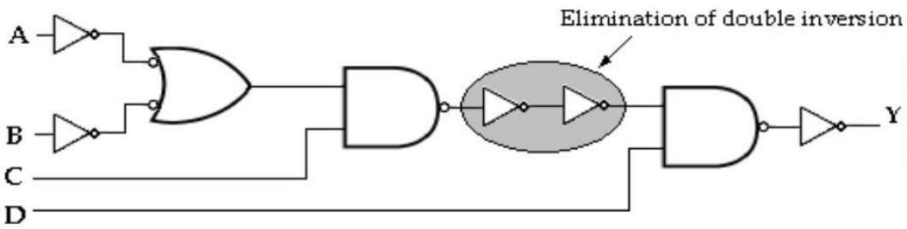
Original Circuit:

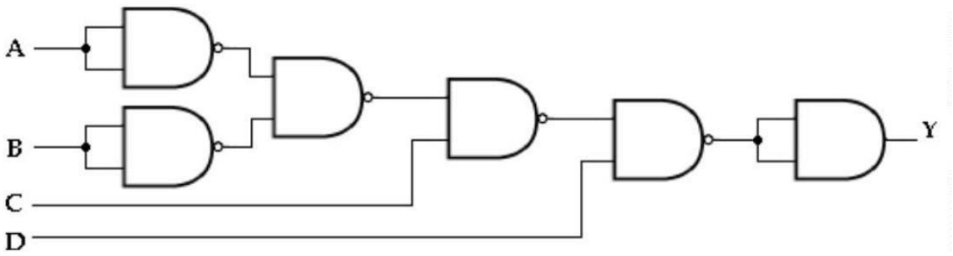


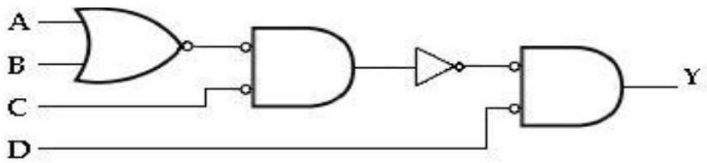
Soln: ___

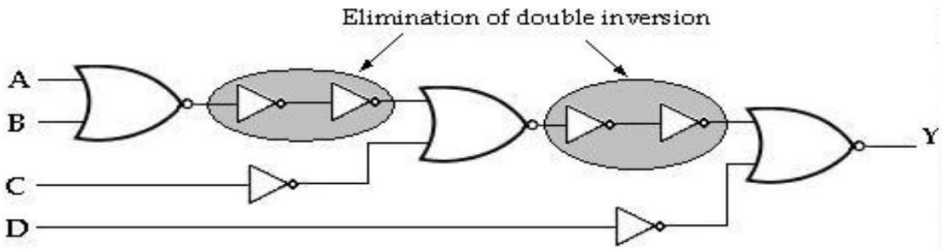
NAND Circuit:

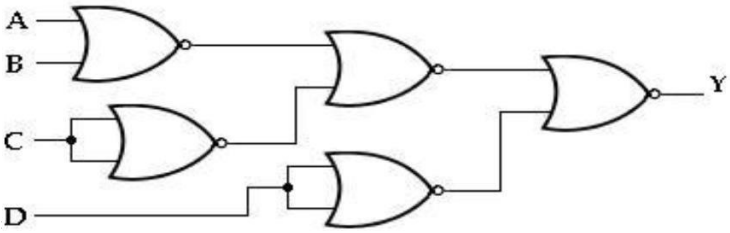






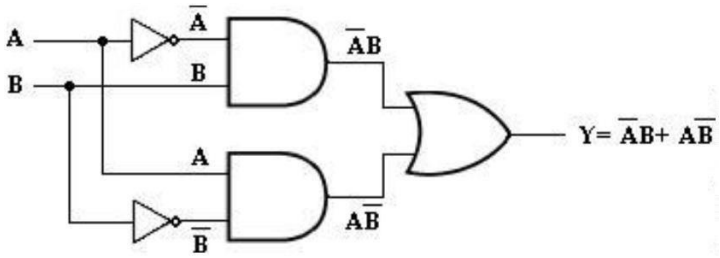






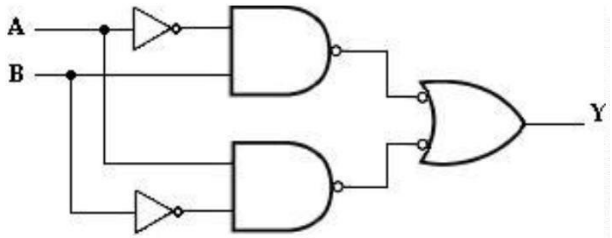
2. Implement Boolean expression for EX-OR gate using NAND gates.

Soln:

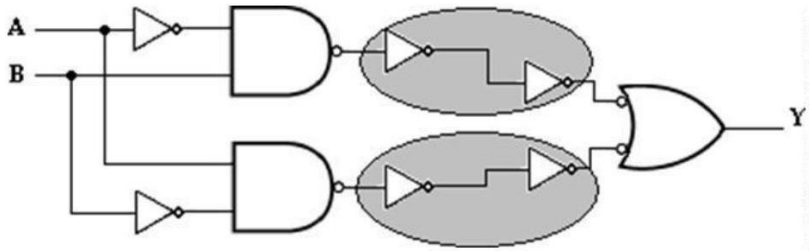


gate.

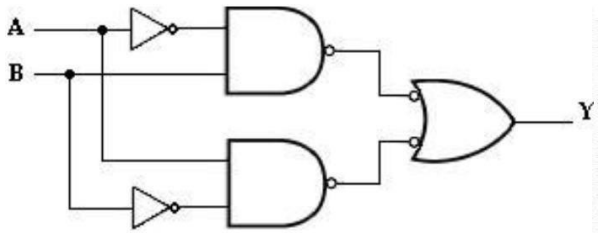
Adding bubbles on the output of each AND gates and on the inputs of each OR



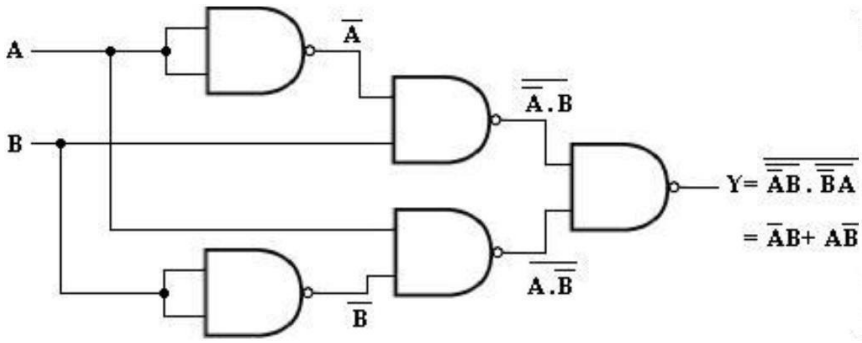
Adding an inverter on each line that received bubble,



Eliminating double inversion,



Replacing inverter and bubbled OR with NAND, we have



UNIT II COMBINATIONAL LOGIC CIRCUITS:

INTRODUCTION:

The digital system consists of two types of circuits, namely

(i) Combinational circuits

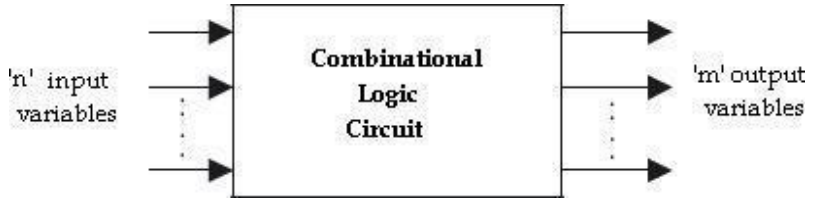
(ii) Sequential circuits

Combinational circuit consists of logic gates whose output at any time is determined from the present combination of inputs. The logic gate is the most basic building block of combinational logic. The logical function performed by a combinational circuit is fully defined by a set of Boolean expressions.

Sequential logic circuit comprises both logic gates and the state of storage elements such as flip-flops. As a consequence, the output of a sequential circuit depends not only on present value of inputs but also on the past state of inputs.

In the previous chapter, we have discussed binary numbers, codes, Boolean algebra and simplification of Boolean function and logic gates. In this chapter, formulation and analysis of various systematic designs of combinational circuits will be discussed.

A combinational circuit consists of input variables, logic gates, and output variables. The logic gates accept signals from inputs and output signals are generated according to the logic circuits employed in it. Binary information from the given data transforms to desired output data in this process. Both input and output are obviously the binary signals, *i.e.*, both the input and output signals are of two possible states, logic1 and logic 0.



Block diagram of a combinational logic circuit

For n number of input variables to a combinational circuit, 2^n possible combinations of binary input states are possible. For each possible combination, there is one and only one possible output combination. A combinational logic circuit can be described by m Boolean functions and each output can be expressed in terms of n input variables.

DESIGN PROCEDURE:

Any combinational circuit can be designed by the following steps of design procedure.

1. The problem is stated.
2. Identify the input and output variables.
3. The input and output variables are assigned letter symbols.
4. Construction of a truth table to meet input -output requirements.
5. Writing Boolean expressions for various output variables in terms of input variables.
6. The simplified Boolean expression is obtained by any method of minimization—algebraic method, Karnaugh map method, or tabulation method.
7. A logic diagram is realized from the simplified boolean expression using logic gates.

The following guidelines should be followed while choosing the preferred form for hardware implementation:

1. The implementation should have the minimum number of gates, with the gates used having the minimum number of inputs.
2. There should be a minimum number of interconnections.
3. Limitation on the driving capability of the gates should not be ignored.

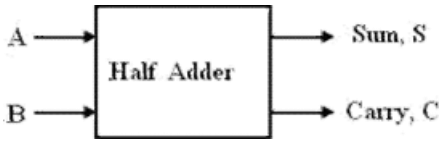
ARITHMETIC CIRCUITS – BASIC BUILDING BLOCKS:

In this section, we will discuss those combinational logic building blocks that can be used to perform addition and subtraction operations on binary numbers. Addition and subtraction are the two most commonly used arithmetic operations, as the other two, namely multiplication and division, are respectively the processes of repeated addition and repeated subtraction.

The basic building blocks that form the basis of all hardware used to perform the arithmetic operations on binary numbers are half-adder, full adder, half-subtractor, full-subtractor.

Half-Adder:

A half-adder is a combinational circuit that can be used to add two binary bits. It has two inputs that represent the two bits to be added and two outputs, with one producing the SUM output and the other producing the CARRY.



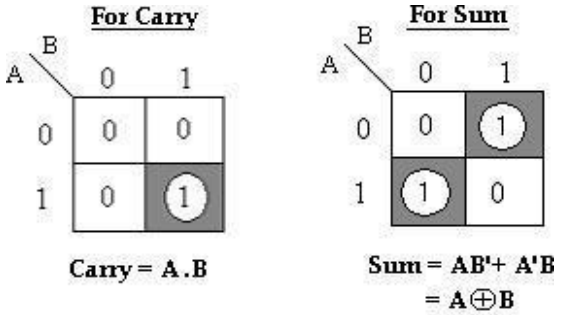
Block schematic of half-adder

The truth table of a half-adder, showing all possible input combinations and the corresponding outputs are shown below.

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

Truth table of half-adder

K-map simplification for carry and sum:



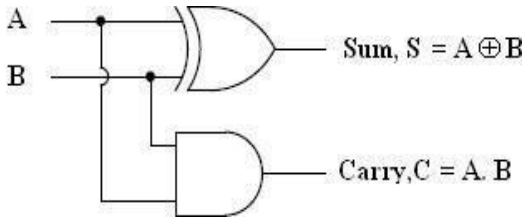
The Boolean expressions for the SUM and CARRY outputs are given by the equations,

Sum, S = A'B+ AB'= A⊔B

Carry, C = A . B

The first one representing the SUM output is that of an EX-OR gate, the second one representing the CARRY output is that of an AND gate.

The logic diagram of the half adder is,

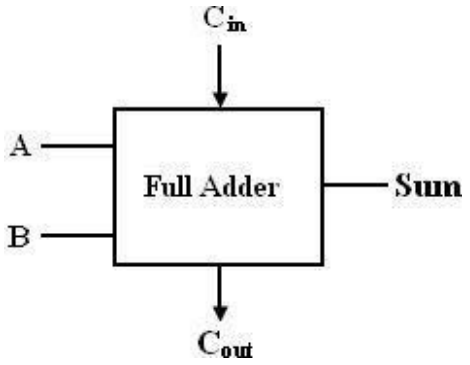


Logic Implementation of Half-adder

Full-Adder:

A full adder is a combinational circuit that forms the arithmetic sum of three input bits. It consists of 3 inputs and 2 outputs.

Two of the input variables, represent the significant bits to be added. The third input represents the carry from previous lower significant position. The block diagram of full adder is given by,



Block schematic of full-adder

The full adder circuit overcomes the limitation of the half-adder, which can be used to add two bits only. As there are three input variables, eight different input combinations are possible. The truth table is shown below,

Truth Table:

Inputs			Outputs	
A	**B**	**C_{in}**	**Sum (S)**	**Carry (C_{out})**
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

To derive the simplified Boolean expression from the truth table, the Karnaugh map method is adopted as,

For Carry

A \ BC_{in}	00	01	11	10
0	0	0	1	0
1	0	1	1	1

Carry, C_{out} = AB+ AC_{in} + BC_{in}

For Sum

A \ BC_{in}	00	01	11	10
0	0	1	0	1
1	1	0	1	0

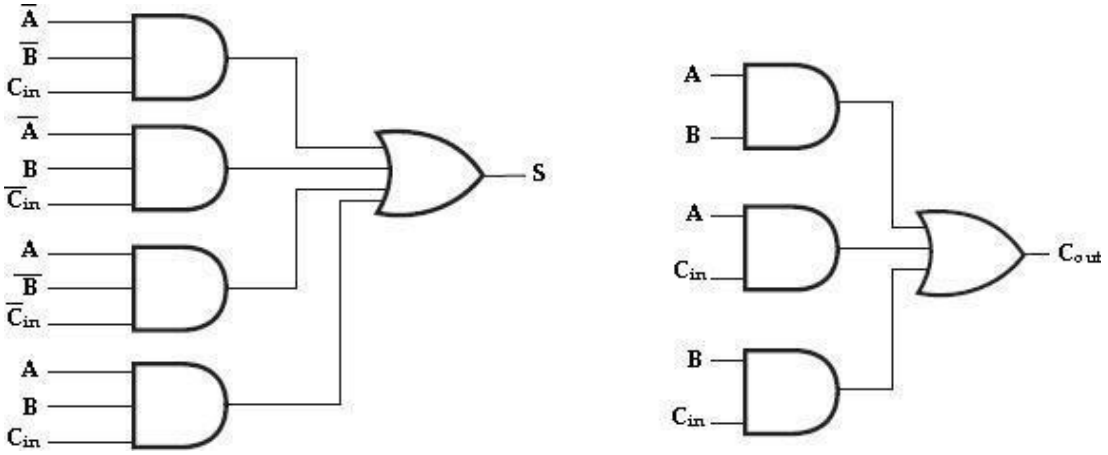
Sum, S = A'B'C_{in}+ A'BC'$_{in}$+ AB'C'$_{in}$+ ABC_{in}

The Boolean expressions for the SUM and CARRY outputs are given by the equations,

Sum, S = A'B'C_{in}+ A'BC'$_{in}$ + AB'C'$_{in}$ + ABC_{in}

Carry, C_{out} = AB+ AC_{in} + BC_{in} .

The logic diagram for the above functions is shown as,



Implementation of full-adder in Sum of Products

The logic diagram of the full adder can also be implemented with two half-adders and one OR gate. The S output from the second half adder is the exclusive-OR of C_{in} and the output of the first half-adder, giving

Sum = C_{in} ⊔ (A ⊔ B) [x⊔y = x'y+ xy']

= C_{in} ⊔ (A'B+AB')

= C'$_{in}$ (A'B+AB') + C_{in} (A'B+AB')' [(x'y+xy')'= (xy+x'y')]

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

= A'BC'$_{in}$ + AB'C'$_{in}$ + ABC_{in} + A'B'C_{in} .

and the carry output is,

Carry, C_{out} = AB+ C_{in} (A'B+AB')

= AB+ A'BC_{in}+ AB'C_{in}

= AB (C_{in}+1) + A'BC_{in}+ AB'C_{in} [C_{in}+1= 1]

= ABC_{in}+ AB+ A'BC_{in}+ AB'C_{in}

= AB+ AC_{in} (B+B') + A'BC_{in}

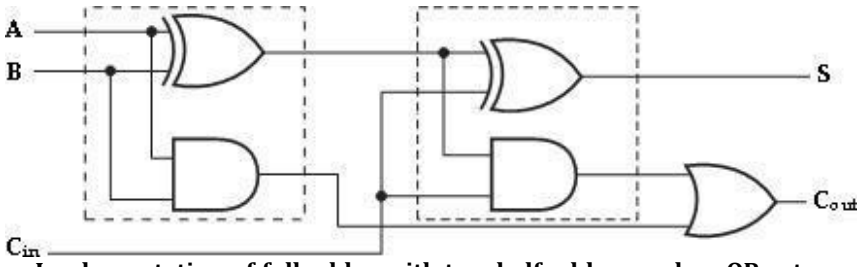
= AB+ AC_{in}+ A'BC_{in}

= AB (C_{in}+1) + AC_{in}+ A'BC_{in} [C_{in}+1= 1]

= ABC_{in}+ AB+ AC_{in}+ A'BC_{in}

= AB+ AC_{in}+ BC_{in} (A +A')

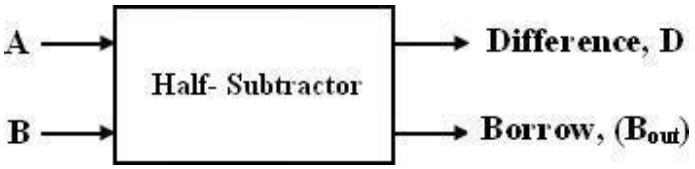
= AB+ AC_{in}+ BC_{in}.



Implementation of full adder with two half-adders and an OR gate

Half -Subtractor:

A *half-subtractor* is a combinational circuit that can be used to subtract one binary digit from another to produce a DIFFERENCE output and a BORROW output. The BORROW output here specifies whether a _1' has been borrowed to perform the subtraction.

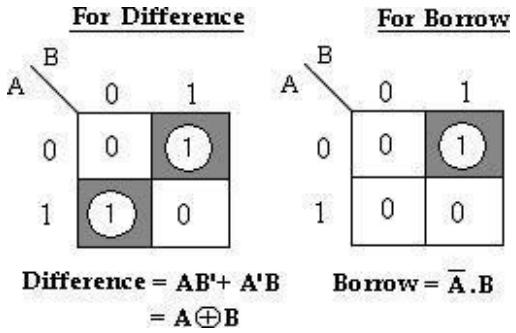


Block schematic of half-subtractor

The truth table of half-subtractor, showing all possible input combinations and the corresponding outputs are shown below.

Input		Output	
A	**B**	**Difference (D)**	**Borrow (B_{out})**
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

K-map simplification for half subtractor:



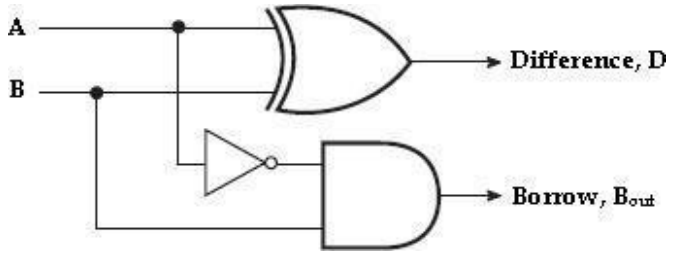
The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = A'B+ AB'= A ⊕ B

Borrow, B_{out} = A' . B

The first one representing the DIFFERENCE (**D**)output is that of an exclusive-OR gate, the expression for the BORROW output (**B_{out}**) is that of an AND gate with input A complemented before it is fed to the gate.

The logic diagram of the half adder is,



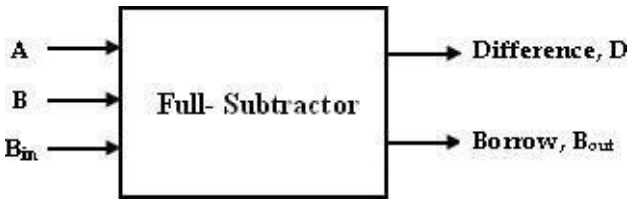
Logic Implementation of Half-Subtractor

Comparing a half-subtractor with a half-adder, we find that the expressions for the SUM and DIFFERENCE outputs are just the same. The expression for BORROW in the case of the half-subtractor is also similar to what we have for CARRY in the case of the half-adder. If the input A, ie., the minuend is complemented, an AND gate can beused to implement the BORROW output.

Full Subtractor:

A *full subtractor* performs subtraction operation on two bits, a minuend and a subtrahend, and also takes into consideration whether a ‗1' has already been borrowed by the previous adjacent lower minuend bit or not.

As a result, there are three bits to be handled at the input of a full subtractor, namely the two bits to be subtracted and a borrow bit designated as B_{in}. There are two outputs, namely the DIFFERENCE output D and the BORROW output B_o. The


A
B
B_{in}
Full- Subtractor
Difference, D
Borrow, B_{out}

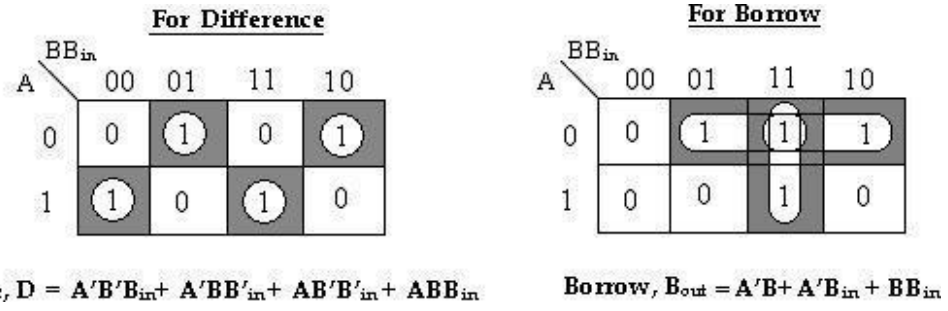
BORROW output bit tells whether the minuend bit needs to borrow a _1' from the next possible higher minuend bit.

Block schematic of full-adder

The truth table for full-subtractor is,

Inputs			Outputs	
A	**B**	**B_{in}**	**Difference(D)**	**Borrow(B_{out})**
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

K-map simplification for full-subtractor:



Difference, $D = A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

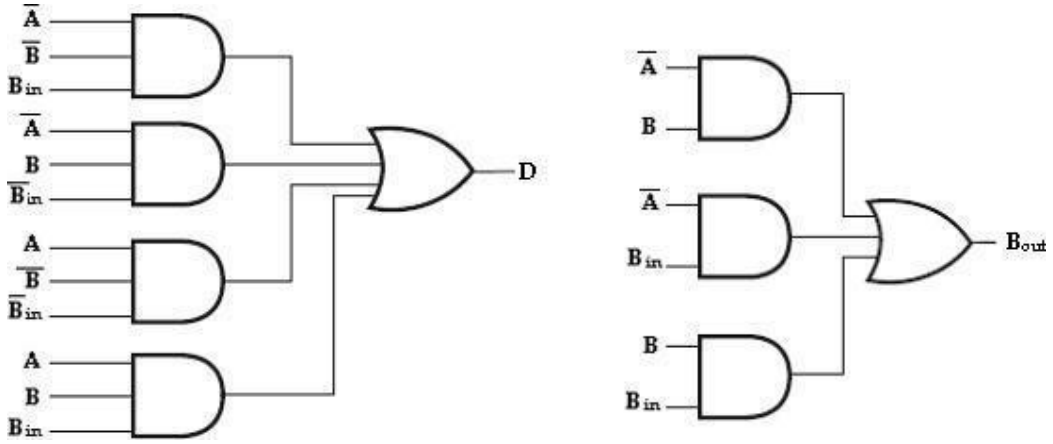
Borrow, $B_{out} = A'B + A'B_{in} + BB_{in}$

The Boolean expressions for the DIFFERENCE and BORROW outputs are given by the equations,

Difference, D = $A'B'B_{in} + A'BB'_{in} + AB'B'_{in} + ABB_{in}$

Borrow, B_{out} = $A'B + A'C_{in} + BB_{in}$.

The logic diagram for the above functions is shown as,



Implementation of full-adder in Sum of Products

The logic diagram of the full-subtractor can also be implemented with two half-subtractors and one OR gate. The difference,D output from the second half subtractor is the exclusive-OR of B_{in} and the output of the first half-subtractor, giving

Difference,D= B_{in} ⊻ (A ⊻ B) [x ⊻ y = x‘y+ xy‘]

= **B_{in}** ⊻ (A‘B+AB‘)

= B‘$_{in}$ (A‘B+AB‘) + B_{in} (A‘B+AB‘)‘ [(x‘y+xy‘)‘= (xy+x‘y‘)]

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

= A‘BB‘$_{in}$ + AB‘B‘$_{in}$ + ABB_{in} + A‘B‘B_{in}

. and the borrow output is,

Borrow, B_{out} = A’B+ B_{in} (A’B+AB’)’ [(x‘y+xy‘)‘= (xy+x‘y‘)]

= A‘B+ B_{in} (AB+A‘B‘)

= A‘B+ ABB_{in}+ A‘B‘B_{in}

= A‘B (B_{in}+1) + ABB_{in}+ A‘B‘B_{in} [C_{in}+1= 1]

= A‘BB_{in}+ A‘B+ ABB_{in}+ A‘B‘B_{in}

= A‘B+ BB_{in} (A+A‘) + A‘B‘B_{in} [A+A‘= 1]

= A‘B+ BB_{in}+ A‘B‘B_{in}

= A‘B (B_{in}+1) + BB_{in}+ A‘B‘B_{in} [C_{in}+1= 1]

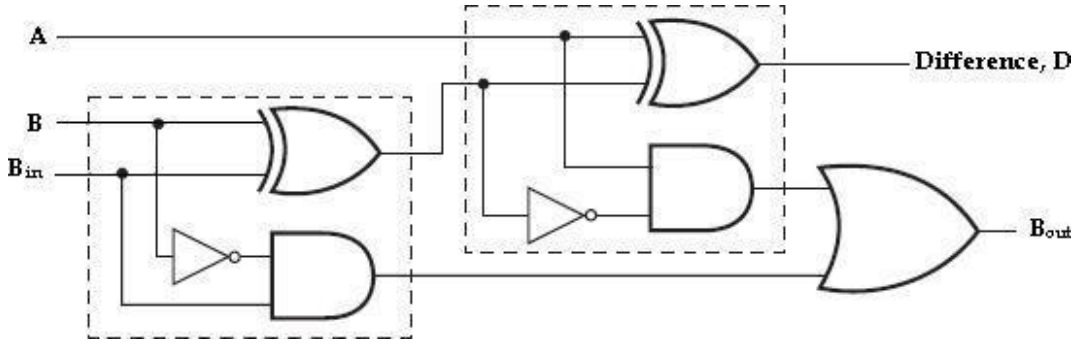
= A‘BB_{in}+ A‘B+ BB_{in}+ A‘B‘B_{in}

= A‘B+ BB_{in}+ A‘B_{in} (B +B‘)

= A‘B+ BB_{in}+ A‘B_{in}.

Therefore,

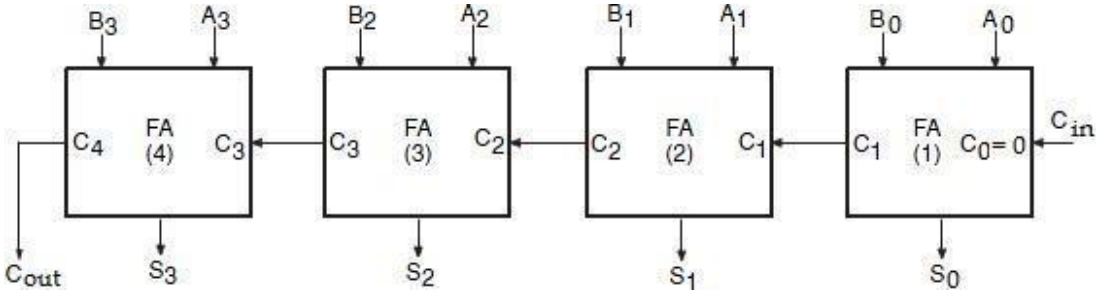
we can implement full-subtractor using two half-subtractors and OR gate as,



Implementation of full-subtractor with two half-subtractors and an OR gate

Binary Adder (Parallel Adder):

The 4-bit binary adder using full adder circuits is capable of adding two 4-bit numbers resulting in a 4-bit sum and a carry output as shown in figure below.



4-bit binary parallel Adder

Since all the bits of augend and addend are fed into the adder circuits simultaneously and the additions in each position are taking place at the same time, this circuit is known as parallel adder.

Let the 4-bit words to be added be represented by,

$A_3A_2A_1A_0$= 1111 and $B_3B_2B_1B_0$= 0011.

Significant place	4	3	2	1
Input carry	1	1	1	0
Augend word A :	1	1	1	1
Addend word B :	0	0	1	1
	1	0	0	1	0 ← Sum

↑ Output Carry

The bits are added with full adders, starting from the least significant position, to form the sum it and carry bit. The input carry C_0 in the least significant position must be 0. The carry output of the lower order stage is connected to the carry input of the next higher order stage. Hence this type of adder is called ripple-carry adder.

In the least significant stage, A_0, B_0 and C_0 (which is 0) are added resulting in sum S_0 and carry C_1. This carry C_1 becomes the carry input to the second stage. Similarly in the second stage, A_1, B_1 and C_1 are added resulting in sum S_1 and carry C_2, in the third stage, A_2, B_2 and C_2 are added resulting in sum S_2 and carry C_3, in the third stage, A_3, B_3 and C_3 are added resulting in sum S_3 and C_4, which is the output carry. Thus the circuit results in a sum ($S_3S_2S_1S_0$) and a carry output (C_{out}).

Though the parallel binary adder is said to generate its output immediately after the inputs are applied, its speed of operation is limited by the carry propagation delay

through all stages. However, there are several methods to reduce this delay.

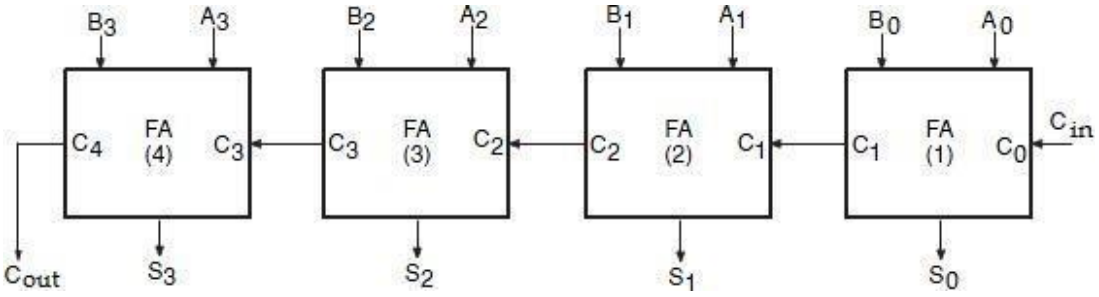
One of the methods of speeding up this process is look-ahead carry addition which eliminates the ripple-carry delay.

Carry Propagation–Look-Ahead Carry Generator:

In Parallel adder, all the bits of the augend and the addend are available for computation at the same time. The carry output of each full-adder stage is connected to the carry input of the next high-order stage. Since each bit of the sum output depends on the value of the input carry, time delay occurs in the addition process. This timedelay is called as **carry propagation delay.**

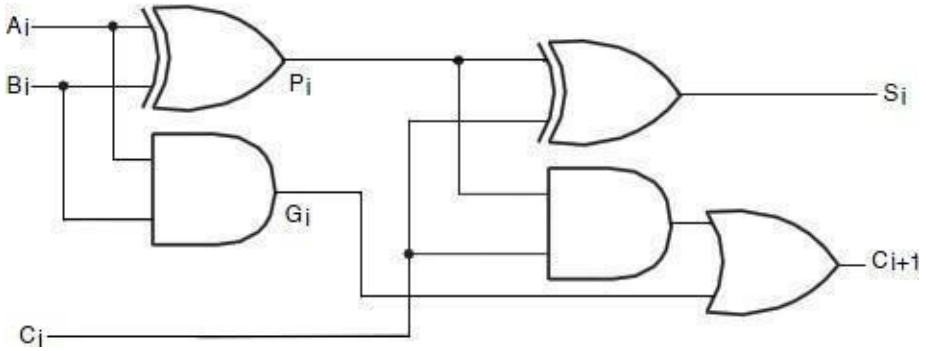
For example, addition of two numbers (0011+ 0101) gives the result as 1000. Addition of the LSB position produces a carry into the second position. This carry when added to the bits of the second position, produces a carry into the third position. This carry when added to bits of the third position, produces a carry into the last position. The sum bit generated in the last position (MSB) depends on the carry that was generated by the addition in the previous position. i.e., the adder will not produce correct result until LSB carry has propagated through the intermediate full-adders. This represents a time delay that depends on the propagation delay produced in an each full-adder. For example, if each full adder is considered to have a propagation delay of

30nsec, then S_3 will not react its correct value until 90 nsec after LSB is generated. Therefore total time required to perform addition is 90+ 30 = 120nsec.



4-bit Parallel Adder

The method of speeding up this process by eliminating inter stage carry delay is called **look ahead-carry addition**. This method utilizes logic gates to look at the lower order bits of the augend and addend to see if a higher-order carry is to be generated. It uses two functions: carry generate and carry propagate.



Full-Adder circuit

Consider the circuit of the full-adder shown above. Here we define two functions: carry generate (G_i) and carry propagate (P_i) as,

Carry generate, $\mathbf{G_i = A_i \cdot B_i}$

Carry propagate, $\mathbf{P_i = A_i \oplus B_i}$

the output sum and carry can be expressed as,

$$\mathbf{S_i = P_i \oplus C_i}$$

$$\mathbf{C_{i+1} = G_i + P_iC_i}$$

G_i (carry generate), it produces a carry 1 when both Ai and Bi are 1, regardless of the input carry C_i.

Pi (carry propagate) because it is the term associated with the propagation of the carry from C_i to C_{i+1}.

The Boolean functions for the carry outputs of each stage and substitute for each C_i its value from the previous equation:

C_0= input carry

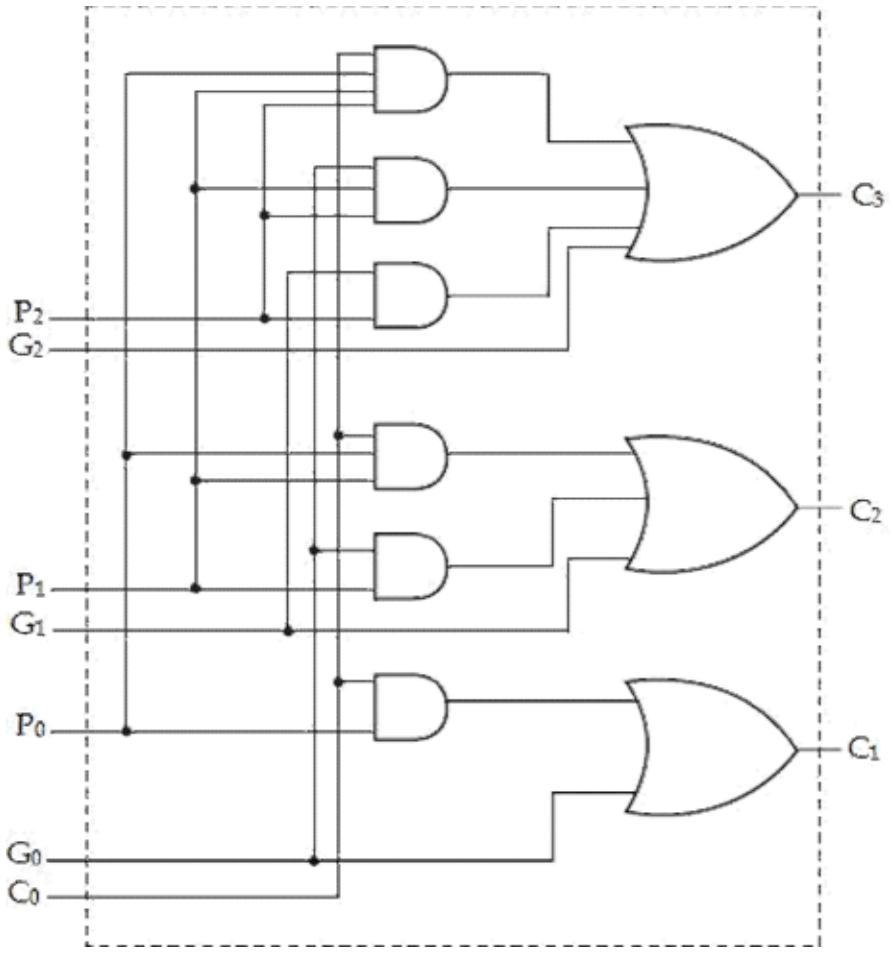
$C_1 = G_0 + P_0C_0$

$C_2 = G_1 + P_1C_1 = G_1 + P_1 (G_0 + P_0C_0)$

$= G_1 + P_1G_0 + P_1P_0C_0$

$C_3 = G_2 + P_2C_2 = G_2 + P_2 (G_1 + P_1G_0 + P_1P_0C_0)$

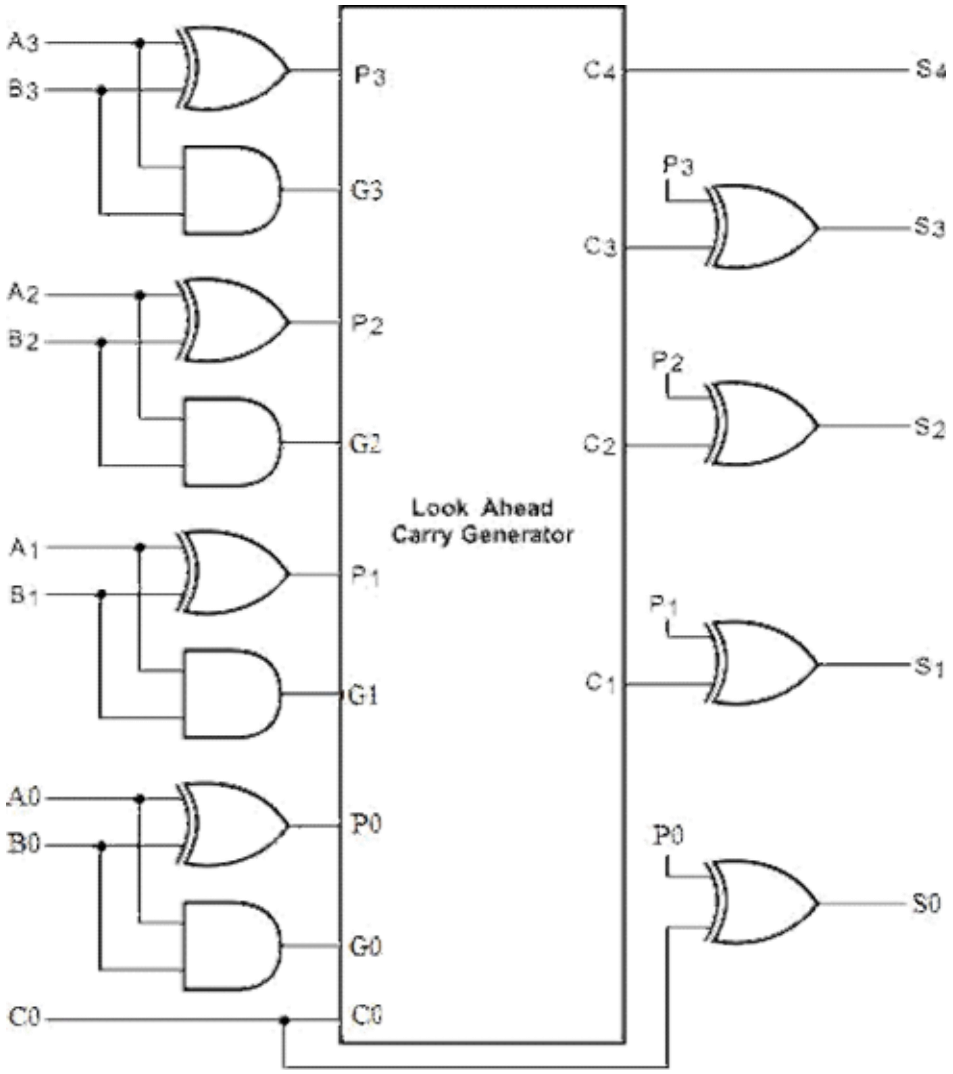
$= G_2 + P_2G_1 + P_2P_1G_0 + P_2P_1P_0C_0$



Since the Boolean function for each output carry is expressed in sum of products, each function can be implemented with one level of AND gates followed by an OR gate. The three Boolean functions for C_1, C_2 and C_3 are implemented in the carry look-ahead generator as shown below. Note that C_3 does not have to wait for C_2 and C_1 to propagate; in fact C_3 is propagated at the same time as C_1 and C_2.

Logic diagram of Carry Look-ahead Generator

Using a Look-ahead Generator we can easily construct a 4-bit parallel adder with a Look-ahead carry scheme. Each sum output requires two exclusive-OR gates. The output of the first exclusive-OR gate generates the P_i variable, and the AND gate generates the G_i variable. The carries are propagated through the carry look-ahead generator and applied as inputs to the second exclusive-OR gate. All output carries are generated after a delay through two levels of gates. Thus, outputs S_1 through S_3 have equal propagation delay times.

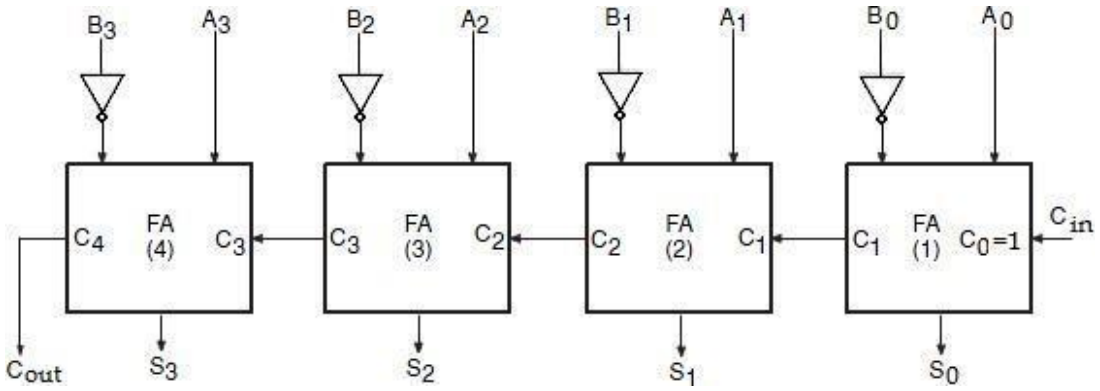


4-Bit Adder with Carry Look-ahead

Binary Subtractor (Parallel Subtractor):

The subtraction of unsigned binary numbers can be done most conveniently by means of complements. The subtraction A-B can be done by taking the 2‘s complement of B and adding it to A. The 2‘s complement can be obtained by taking the 1‘s complement and adding 1 to the least significant pair of bits. The 1‘s complement can be implemented with inverters and a 1 can be added to the sum through the input carry.

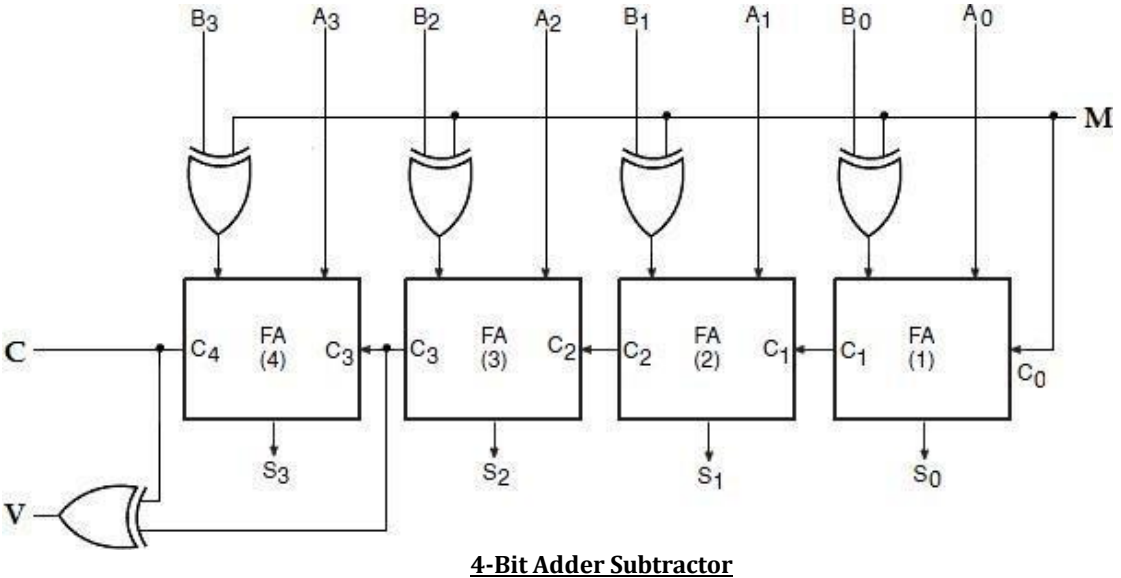
The circuit for subtracting A-B consists of an adder with inverters placed between each data input B and the corresponding input of the full adder. The input carry C_0 must be equal to 1 when performing subtraction. The operation thus performed becomes A, plus the 1's complement of B, plus1. This is equal to A plus the 2's complement of B.



4-bit Parallel Subtractor

Parallel Adder/ Subtractor:

The addition and subtraction operation can be combined into one circuit with one common binary adder. This is done by including an exclusive-OR gate with each full adder. A 4-bit adder Subtractor circuit is shown below.



4-Bit Adder Subtractor

The mode input M controls the operation. When M= 0, the circuit is an adder and when M=1, the circuit becomes a Subtractor. Each exclusive-OR gate receives input M

and one of the inputs of B. When M=0, we have B⊕0= B. The full adders receive the value of B, the input carry is 0, and the circuit performs A plus B. When M=1, we have B ⊕1= B' and C_0=1. The B inputs are all complemented and a 1 is added through the input carry. The circuit performs the operation A plus the 2's complement of B. The exclusive-OR with output V is for detecting an overflow.

Decimal Adder (BCD Adder):

The digital system handles the decimal number in the form of binary coded decimal numbers (BCD). A BCD adder is a circuit that adds two BCD bits and producesa sum digit also in BCD.

Consider the arithmetic addition of two decimal digits in BCD, together with an input carry from a previous stage. Since each input digit does not exceed 9, the output sum cannot be greater than 9+ 9+1 = 19; the 1 is the sum being an input carry. The adder will form the sum in binary and produce a result that ranges from 0 through 19.

These binary numbers are labeled by symbols K, Z_8, Z_4, Z_2, Z_1, K is the carry. The columns under the binary sum list the binary values that appear in the outputs of the 4-bit binary adder. The output sum of the two decimal digits must be represented in BCD.

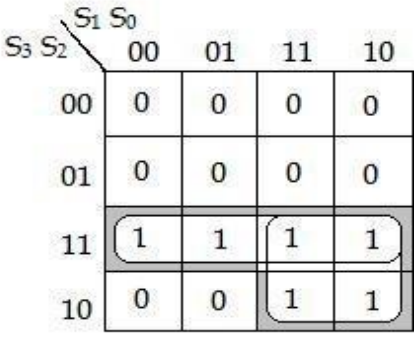
Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9

0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

In examining the contents of the table, it is apparent that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical, and therefore no conversion is needed. When the binary sum is greater than 9 (1001), we obtain a non-valid BCD representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an output carry as required.

The logic circuit to detect sum greater than 9 can be determined by simplifying the boolean expression of the given truth table.

Inputs				Output
S_3	S_2	S_1	S_0	Y
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

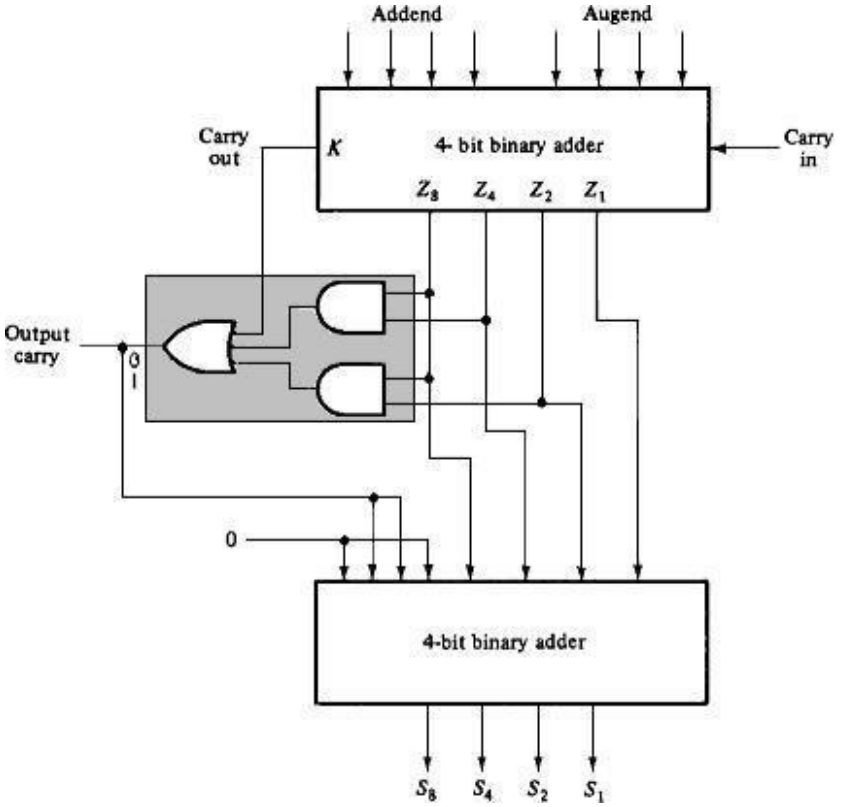


$Y = S_3S_2 + S_3S_1$

To implement BCD adder we require:

- 4-bit binary adder for initial addition
- Logic circuit to detect sum greater than 9 and
- One more 4-bit adder to add 0110_2 in the sum if the sum is greater than 9 or carry is 1.

The two decimal digits, together with the input carry, are first added in the top4-bit binary adder to provide the binary sum. When the output carry is equal to zero, nothing is added to the binary sum. When it is equal to one, binary 0110 is added to the binary sum through the bottom 4-bit adder. The output carry generated from the bottom adder can be ignored, since it supplies information already available at the output carry terminal. The output carry from one stage must be connected to the input carry of the next higher-order stage.



Block diagram of BCD adder

Binary Multiplier:

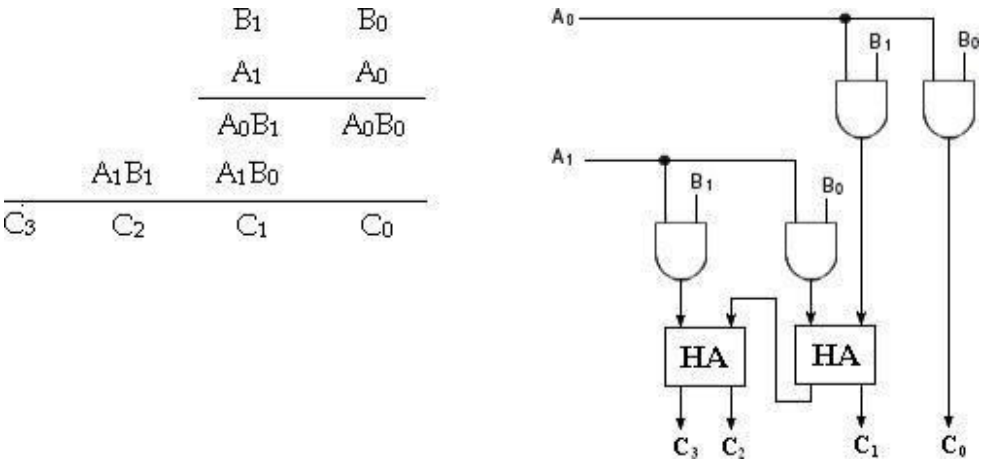
Multiplication of binary numbers is performed in the same way as in decimal numbers. The multiplicand is multiplied by each bit of the multiplier starting from the least significant bit. Each such multiplication forms a partial product. Such partial

products are shifted one position to the left. The final product is obtained from the sum of partial products.

Consider the multiplication of two 2-bit numbers. The multiplicand bits are B_1 and B_0, the multiplier bits are A_1 and A_0, and the product is C_3, C_2, C_1 and C_0. The first partial product is formed by multiplying A_0 by B_1B_0. The multiplication of two bits such as A_0 and B_0 produces a 1 if both bits are 1; otherwise, it produces a 0. This is identicalto an AND operation. Therefore the partial product can be implemented with ANDgates as shown in the diagram below.

The second partial product is formed by multiplying A_1 by B_1B_0 and shifted one position to the left. The two partial products are added with two half adder (HA) circuits.

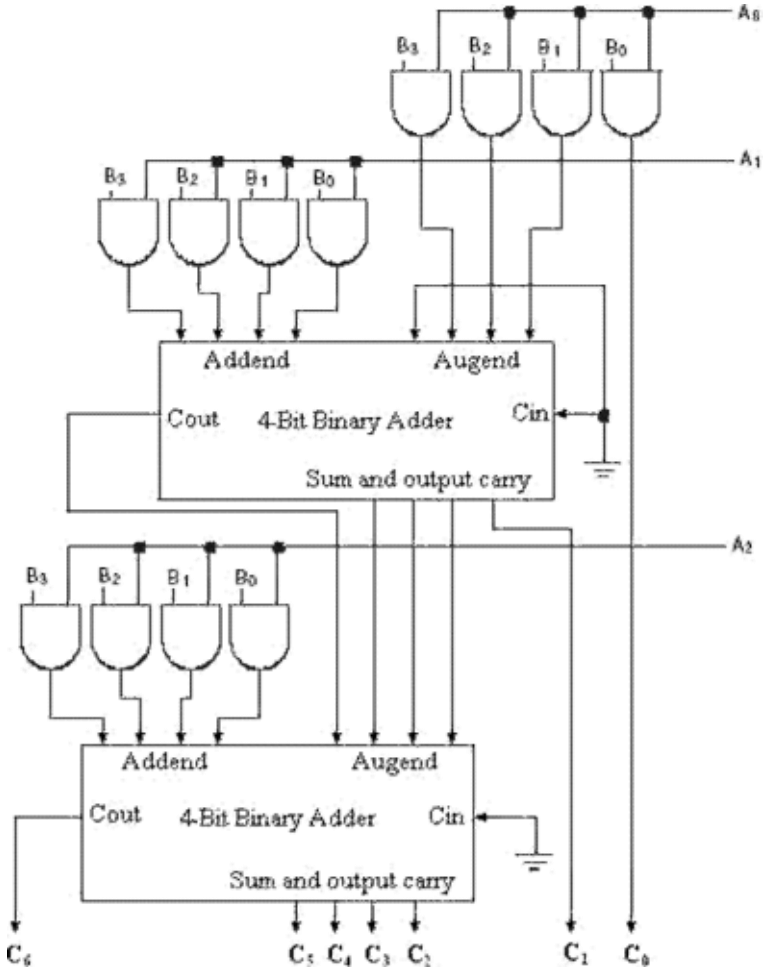
2- bit by 2-bit Binary multiplier



Usually there are more bits in the partial products and it is necessary to use full adders to produce the sum of the partial products. The least significant bit of the product does not have to go through an adder since it is formed by the output of thefirst AND gate.

A combinational circuit binary multiplier with more bits can be constructed in a similar fashion. A bit of the multiplier is ANDed with each bit of the multiplicand in as many levels as there are bits in the multiplier. The binary output in each level of AND gates are added with the partial product of the previous level to form a new partial product. The last level produces the product. For J multiplier bits and K multiplicand bits we need (J x K) AND gates and (J-1) k-bit adders to produce a product of J+K bits.

Consider a multiplier circuit that multiplies a binary number of four bits by a number of three bits. Let the multiplicand be represented by B3, B2, B1, B0 and the multiplier by A_2, A_1, and A_0. Since K= 4 and J= 3, we need 12 AND gates and two 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is shown below.



4-bit by 3-bit Binary multiplier

PARITY GENERATOR/ CHECKER:

A ***Parity*** is a very useful tool in information processing in digital computers to indicate any presence of error in bit information. External noise and loss of signal strength causes loss of data bit information while transporting data from one device to other device, located inside the computer or externally. To indicate any occurrence of error, an extra bit is included with the message according to the total number of 1s in a set of data, which is called ***parity***.

If the extra bit is considered 0 if the total number of 1s is even and 1 for odd quantities of 1s in a set of data, then it is called ***even parity***. On the other hand, if the extra bit is 1 for even quantities of 1s and 0 for an odd number of 1s, then it is called ***oddparity.***

The message including the parity is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted. The circuit that generates the parity bit in the transmitter is called a parity generator and the circuit that checks the parity in the receiver is called a parity checker.

Parity Generator:

A parity generator is a combination logic system to generate the parity bit at the transmitting side. A table illustrates even parity as well as odd parity for a message consisting of three bits.

3-bit Message			Odd Party bit	Even Parity bit
A	B	C		
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

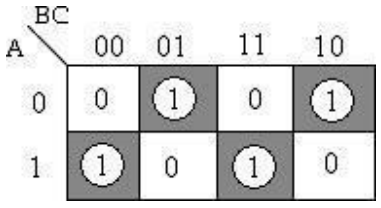
Parity generator truth table for even and odd parity

If the message bit combination is designated as A, B, C and P_e, P_o are the even and odd parity respectively, then it is obvious from table that the boolean expressions of even parity and odd parity are

P_e = A ⊕ (B ⊕ C) and

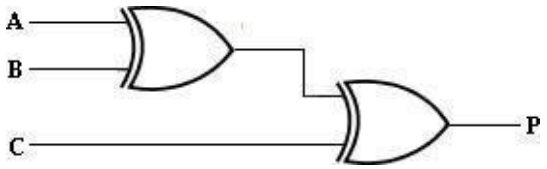
P_o = (A ⊕ B ⊕ C)′.

K-map Simplification:



P= A'B'C+ A'BC'+ A'B'C'+ ABC
= A' (B'C+ BC') + A (B'C'+ BC)
= A' (B⊕C) + A (B⊕C)'
= A ⊕ (B ⊕ C)

Logic Diagram:



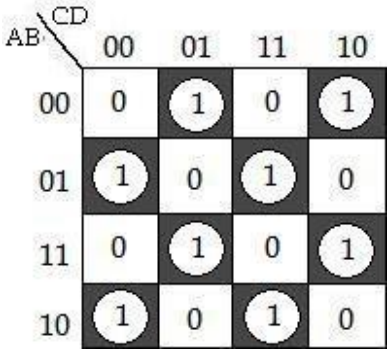
3-bit even parity generator

Parity Checker:

The message bits with the parity bit are transmitted to their destination, where they are applied to a parity checker circuit. The circuit that checks the parity at the receiver side is called the *parity checker*. The parity checker circuit produces a check bit and is very similar to the parity generator circuit. If the check bit is 1, then it is assumed that the received data is incorrect. The check bit will be 0 if the received data is correct. The table shows the truth table for the even parity checker.

4-Bit Received				Parity Error Check (PEC)
A	B	C	D	
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

K-map Simplification:



PEC= A'B' (C'D+ CD') + A'B (C'D'+ CD) + AB (C'D+ CD') + AB' (C'D'+ CD)

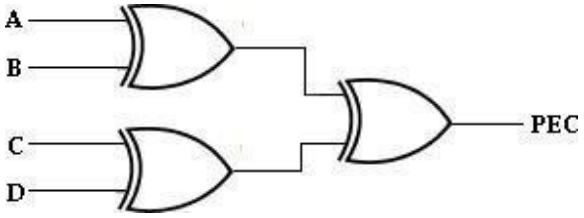
= A'B' (C⊻D) + A'B (C⊻D)' + AB (C⊻D) + AB' (C⊻D)'

= (A'B'+ AB) (C⊻D) + (A'B+ AB') (C⊻D)'

= (A⊻B)' (C⊻D) + (A⊻B) (C⊻D)'

= (A⊻B) ⊻ (C⊻D)

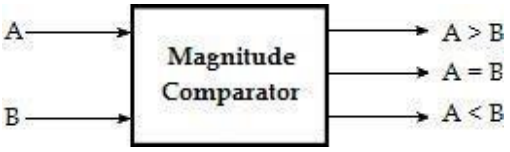
Logic Diagram:



4-bit even parity checker

MAGNITUDE COMPARATOR:

A ***magnitude comparator*** is a combinational circuit that compares two given numbers (A and B) and determines whether one is equal to, less than or greater than the other. The output is in the form of three binary variables representing the conditions A = B, A>B and A<B, if A and B are the two numbers being compared.



Block diagram of magnitude comparator

For comparison of two *n*-bit numbers, the classical method to achieve the Boolean expressions requires a truth table of 2^{2n} entries and becomes too lengthy and cumbersome.

2-bit Magnitude Comparator:

The truth table of 2-bit comparator is given in table below—

Truth table:

Inputs				Outputs		
A_3	A_2	A_1	A_0	A>B	A=B	A<B
0	0	0	0	0	1	0
0	0	0	1	0	0	1
0	0	1	0	0	0	1
0	0	1	1	0	0	1
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	1
1	0	0	0	1	0	0
1	0	0	1	1	0	0
1	0	1	0	0	1	0
1	0	1	1	0	0	1
1	1	0	0	1	0	0
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	0	1	0

K-map Simplification:

For A>B

A_1A_0 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

$A>B = A_0B_1'B_0' + A_1B_1' + A_1A_0B_0'$

For A=B

A_1A_0 \ B_1B_0	00	01	11	10
00	1	0	0	0
01	0	1	0	0
11	0	0	1	0
10	0	0	0	1

$A=B = A_1'A_0'B_1'B_0' + A_1'A_0B_1'B_0 + A_1A_0B_1B_0 + A_1A_0'B_1B_0'$

$= A_1'B_1' (A_0'B_0' + A_0B_0) + A_1B_1 (A_0B_0 + A_0'B_0')$

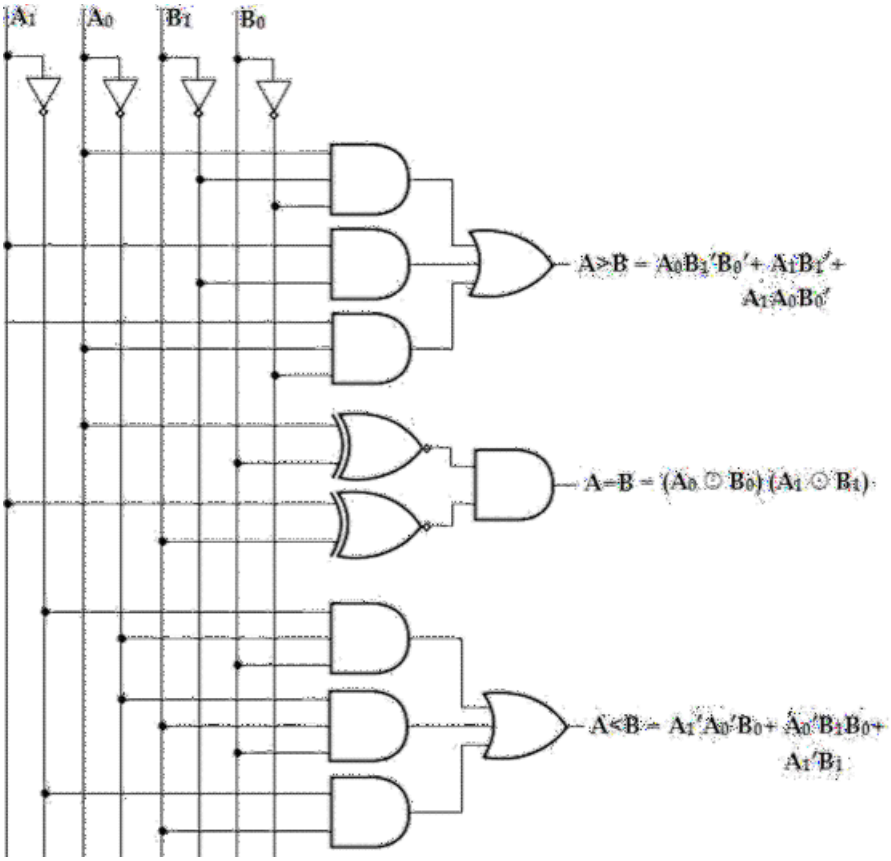
$= (A_0 \odot B_0)(A_1 \odot B_1)$

For A<B

A_1A_0 \ B_1B_0	00	01	11	10
00	0	1	1	1
01	0	0	1	1
11	0	0	0	0
10	0	0	1	0

$A<B = A_1'A_0'B_0 + A_0'B_1B_0 + A_1'B_1$

Logic Diagram:



2- bit Magnitude Comparator

4-bit Magnitude Comparator:

Let us consider the two binary numbers A and B with four digits each. Write the coefficient of the numbers in descending order as,

A = $A_3A_2A_1A_0$
B = B_3 B_2 B_1 B_0,

Each subscripted letter represents one of the digits in the number. It is observed from the bit contents of two numbers that A = B when $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_0 = B_0$. When the numbers are binary they possess the value of either 1 or 0, the equality relation of each pair can be expressed logically by the equivalence function as

$X_i = A_iB_i + A_i'B_i'$ for i = 1, 2, 3, 4.
Or, **$X_i = (A \llcorner B)'$.** or, **$X_i' = A \llcorner B$**
Or, **$X_i = (A_iB_i' + A_i'B_i)'$.**

where,
X*i =1* only if the pair of bits in position i are equal (ie., if both are 1 or both are 0).

To satisfy the equality condition of two numbers A and B, it is necessary that all X*i* must be equal to logic 1. This indicates the AND operation of all X*i* variables. In other words, we can write the Boolean expression for two equal 4-bit numbers.

$$(A = B) = X_3X_2X_1 X_0.$$

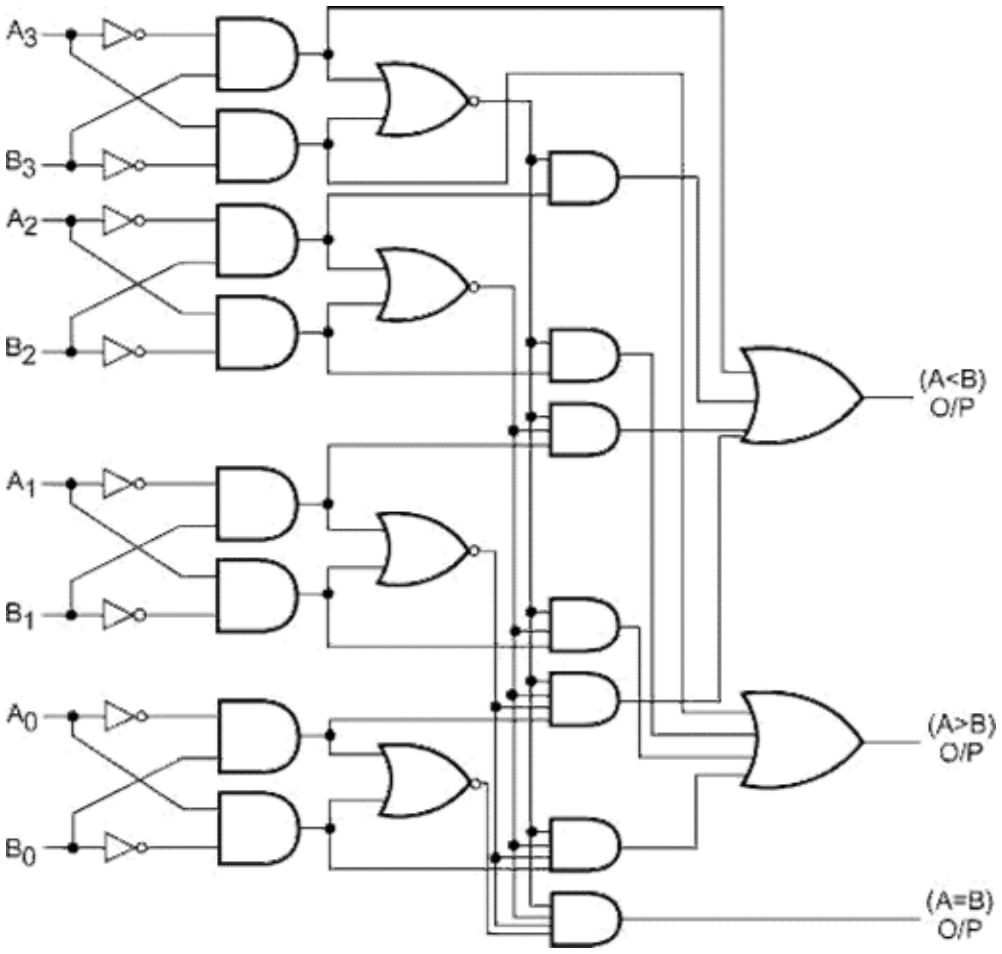
The binary variable (A=B) is equal to 1 only if all pairs of digits of the two numbers are equal.

To determine if A is greater than or less than B, we inspect the relative magnitudes of pairs of significant bits starting from the most significant bit. If the two digits of the most significant position are equal, the next significant pair of digits is compared. The comparison process is continued until a pair of unequal digits is found. It may be concluded that A>B, if the corresponding digit of A is 1 and B is 0. If the corresponding digit of A is 0 and B is 1, we conclude that A<B. Therefore, we can derive the logical expression of such sequential comparison by the following two Boolean functions,

$$(A>B) = A_3B_3' + X_3A_2B_2' + X_3X_2A_1B_1' + X_3X_2X_1A_0B_0'$$
$$(A<B) = A_3'B_3 + X_3A_2'B_2 + X_3X_2A_1'B_1 + X_3X_2X_1A_0'B_0$$

The symbols (A>B) and (A<B) are binary output variables that are equal to 1 when A>B or A<B, respectively.

The gate implementation of the three output variables just derived is simpler than it seems because it involves a certain amount of repetition. The unequal outputs can use the same gates that are needed to generate the equal output. The logic diagram of the 4-bit magnitude comparator is shown below,



4-bit Magnitude Comparator

The four x outputs are generated with exclusive-NOR circuits and applied to an AND gate to give the binary output variable (A=B). The other two outputs use the x variables to generate the Boolean functions listed above. This is a multilevel implementation and has a regular pattern.

CODE CONVERTERS:

A code converter is a logic circuit that changes data presented in one type of binary code to another code of binary code. The following are some of the most commonly used code converters:

i. **Binary-to-Gray code**
ii. **Gray-to-Binary code**
iii. **BCD-to-Excess-3**
iv. **Excess-3-to-BCD**
v. **Binary-to-BCD**
vi. **BCD-to-binary**
vii. **Gray-to-BCD**
viii. **BCD-to-Gray**
ix. **8 4 -2 -1 to BCD converter**

1. Binary to Gray Converters:

The gray code is often used in digital systems because it has the advantage that only one bit in the numerical representation changes between successive numbers. The truth table for the binary-to-gray code converter is shown below,

Truth table:

Decimal	Binary code				Gray code			
	B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	1
2	0	0	1	0	0	0	1	1
3	0	0	1	1	0	0	1	0
4	0	1	0	0	0	1	1	0
5	0	1	0	1	0	1	1	1
6	0	1	1	0	0	1	0	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	1	1	0	0
9	1	0	0	1	1	1	0	1
10	1	0	1	0	1	1	1	1
11	1	0	1	1	1	1	1	0
12	1	1	0	0	1	0	1	0
13	1	1	0	1	1	0	1	1
14	1	1	1	0	1	0	0	1
15	1	1	1	1	1	0	0	0

K-map simplification:

For G_3

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

$G_3 = B_3$

For G_2

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

$G_2 = B_3'B_2 + B_3B_2'$
$= B_3 \oplus B_2$

For G_1

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	1	1	0	0
10	0	0	1	1

$G_1 = B_2'B_1 + B_2B_1'$
$= B_2 \oplus B_1$

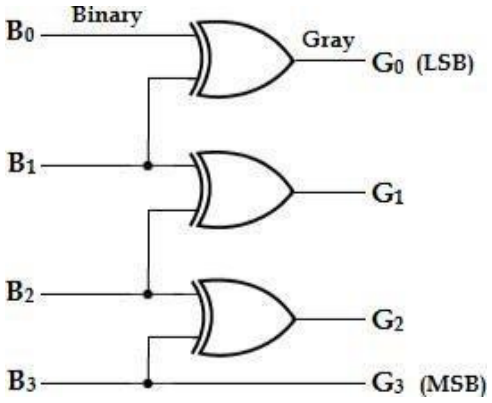
For G_0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	0	1	0	1
10	0	1	0	1

$G_0 = B_1'B_0 + B_1B_0'$
$= B_1 \oplus B_0$

Now, the above expressions can be implemented using EX-OR gates as,

Logic Diagram:



2. Gray to Binary Converters:

The truth table for the gray-to-binary code converter is shown below,

Truth table:

Gray code				Binary code			
G_3	G_2	G_1	G_0	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	1
0	1	0	1	0	1	1	0
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
1	0	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	0	1	0	1	1
1	1	1	1	1	0	1	0

From the truth table, the logic expression for the binary code outputs can be written as,

$G_3 = \sum_m (8, 9, 10, 11, 12, 13, 14, 15)$

$G_2 = \sum_m (4, 5, 6, 7, 8, 9, 10, 11)$

$G_1 = \sum_m (2, 3, 4, 5, 8, 9, 14, 15)$

$G_0 = \sum_m (1, 2, 4, 7, 8, 11, 13, 14)$

K-map Simplification:

For B_3

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	1	1	1	1

$B_3 = G_3$

For B_2

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	1	1

$B_2 = G_3'G_2 + G_3G_2'$
$= G_3 \oplus G_2$

For B_1

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	0	0	1	1
10	1	1	0	0

For B_0

G_3G_2 \ G_1G_0	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

From the above K-map,

$B_3 = G_3$

$B_2 = G_3'G_2 + G_3G_2'$

$B_2 = G_3 \veebar G_2$

$B_1 = G_3'G_2'G_1 + G_3'G_2G_1' + G_3G_2G_1 + G_3G_2'G_1'$

$= G_3' (G_2'G_1 + G_2G_1') + G_3 (G_2G_1 + G_2'G_1')$

$= G_3' (G_2 \veebar G_1) + G_3 (G_2 \barwedge G_1)'$ $\quad [x \veebar y = x'y + xy'], [(x \veebar y)' = xy + x'y']$

$B_1 = G_3 \barwedge G_2 \barwedge G_1$

$B_0 = G_3'G_2' G_1'G_0 + G_3'G_2 \barwedge G_1G_0' + G_3G_2G_1 \barwedge G_0 + G_3G_2G_1 G_0' + G_3'G_2G_1'G_0' + G_3G_2 \barwedge G_1'G_0' + G_3G_2 \barwedge G_1 G_0 + G_3'G_2G_1G_0 + G_3G_2 \barwedge G_1 G_0.$

$= G_3'G_2' (G_1'G_0 + G_1G_0') + G_3G_2 (G_1'G_0 + G_1G_0') + G_1'G_0' (G_3'G_2 + G_3G_2') + G_1G_0 (G_3'G_2 + G_3G_2').$

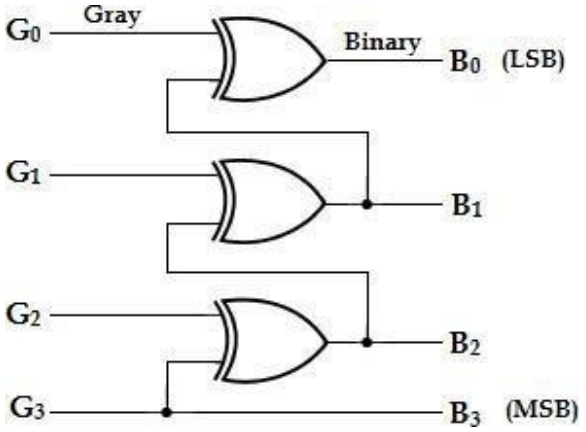
$= G_3'G_2' (G_0 \barwedge G_1) + G_3G_2 (G_0 \veebar G_1) + G_1'G_0' (G_2 \barwedge G_3) + G_1G_0 (G_2 \barwedge G_3).$

$= G_0 \veebar G_1 (G_3'G_2' + G_3G_2) + G_2 \veebar G_3 (G_1'G_0' + G_1G_0)$

$= (G_0 \barwedge G_1) (G_2 \barwedge G_3)' + (G_2 \barwedge G_3) (G_0 \barwedge G_1)$ $\quad [x \veebar y = x'y + xy']$

$B_0 = (G_0 \barwedge G_1) \barwedge (G_2 \barwedge G_3).$

Now, the above expressions can be implemented using EX-OR gates as,



Logic diagram of 4-bit gray-to-binary converter

3. BCD –to-Excess-3 Converters:

Excess-3 is a modified form of a BCD number. The excess-3 code can be derived from the natural BCD code by adding 3 to each coded number.

For example, decimal 12 can be represented in BCD as 0001 0010. Now adding 3 to each digit we get excess-3 code as 0100 0101 (12 in decimal). With this information the truth table for BCD to Excess-3 code converter can be determined as,

Truth Table:

Decimal	BCD code				Excess-3 code			
	B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

From the truth table, the logic expression for the Excess-3 code outputs can be written as,

$$E_3 = \sum_m (5, 6, 7, 8, 9) + \sum_d (10, 11, 12, 13, 14, 15)$$
$$E_2 = \sum_m (1, 2, 3, 4, 9) + \sum_d (10, 11, 12, 13, 14, 15)$$
$$E_1 = \sum_m (0, 3, 4, 7, 8) + \sum_d (10, 11, 12, 13, 14, 15)$$
$$E_0 = \sum_m (0, 2, 4, 6, 8) + \sum_d (10, 11, 12, 13, 14, 15)$$

K-map Simplification:

For E_3

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	1	1	1
11	x	x	x	x
10	1	1	x	x

$E_3 = B_3 + B_2 (B_0 + B_1)$

For E_2

B_3B_2 \ B_1B_0	00	01	11	10
00	0	1	1	1
01	1	0	0	0
11	x	x	x	x
10	0	1	x	x

$E_2 = B_2B_1'B_0' + B_2' (B_0 + B_1)$

For E_1

B_3B_2 \ B_1B_0	00	01	11	10
00	1	0	1	0
01	1	0	1	0
11	x	x	x	x
10	1	0	x	x

$E_1 = B_1'B_0' + B_1B_0$

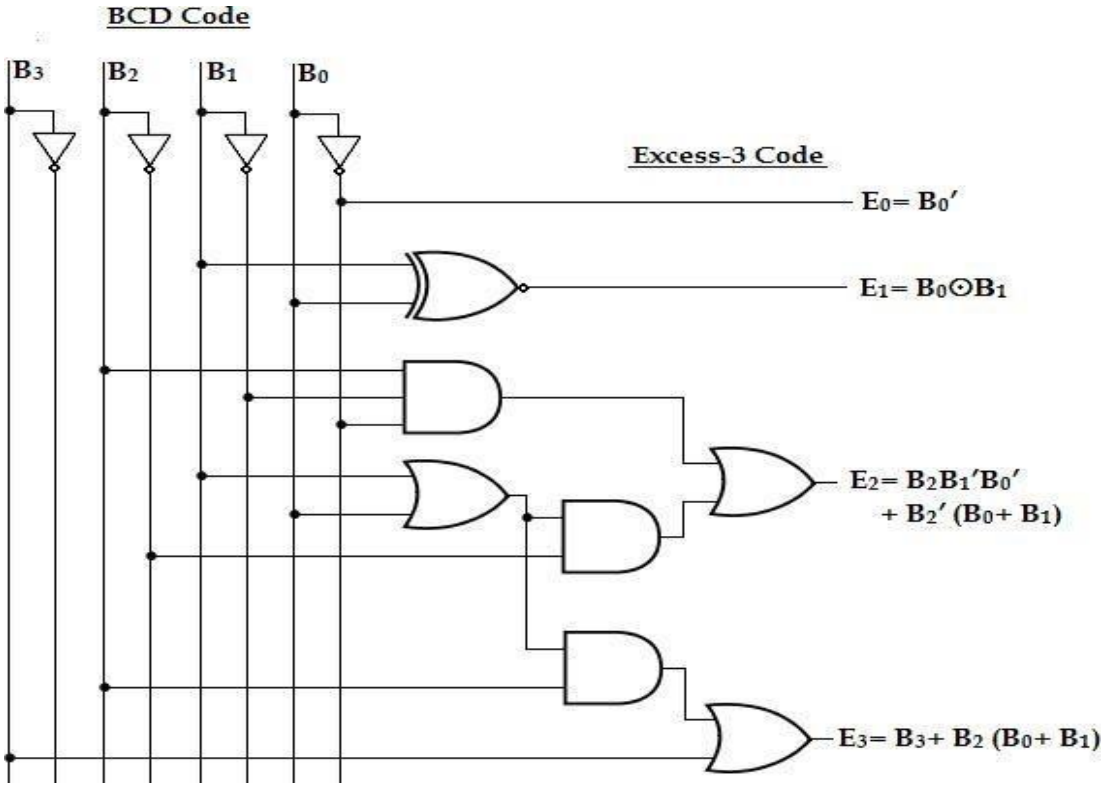
$= B_1 \odot B_0$

For E_0

B_3B_2 \ B_1B_0	00	01	11	10
00	1	0	0	1
01	1	0	0	1
11	x	x	x	x
10	1	0	x	x

$E_0 = B_0'$

Logic Diagram:



4. **Excess-3 to BCD**

Converter: Truth table:

Decimal	Excess-3 code				BCD code			
	E_3	E_2	E_1	E_0	B_3	B_2	B_1	B_0
3	0	0	1	1	0	0	0	0
4	0	1	0	0	0	0	0	1
5	0	1	0	1	0	0	1	0
6	0	1	1	0	0	0	1	1
7	0	1	1	1	0	1	0	0
8	1	0	0	0	0	1	0	1
9	1	0	0	1	0	1	1	0
10	1	0	1	0	0	1	1	1
11	1	0	1	1	1	0	0	0
12	1	1	0	0	1	0	0	1

From the truth table, the logic expression for the Excess-3 code outputs can be written as,

$B_3 = \sum_m (11, 12) + \sum_d (0, 1, 2, 13, 14, 15)$

$B_2 = \sum_m (7, 8, 9, 10) + \sum_d (0, 1, 2, 13, 14, 15)$

$B_1 = \sum_m (5, 6, 9, 10) + \sum_d (0, 1, 2, 13, 14, 15)$

$B_0 = \sum_m (4, 6, 8, 10, 12) + \sum_d (0, 1, 2, 13, 14, 15)$

K-map Simplification:

For B_3

E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	0	0	0	0
11	1	X	X	X
10	0	0	1	0

$B_3 = E_3E_2 + E_3E_1E_0$

For B_2

E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	0	0	1	0
11	0	X	X	X
10	1	1	0	1

$B_2 = E_2'E_1' + E_2E_1E_0 + E_3E_1E_0'$

For B_1

E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	0	1	0	1
11	0	X	X	X
10	0	1	0	1

$B_1 = E_1'E_0 + E_1E_0'$
$= E_1 \oplus E_0$

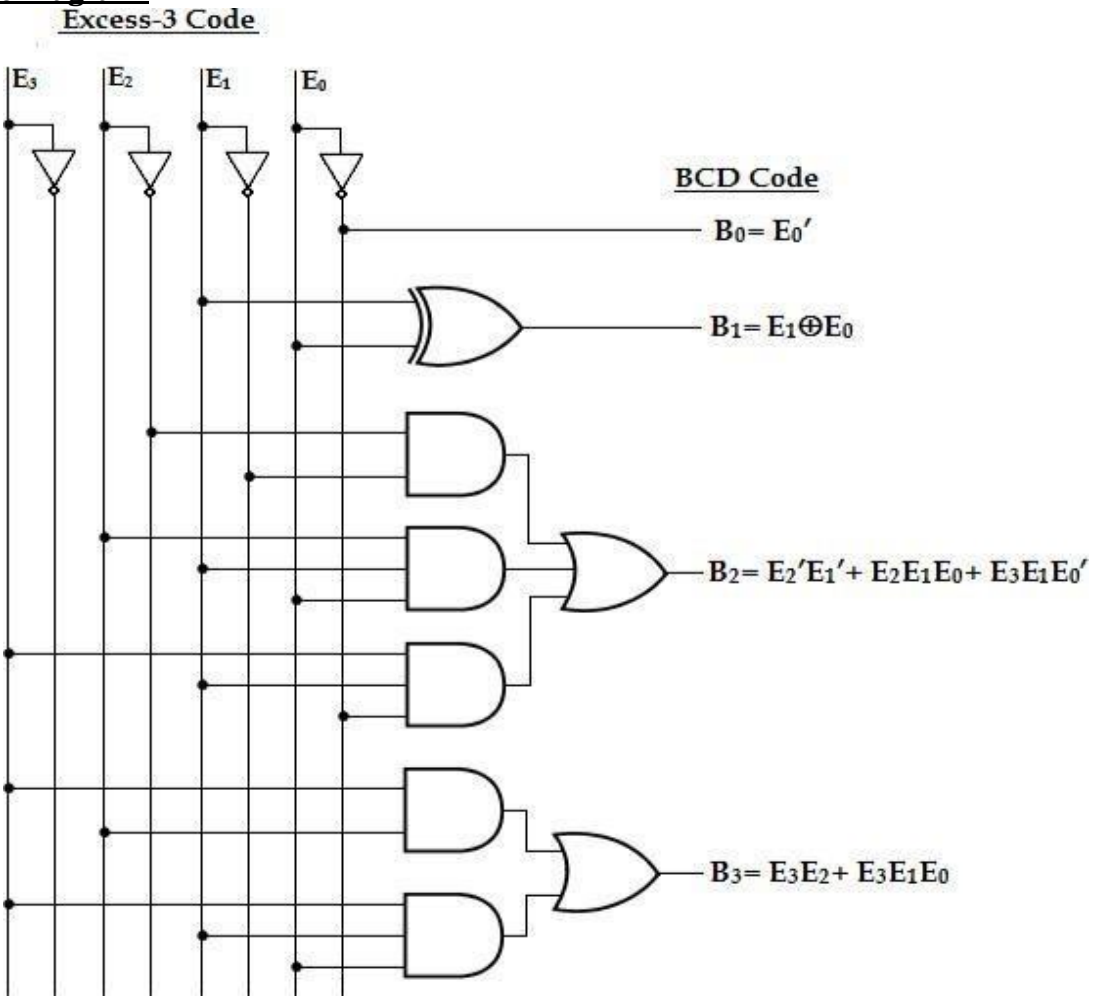
For B_0

E_3E_2 \ E_1E_0	00	01	11	10
00	X	X	0	X
01	1	0	0	1
11	1	X	X	X
10	1	0	0	1

$B_0 = E_0'$

Now, the above expressions the logic diagram can be implemented as,

Logic Diagram:



5. BCD –to-Binary Converters:

The steps involved in the BCD-to-binary conversion process are as follows:

1. The value of each bit in the BCD number is represented by a binary equivalent or weight.
2. All the binary weights of the bits that are 1‘s in the BCD are added.
3. The result of this addition is the binary equivalent of the BCD number.

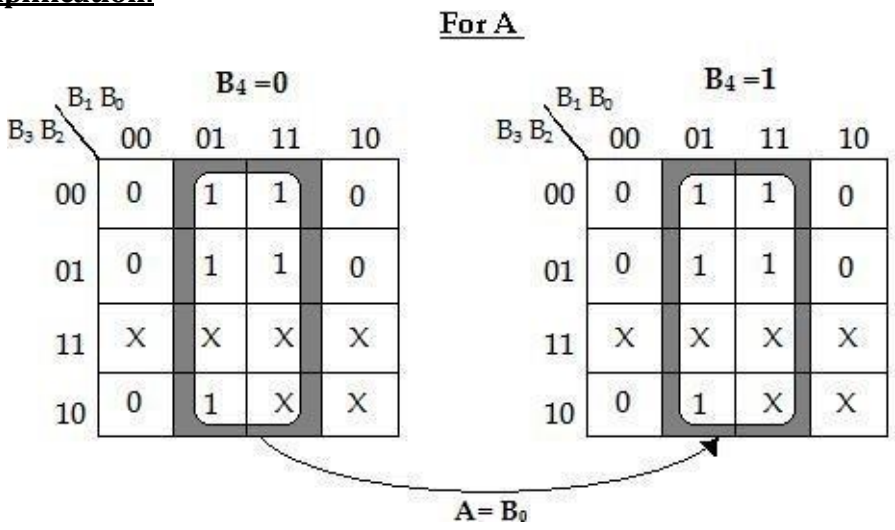
Two-digit decimal values ranging from 00 to 99 can be represented in BCD by two 4-bit code groups. For example, 19_{10} is represented as,

$$\underbrace{1}_{0001}\quad\underbrace{9}_{1001}$$

The left-most four-bit group represents 10 and right-most four-bit group represents 9. The binary representation for decimal 19 is $19_{10} = 11001_2$.

BCD Code					Binary				
B_4	B_3	B_2	B_1	B_0	E	D	C	B	A
0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	0	0	1	0
0	0	0	1	1	0	0	0	1	1
0	0	1	0	0	0	0	1	0	0
0	0	1	0	1	0	0	1	0	1
0	0	1	1	0	0	0	1	1	0
0	0	1	1	1	0	0	1	1	1
0	1	0	0	0	0	1	0	0	0
0	1	0	0	1	0	1	0	0	1
1	0	0	0	0	0	1	0	1	0
1	0	0	0	1	0	1	0	1	1
1	0	0	1	0	0	1	1	0	0
1	0	0	1	1	0	1	1	0	1
1	0	1	0	0	0	1	1	1	0
1	0	1	0	1	0	1	1	1	1
1	0	1	1	0	1	0	0	0	0
1	0	1	1	1	1	0	0	0	1
1	1	0	0	0	1	0	0	1	0
1	1	0	0	1	1	0	0	1	1

K-map Simplification:



For B

B_4 =0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	X	X	X	X
10	0	0	X	X

B_4 =1

B_3B_2 \ B_1B_0	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	X	X	X	X
10	1	1	X	X

$$B = B_1B_4' + B_1'B_4$$
$$= B_1 \oplus B_4$$

For C

B_4 =0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	X	X	X	X
10	0	0	X	X

B_4 =1

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$$C = B_4'B_2 + B_2B_1' + B_4B_2'B_1$$

For D

B_4 =0

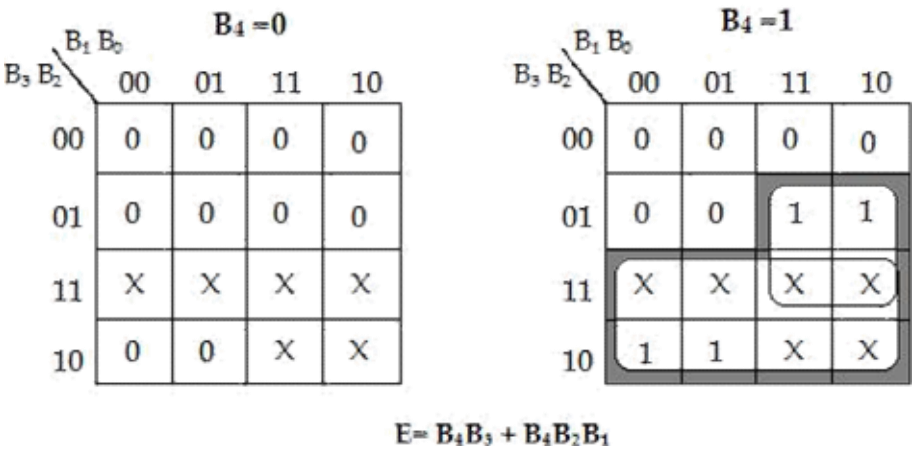
B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	1	1	X	X

B_4 =1

B_3B_2 \ B_1B_0	00	01	11	10
00	1	1	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$$D = B_4'B_3 + B_4B_3'B_2' + B_4B_3'B_1'$$

For E



$E = B_4B_3 + B_4B_2B_1$

From the above K-map,

A= B_0

B= $B_1B_4' + B_1'B_4$
= $B_1 \perp B_4$

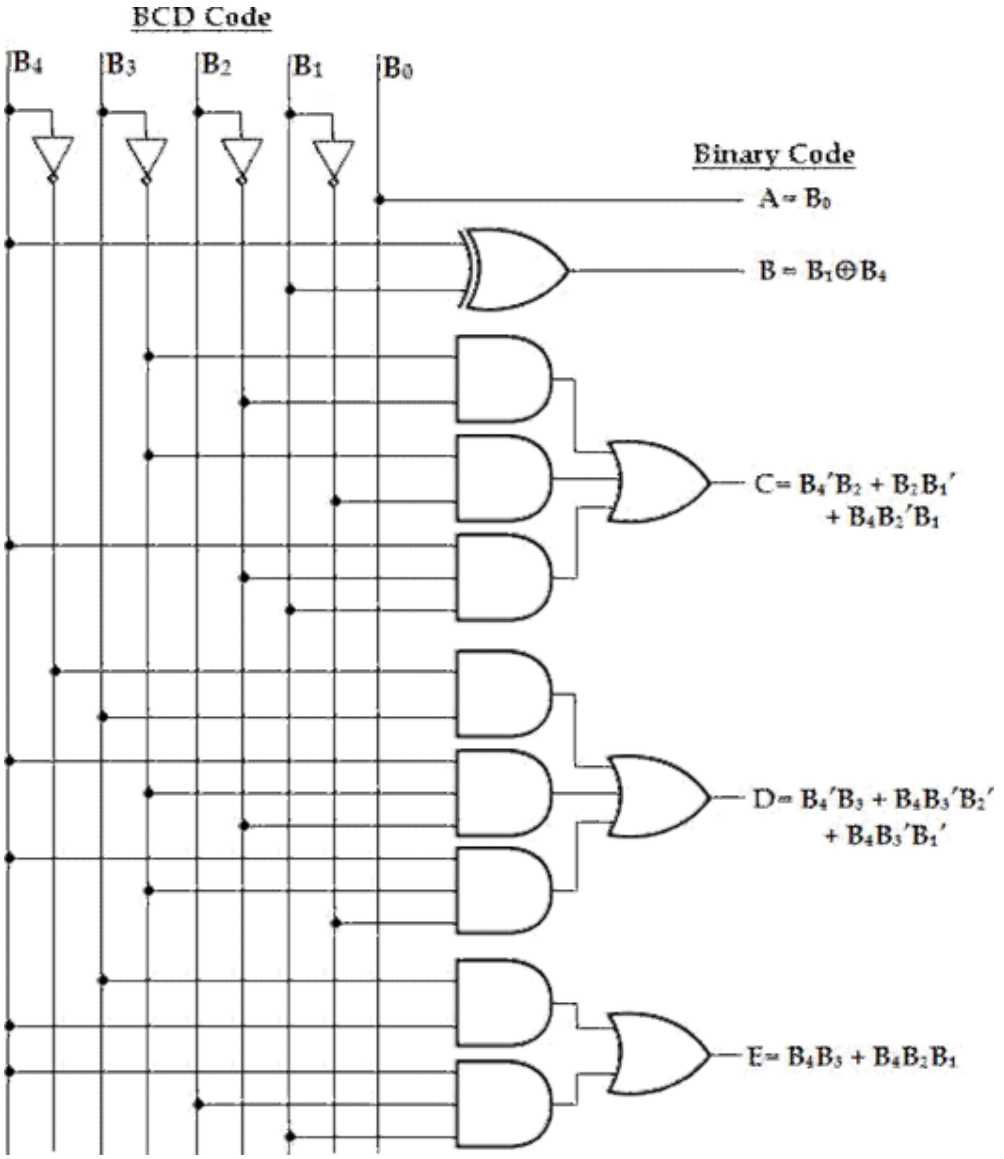
C= $B_4'B_2 + B_2B_1' + B_4B_2'B_1$

D= $B_4'B_3 + B_4B_3'B_2' + B_4B_3'B_1'$

E= $B_4B_3 + B_4B_2B_1$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:


BCD Code
B4
B3
B2
B1
B0
Binary Code
A = B0
B = B1⊕B4
C = B4′B2 + B2B1′
+ B4B2′B1
D = B4′B3 + B4B3′B2′
+ B4B3′B1′
E = B4B3 + B4B2B1

6. **Binary to BCD Converter:**

The truth table for binary to BCD converter can be written as,

Truth Table:

Decimal	Binary Code				BCD Code				
	D	C	B	A	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1
2	0	0	1	0	0	0	0	1	0
3	0	0	1	1	0	0	0	1	1
4	0	1	0	0	0	0	1	0	0
5	0	1	0	1	0	0	1	0	1
6	0	1	1	0	0	0	1	1	0
7	0	1	1	1	0	0	1	1	1
8	1	0	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	0	1
10	1	0	1	0	1	0	0	0	0
11	1	0	1	1	1	0	0	0	1
12	1	1	0	0	1	0	0	1	0
13	1	1	0	1	1	0	0	1	1
14	1	1	1	0	1	0	1	0	0
15	1	1	1	1	1	0	1	0	1

From the truth table, the logic expression for the BCD code outputs can be written as,

$B_0 = \sum_m (1, 3, 5, 7, 9, 11, 13, 15)$

$B_1 = \sum_m (2, 3, 6, 7, 12, 13)$

$B_2 = \sum_m (4, 5, 6, 7, 14, 15)$

$B_3 = \sum_m (8, 9)$

$B_4 = \sum_m (10, 11, 12, 13, 14, 15)$

K-map Simplification:

For B_0

DC \ BA	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$B_0 = A$

For B_1

DC \ BA	00	01	11	10
00	0	0	1	1
01	0	0	1	1
11	1	1	0	0
10	0	0	0	0

$B_1 = DCB' + D'B$

For B_2

DC \ BA	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	1	1
10	0	0	0	0

$B_2 = D'C + CB$

For B_3

DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	0	0
10	1	1	0	0

$B_3 = DC'B'$

For B_4

DC \ BA	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	1	1
10	0	0	1	1

$B_4 = DC + DB$

From the above K-map, the logical expression can be obtained as, **$B_0 = A$**

$B_1 = DCB' + D'B$

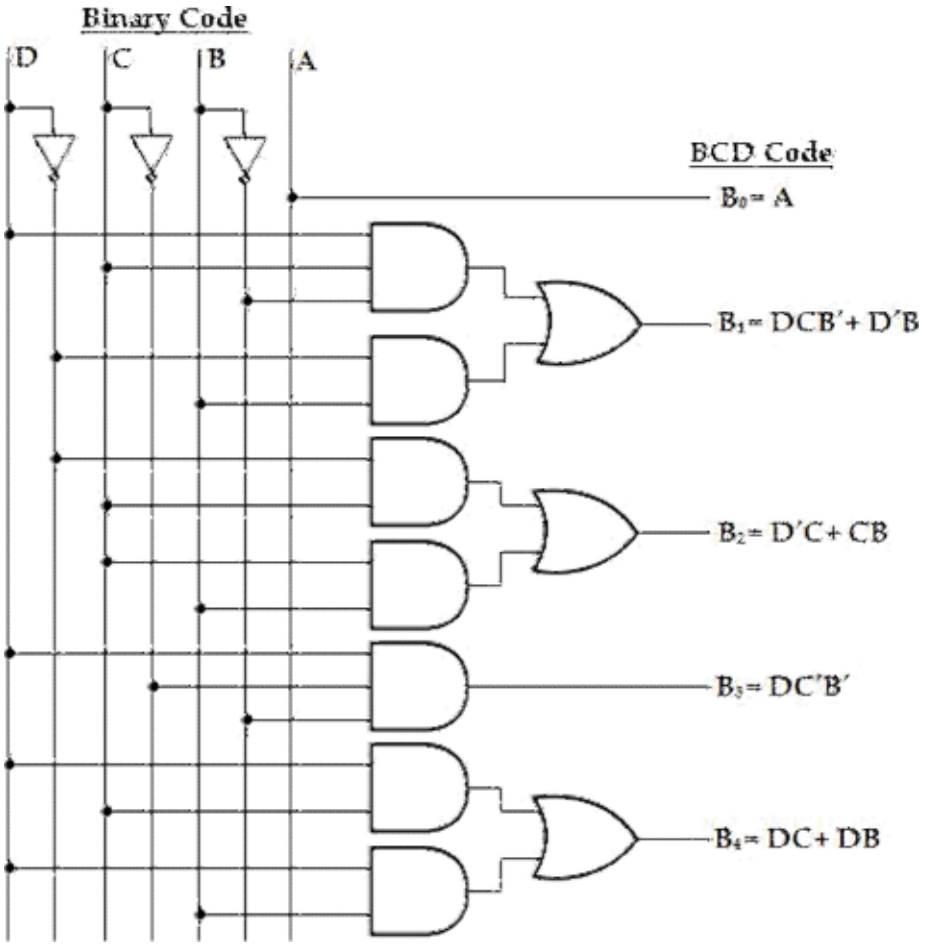
$B_2 = D'C + CB$

$B_3 = DC'B'$

$B_4 = DC + DB$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



7. **Gray to BCD Converter:**
   The truth table for gray to BCD converter can be written as,

Truth Table:

Gray Code				BCD Code				
G_3	G_2	G_1	G_0	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	1
0	0	1	1	0	0	0	1	0
0	0	1	0	0	0	0	1	1
0	1	1	0	0	0	1	0	0
0	1	1	1	0	0	1	0	1
0	1	0	1	0	0	1	1	0
0	1	0	0	0	0	1	1	1
1	1	0	0	0	1	0	0	0

1	1	0	1	0	1	0	0	1
1	1	1	1	1	0	0	0	0
1	1	1	0	1	0	0	0	1
1	0	1	0	1	0	0	1	0
1	0	1	1	1	0	0	1	1
1	0	0	1	1	0	1	0	0
1	0	0	0	1	0	1	0	1

K-map Simplification:

For B_0

G_3G_2 \ G_1G_0	00	01	11	10
00	0	1	0	1
01	1	0	1	0
11	0	1	0	1
10	1	0	1	0

$B_0 = (G_0 \oplus G_1) \oplus (G_2 \oplus G_3)$

For B_1

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	0	0	0	0
10	0	0	1	1

$B_1 = G'_2G_1 + G'_3G_2G'_1$

For B_2

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	0	0	0	0
10	1	1	0	0

$B_2 = G'_3G_2 + G_3G'_2G'_1$

For B_3

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	1	1	0	0
10	0	0	0	0

$B_3 = G_3G_2G'_1$

For B_4

G_3G_2 \ G_1G_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	0	0	1	1
10	1	1	1	1

$B_4 = G_3G'_2 + G_3G_1$

From the above K-map, the logical expression can be obtained as,

$B_0 = (G_0 \llcorner G_1) \llcorner (G_2 \llcorner G_3)$

$B_1 = G'_2G_1 + G'_3G_2G'_1$

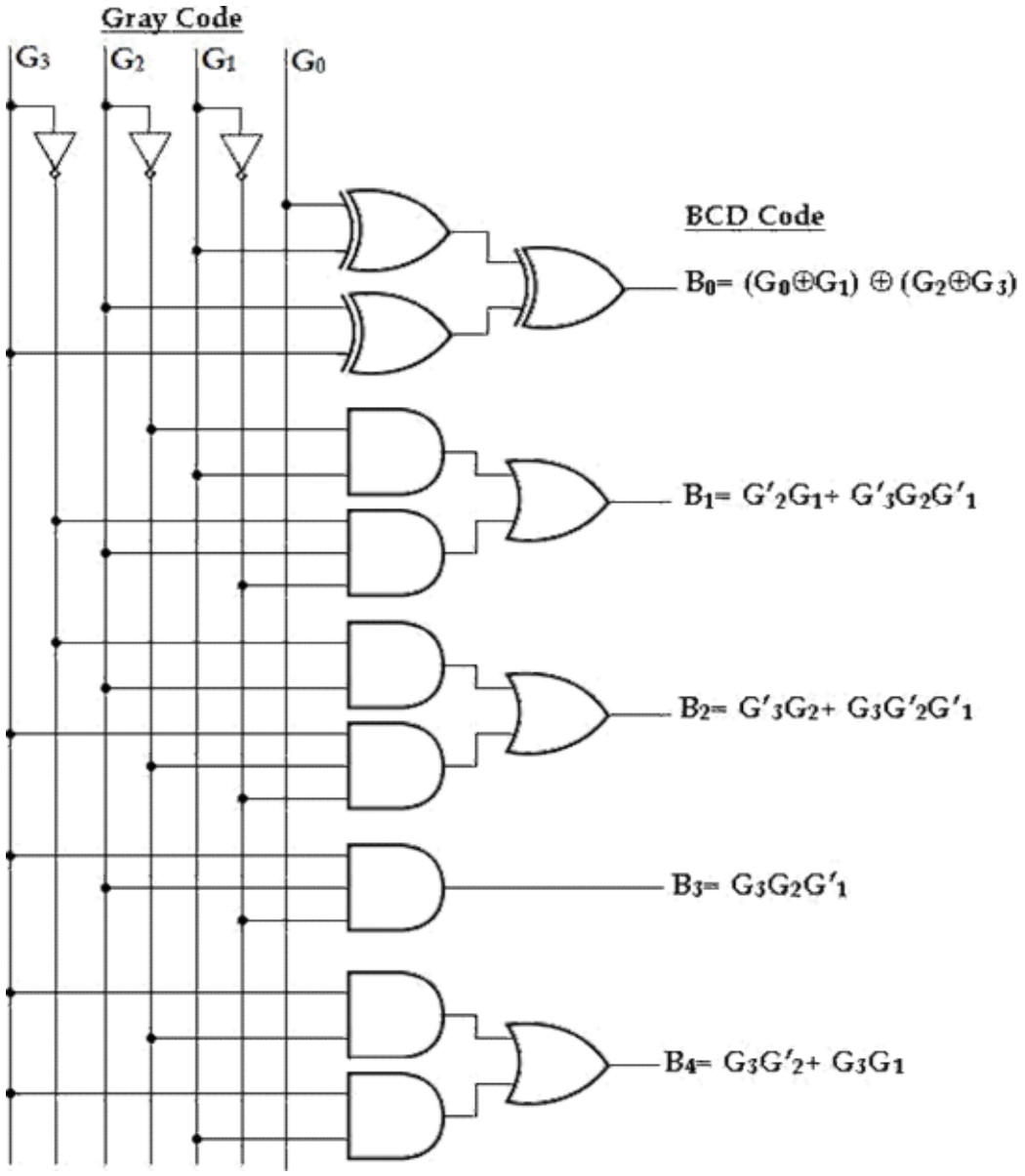
$B_2 = G'_3G_2 + G_3G'_2G'_1$

$B_3 = G_3G_2G'_1$

$B_4 = G_3G'_2 + G_3G_1$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



8. <u>BCD to Gray Converter:</u>

The truth table for gray to BCD converter can be written as,

<u>Truth table:</u>

BCD Code (8421)				Gray code			
B_3	**B_2**	**B_1**	**B_0**	**G_3**	**G_2**	**G_1**	**G_0**
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0

0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1

K-map Simplification:

For G_3

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	0	0	0	0
11	X	X	X	X
10	1	1	X	X

$G_3 = B_3$

For G_2

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	0	0
01	1	1	1	1
11	X	X	X	X
10	1	1	X	X

$G_2 = B_3 + B_2$

For G_1

B_3B_2 \ B_1B_0	00	01	11	10
00	0	0	1	1
01	1	1	0	0
11	X	X	X	X
10	0	0	X	X

$G_1 = B_2'B_1 + B_2B_1'$
$= B_2 \oplus B_1$

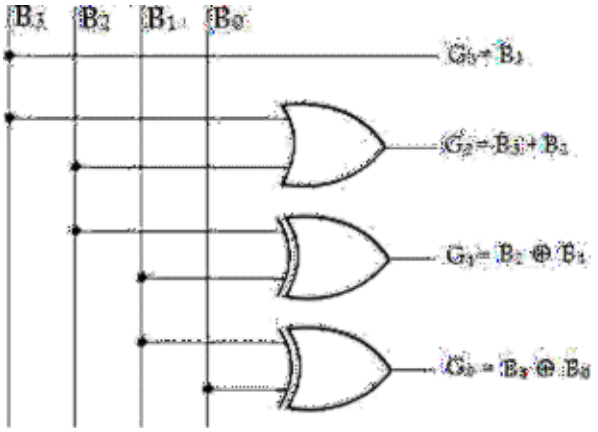
For G_0

B_3B_2 \ B_1B_0	00	01	11	10
00	0	1	0	1
01	0	1	0	1
11	X	X	X	X
10	0	1	X	X

$G_0 = B_1'B_0 + B_1B_0'$
$= B_1 \oplus B_0$

Now, from the above expressions the logic diagram can be implemented as,

Logic Diagram:



9. **8 4 -2 -1 to BCD Converter:**

The truth table for 8 4 -2 -1 to BCD converter can be written as,

Truth Table:

Gray Code				BCD Code				
D	**C**	**B**	**A**	**B_4**	**B_3**	**B_2**	**B_1**	**B_0**
0	0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0	1
0	1	1	0	0	0	0	1	0
0	1	0	1	0	0	0	1	1
0	1	0	0	0	0	1	0	0
1	0	1	1	0	0	1	0	1
1	0	1	0	0	0	1	1	0
1	0	0	1	0	0	1	1	1
1	0	0	0	0	1	0	0	0
1	1	1	1	0	1	0	0	1
1	1	1	0	1	0	0	0	0
1	1	0	1	1	0	0	0	1
1	1	0	0	1	0	0	1	0

K-map Simplification:

For B_0

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	1	1	0
11	0	1	1	0
10	0	1	1	0

$B_0 = A$

For B_1

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	1	0	1
11	1	0	0	0
10	0	1	0	1

$B_1 = DCB'A' + D'B'A + D'BA' + C'B'A + C'BA'$
$= A'B'CD + D'(B'A + BA') + C'(B'A + BA')$
$= A'B'CD + D'(A \oplus B) + C'(A \oplus B)$
$= A'B'CD + (A \oplus B)(C' + D')$

For B_2

DC \ BA	00	01	11	10
00	0	X	X	X
01	1	0	0	0
11	0	0	0	0
10	0	1	1	1

$B_2 = D'CB'A' + C'A + C'B$
$= D'CB'A' + C'(A + B)$

For B_3

DC \ BA	00	01	11	10
00	0	X	X	X
01	0	0	0	0
11	0	0	1	0
10	1	0	0	0

$B_3 = ABCD + A'B'C'D$
$= D(ABC + A'B'C')$

For B_4

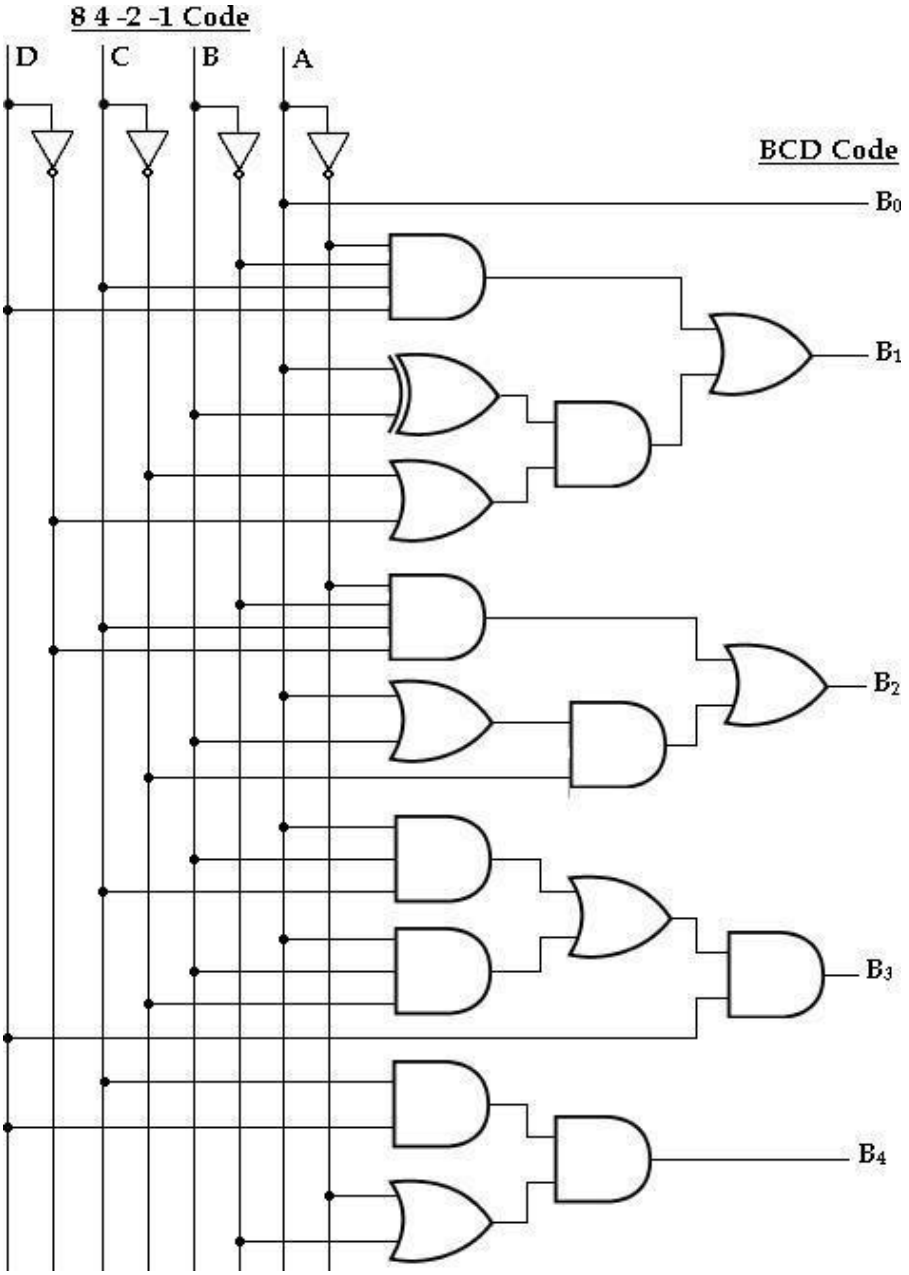
DC \ BA	00	01	11	10
00	0	X	X	X
01	0	0	0	0
11	1	1	0	1
10	0	0	0	0

$B_4 = B'CD + A'CD$
$= CD(A' + B')$

From the above K-map, the logical expression can be obtained as,

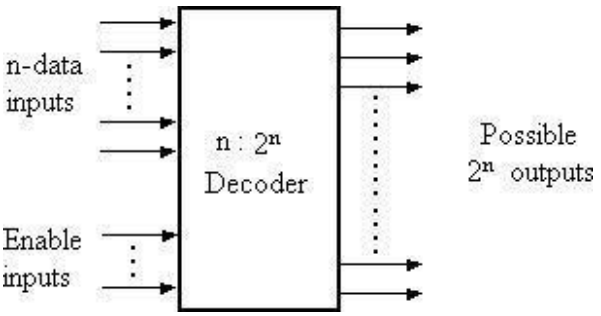
B_0= A
B_1= A'B'CD+ (A⊕B) (C'+D')
B_2= D'CB'A'+ C' (A+B)
B_3= D (ABC+ A'B'C')
B_4= CD (A'+B')

Logic Diagram:



DECODERS:

A decoder is a combinational circuit that converts binary information from ‗n' input lines to a maximum of ‗2^n' unique output lines. The general structure of decoder circuit is –

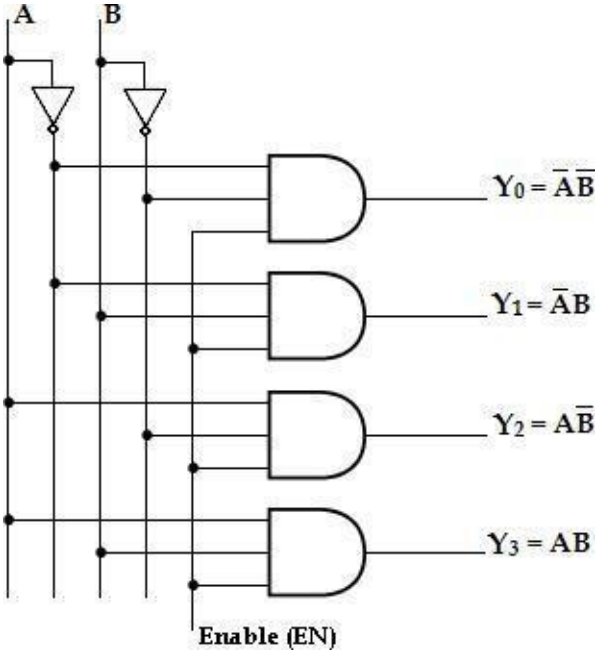


General structure of decoder

The encoded information is presented as ‗n' inputs producing ‗2^n' possible outputs. The 2^n output values are from 0 through 2^n-1. A decoder is provided with enable inputs to activate decoded output based on data inputs. When any one enable input is unasserted, all outputs of decoder are disabled.

Binary Decoder (2 to 4 decoder):

A binary decoder has ‗n' bit binary input and a one activated output out of 2^n outputs. A binary decoder is used when it is necessary to activate exactly one of 2^n outputs based on an n-bit input value.



2-to-4 Line decoder

Here the 2 inputs are decoded into 4 outputs, each output representing one of the minterms of the two input variables.

Inputs			Outputs			
Enable	**A**	**B**	Y_3	Y_2	Y_1	Y_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

As shown in the truth table, if enable input is 1 (EN= 1) only one of the outputs (Y_0 – Y_3), is active for a given input.

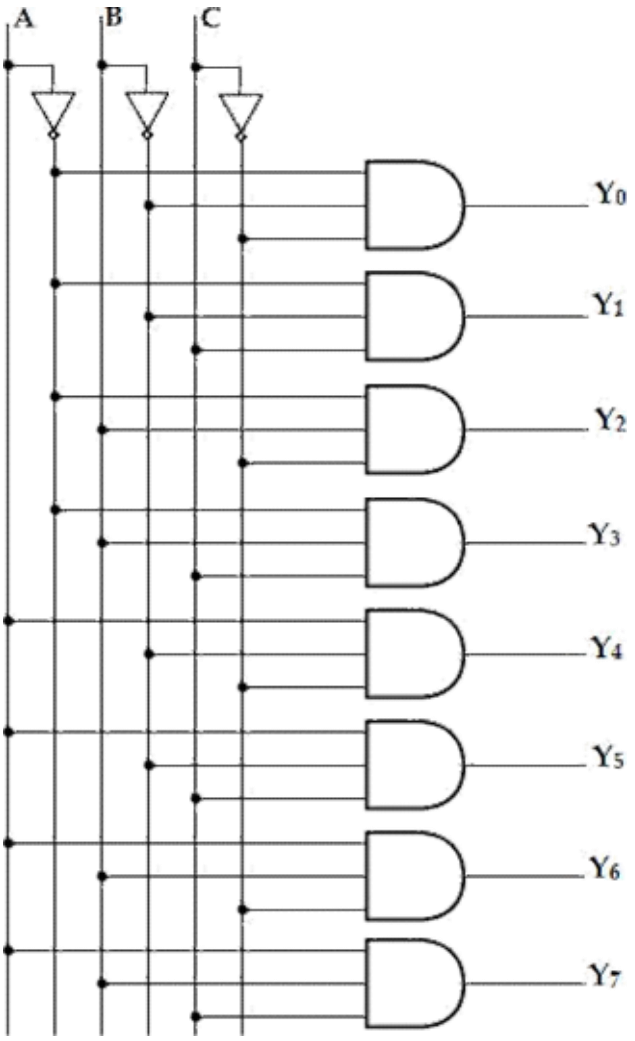
The output Y_0 is active, ie., Y_0= 1 when inputs A= B= 0,
Y_1 is active when inputs, A= 0 and B= 1,
Y_2 is active, when input A= 1 and B= 0,
Y_3 is active, when inputs A= B= 1.

3- to-8 Line Decoder:

A 3-to-8 line decoder has three inputs (A, B, C) and eight outputs (Y_0- Y_7). Based on the 3 inputs one of the eight outputs is selected.

The three inputs are decoded into eight outputs, each output representing one of the minterms of the 3-input variables. This decoder is used for binary-to-octal conversion. The input variables may represent a binary number and the outputs will represent the eight digits in the octal number system. The output variables are mutually exclusive because only one output can be equal to 1 at any one time. The output line whose value is equal to 1 represents the minterm equivalent of the binary number presently available in the input lines.

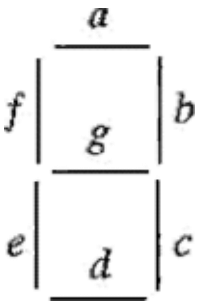
Inputs			Outputs							
A	**B**	**C**	Y_0	Y_1	Y_2	Y_3	Y_4	Y_5	Y_6	Y_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1



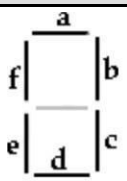
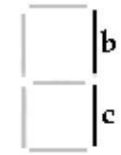
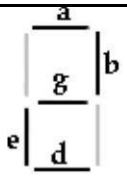
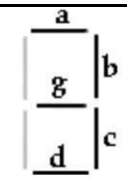
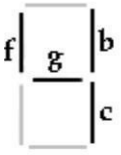
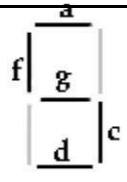
3-to-8 line decoder

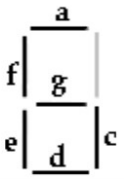
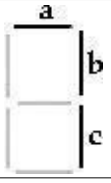
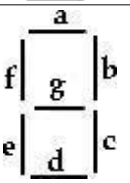
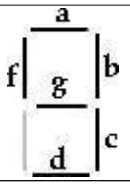
BCD to 7-Segment Display Decoder:

A seven-segment display is normally used for displaying any one of the decimal digits, 0 through 9. A BCD-to-seven segment decoder accepts a decimal digit in BCD and generates the corresponding seven-segment code.



Each segment is made up of a material that emits light when current is passed through it. The segments activated during each digit display are tabulated as—

Digit	Display	Segments Activated
0		a, b, c, d, e, f
1		b, c
2		a, b, d, e, g
3		a, b, c, d, g
4		b, c, f, g
5		a, c, d, f, g

6		a, c, d, e, f, g
7		a, b, c
8		a, b, c, d, e, f, g
9		a, b, c, d, f, g

Truth table:

	BCD code				7-Segment code						
Digit	**A**	**B**	**C**	**D**	**a**	**b**	**c**	**d**	**e**	**f**	**g**
0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	1	1	0	0	0	0
2	0	0	1	0	1	1	0	1	1	0	1
3	0	0	1	1	1	1	1	1	0	0	1
4	0	1	0	0	0	1	1	0	0	1	1
5	0	1	0	1	1	0	1	1	0	1	1
6	0	1	1	0	1	0	1	1	1	1	1
7	0	1	1	1	1	1	1	0	0	0	0
8	1	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	1	1	1	1	0	1	1

K-map Simplification:

For (a)

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	1	1
11	X	X	X	X
10	1	1	X	X

a= A+ C+ BD+ B'D'

For (b)

AB \ CD	00	01	11	10
00	1	1	1	1
01	1	0	1	0
11	X	X	X	X
10	1	1	X	X

b= B'+ C'D'+ CD

For (c)

AB \ CD	00	01	11	10
00	1	1	1	0
01	1	1	1	1
11	X	X	X	X
10	1	1	X	X

c= B+ C'+ D

For (d)

AB \ CD	00	01	11	10
00	1	0	1	1
01	0	1	0	1
11	X	X	X	X
10	1	1	X	X

d= B'D'+ CD'+ BC'D+ B'C+ A

For (e)

AB \ CD	00	01	11	10
00	1	0	0	1
01	0	0	0	1
11	X	X	X	X
10	1	0	X	X

e= B'D'+ CD'

For (f)

AB \ CD	00	01	11	10
00	1	0	0	0
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

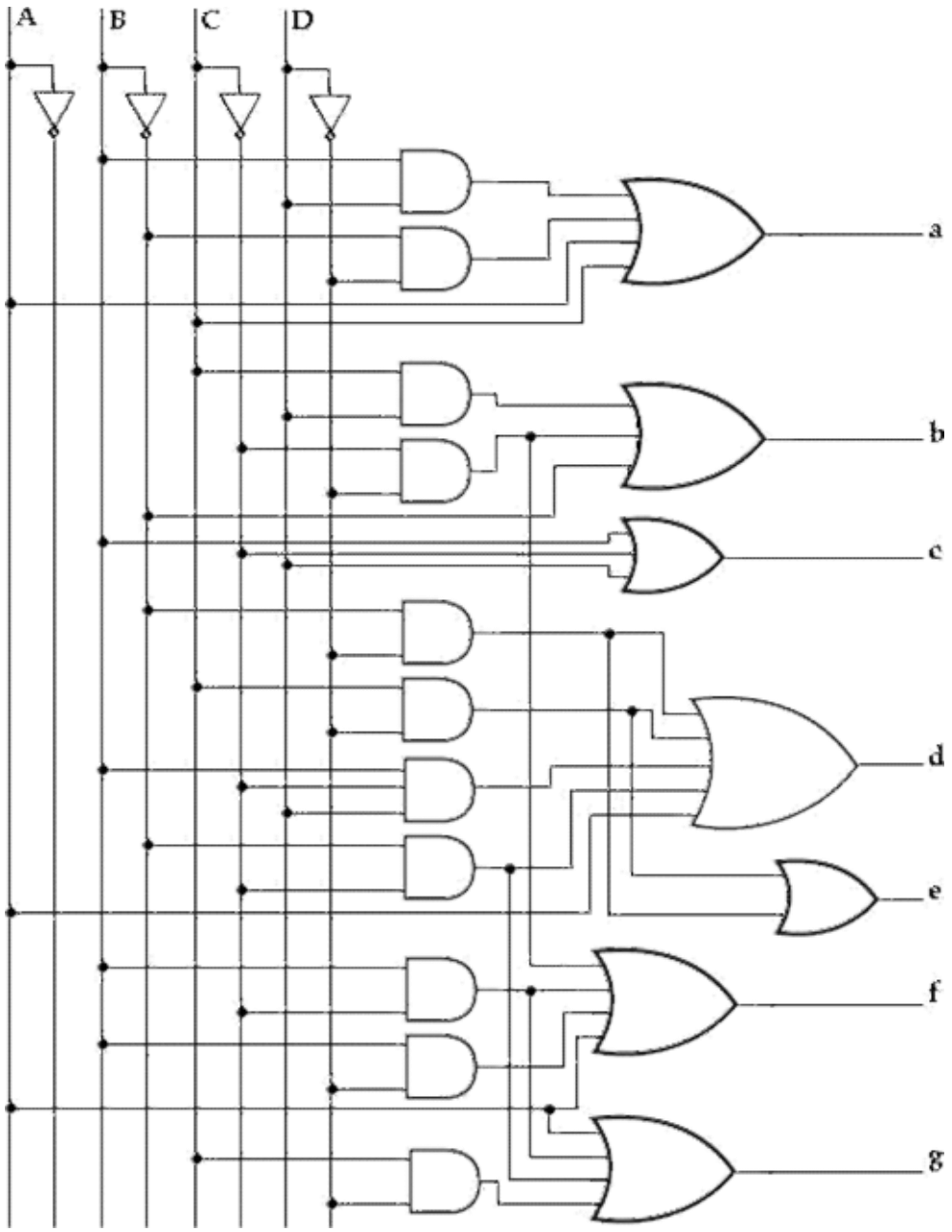
f= A+ C'D'+ BC'+ BD'

For (g)

AB \ CD	00	01	11	10
00	0	0	1	1
01	1	1	0	1
11	X	X	X	X
10	1	1	X	X

g= A+ BC'+ B'C+ CD'

Logic Diagram:



BCD to 7-segment display decoder

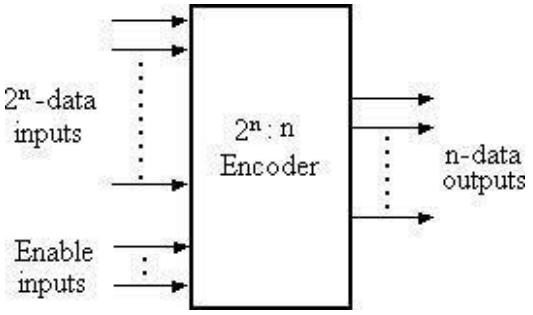
Applications of decoders:

1. Decoders are used in counter system.
2. They are used in analog to digital converter.
3. Decoder outputs can be used to drive a display system.

ENCODERS:

An encoder is a digital circuit that performs the inverse operation of a decoder. Hence, the opposite of the decoding process is called encoding. An encoder is a combinational circuit that converts binary information from 2^n input lines to a maximum of _n' unique output lines.

The general structure of encoder circuit is –



General structure of Encoder

It has 2^n input lines, only one which 1 is active at any time and _n' output lines. It encodes one of the active inputs to a coded binary output with _n' bits. In an encoder, the number of outputs is less than the number of inputs.

Octal-to-Binary Encoder:

It has eight inputs (one for each of the octal digits) and the three outputs that generate the corresponding binary number. It is assumed that only one input has a value of 1 at any given time.

Inputs								Outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	A	B	C
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
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	1	0
0	0	0	0	0	0	0	1	1	1	1

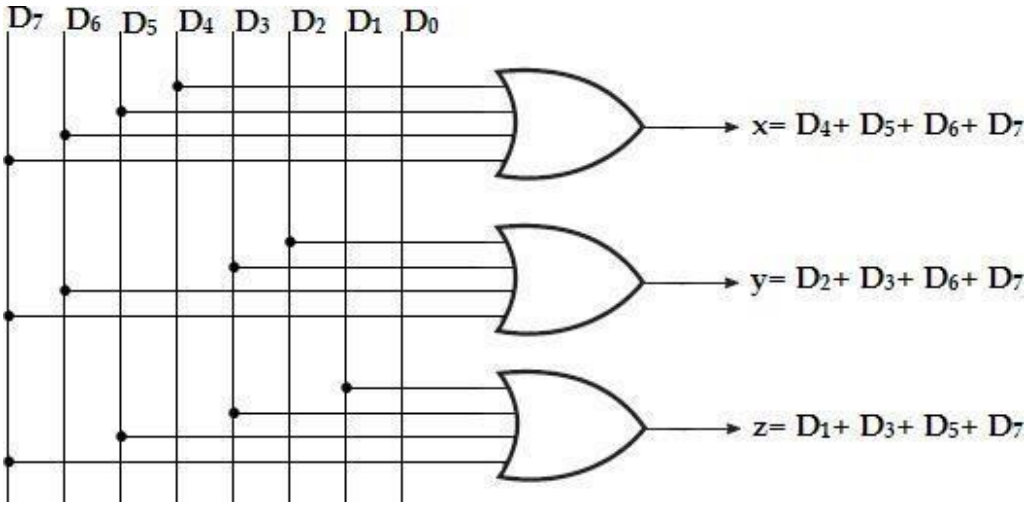
The encoder can be implemented with OR gates whose inputs are determined directly from the truth table. Output z is equal to 1, when the input octal digit is 1 or 3 or 5 or 7. Output y is 1 for octal digits 2, 3, 6, or 7 and the output is 1 for digits 4, 5, 6 or 7. These conditions can be expressed by the following output Boolean

functions: $\mathbf{z = D_1 + D_3 + D_5 + D_7}$

$y = D_2 + D_3 + D_6 + D_7$
$x = D_4 + D_5 + D_6 + D_7$

The encoder can be implemented with three OR gates. The encoder defined in the below table, has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination.

For eg., if D_3 and D_6 are 1 simultaneously, the output of the encoder may be 111. This does not represent either D_6 or D_3. To resolve this problem, encoder circuits must establish an input priority to ensure that only one input is encoded. If we establish a higher priority for inputs with higher subscript numbers and if D_3 and D_6 are 1 at the same time, the output will be 110 because D_6 has higher priority than D_3.



Octal-to-Binary Encoder

Another problem in the octal-to-binary encoder is that an output with all 0's is generated when all the inputs are 0; this output is same as when D_0 is equal to 1. The discrepancy can be resolved by providing one more output to indicate that atleast one input is equal to 1.

Priority Encoder:

A priority encoder is an encoder circuit that includes the priority function. In priority encoder, if two or more inputs are equal to 1 at the same time, the input having the highest priority will take precedence.

In addition to the two outputs x and y, the circuit has a third output, V (valid bit indicator). It is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid input and V is equal to 0.

The higher the subscript number, higher the priority of the input. Input D_3, has the highest priority. So, regardless of the values of the other inputs, when D_3 is 1, the output for xy is 11.

D_2 has the next priority level. The output is 10, if D_2= 1 provided D_3= 0. Theoutput for D_1 is generated only if higher priority inputs are 0, and so on down the priority levels.

Truth table:

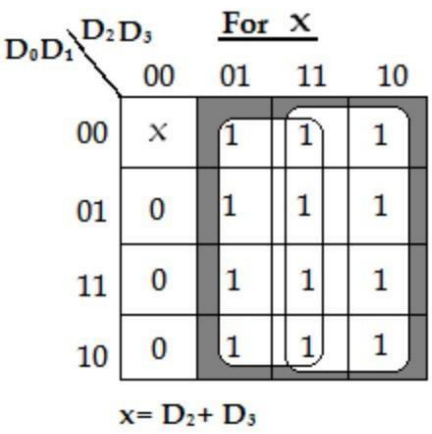
Inputs				Outputs		
D_0	**D_1**	**D_2**	**D_3**	**x**	**y**	**V**
0	0	0	0	x	x	0
1	0	0	0	0	0	1
x	1	0	0	0	1	1
x	x	1	0	1	0	1
x	x	x	1	1	1	1

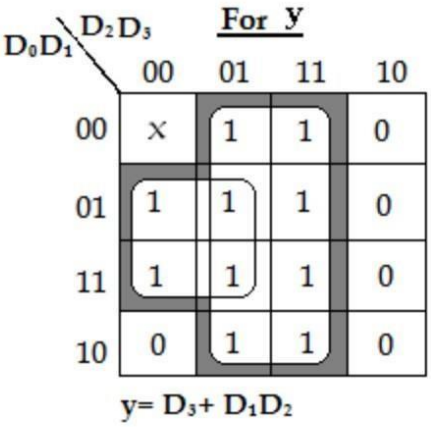
Although the above table has only five rows, when each don‘t care condition is replaced first by 0 and then by 1, we obtain all 16 possible input combinations. For example, the third row in the table with X100 represents minterms 0100 and 1100. The don‘t care condition is replaced by 0 and 1 as shown in the table below.

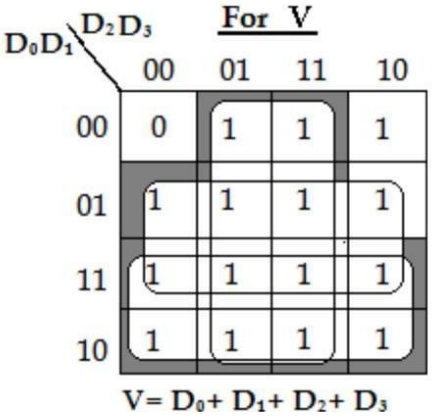
Modified Truth table:

Inputs				Outputs		
D_0	**D_1**	**D_2**	**D_3**	**x**	**y**	**V**
0	0	0	0	x	x	0
1	0	0	0	0	0	1
0 1	1 1	0 0	0 0	0	1	1
0 0 1 1	0 1 0 1	1 1 1 0	0 0 0 0	1	0	1
0 0 0 0 1 1 1	0 0 1 1 0 0 1	0 1 0 1 0 1 0	1 1 1 1 1 1 1	1	1	1
1	1	1	1			

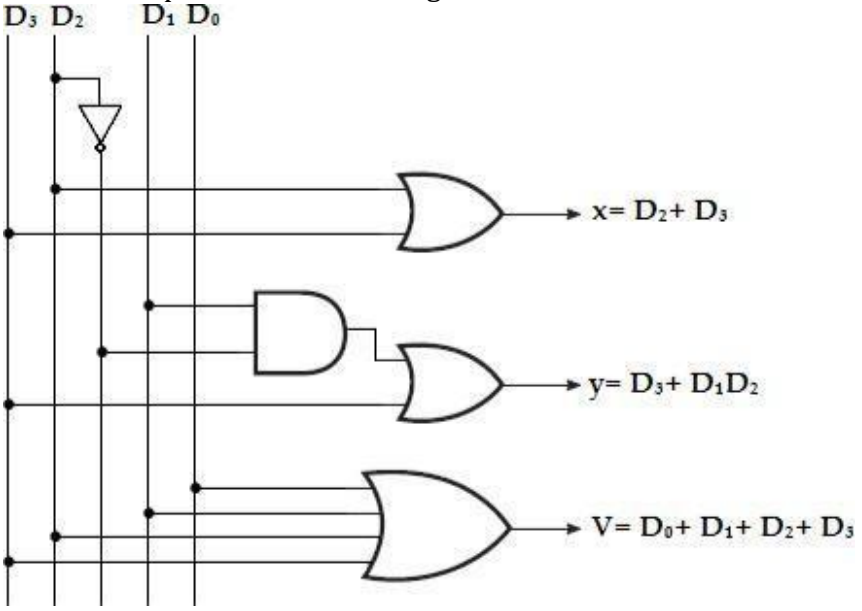
K-map Simplification:







The priority encoder is implemented according to the above Boolean functions.

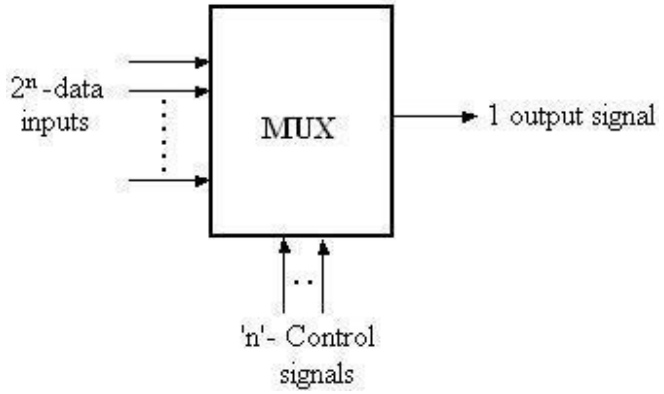


4- **Input Priority Encoder**

MULTIPLEXER: (Data Selector)

A ***multiplexer*** or *MUX*, is a combinational circuit with more than one input line, one output line and more than one selection line. A multiplexer selects binary information present from one of many input lines, depending upon the logic status of the selection inputs, and routes it to the output line. Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected. The multiplexer is often labeled as MUX in block diagrams.

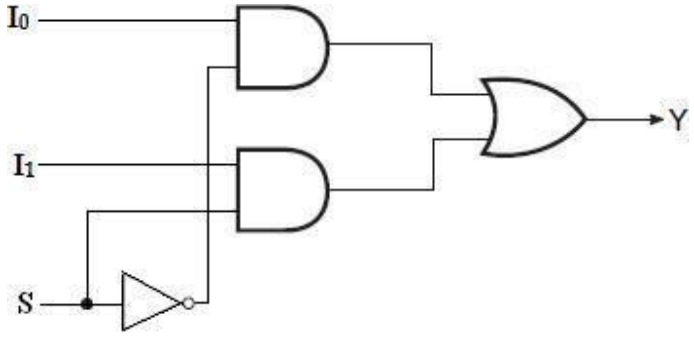
A multiplexer is also called a **data selector**, since it selects one of many inputs and steers the binary information to the output line.



Block diagram of Multiplexer

2-to-1- line Multiplexer:

The circuit has two data input lines, one output line and one selection line, S.
When S= 0, the upper AND gate is enabled and I_0 has a path to the output.
When S=1, the lower AND gate is enabled and I_1 has a path to the output.



Logic diagram

The multiplexer acts like an electronic switch that selects one of the two sources.

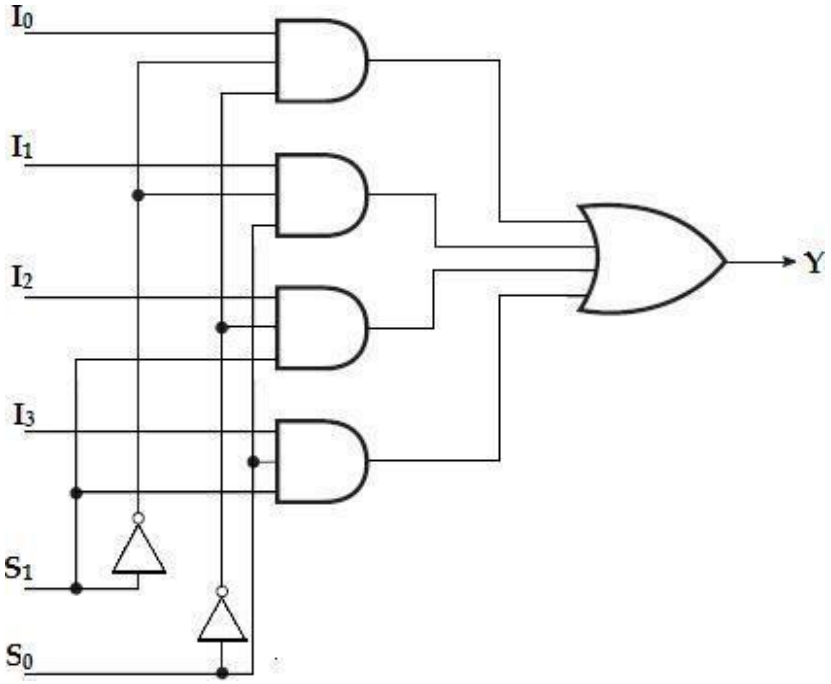
Truth table:

S	Y
0	I_0
1	I_1

4-to-1-line Multiplexer:

A 4-to-1-line multiplexer has four (2_n) input lines, two (n) select lines and one output line. It is the multiplexer consisting of four input channels and information of one of the channels can be selected and transmitted to an output line according to the select inputs combinations. Selection of one of the four input channel is possible by two selection inputs.

Each of the four inputs I_0 through I_3, is applied to one input of AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate. The outputs of the AND gate are applied to a single OR gate that provides the 1-line output.



4-to-1-Line Multiplexer

Function table:

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

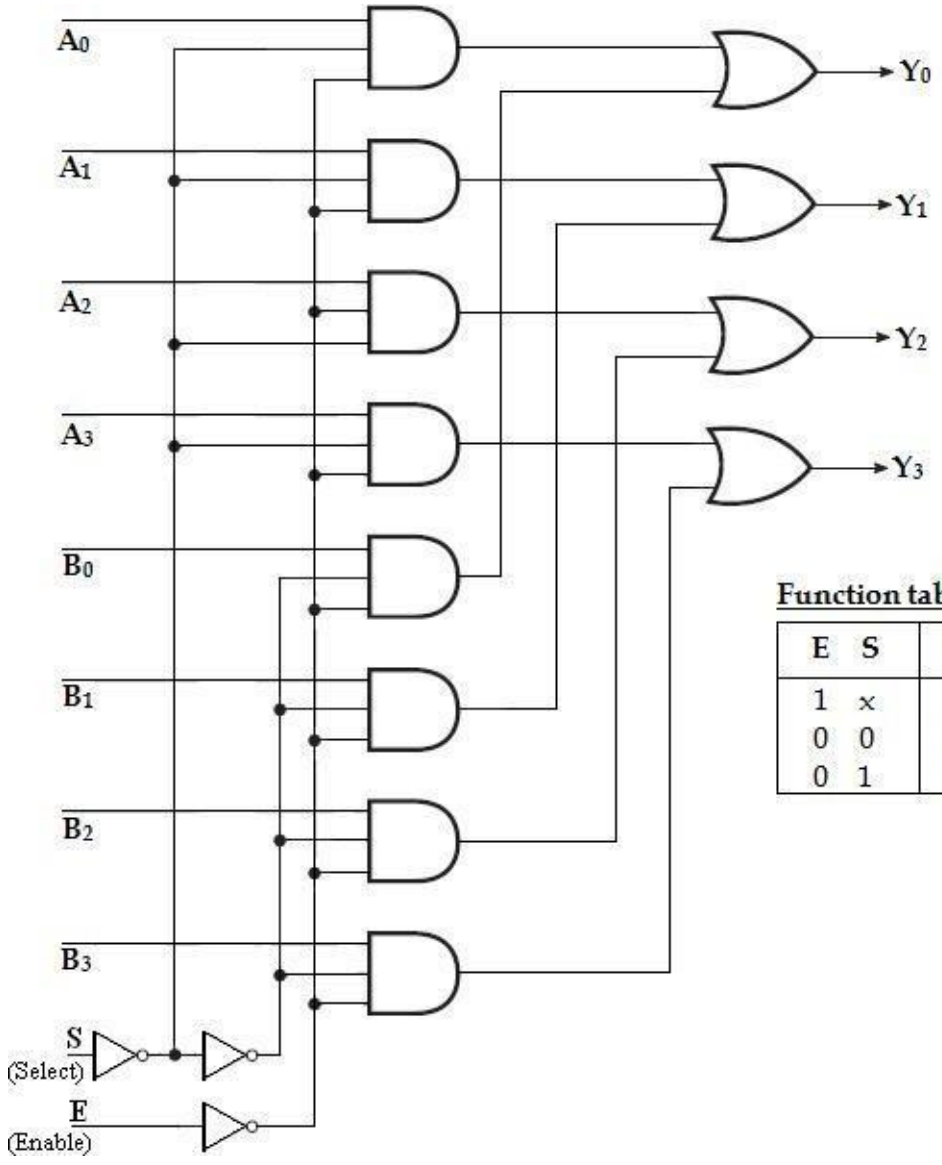
To demonstrate the circuit operation, consider the case when S_1S_0= 10. The AND gate associated with input I_2 has two of its inputs equal to 1 and the third input connected to I_2. The other three AND gates have atleast one input equal to 0, which makes their outputs equal to 0. The OR output is now equal to the value of I_2, providinga path from the selected input to the output.

The data output is equal to I_0 only if S_1= 0 and S_0= 0; Y= $I_0S_1'S_0'$.
The data output is equal to I_1 only if S_1= 0 and S_0= 1; Y= $I_1S_1'S_0$.
The data output is equal to I_2 only if S_1= 1 and S_0= 0; Y= $I_2S_1S_0'$.
The data output is equal to I_3 only if S_1= 1 and S_0= 1; Y= $I_3S_1S_0$.
When these terms are ORed, the total expression for the data output is,

$Y= I_0S_1'S_0'+ I_1S_1'S_0 +I_2S_1S_0'+ I_3S_1S_0$.

As in decoder, multiplexers may have an enable input to control the operation of the unit. When the enable input is in the inactive state, the outputs are disabled, and when it is in the active state, the circuit functions as a normal multiplexer.

Quadruple 2-to-1 Line Multiplexer:



Function table:

E	S	Output Y
1	x	All 0's
0	0	Select A
0	1	Select B

This circuit has four multiplexers, each capable of selecting one of two input lines. Output Y_0 can be selected to come from either A0 or B0. Similarly, output Y1 may have the value of A1 or B1, and so on. Input selection line, S selects one of the lines in each of the four multiplexers. The enable input E must be active for normal operation.

Although the circuit contains four 2-to-1-Line multiplexers, it is viewed as a circuit that selects one of two 4-bit sets of data lines. The unit is enabled when E= 0. Then if S= 0, the four A inputs have a path to the four outputs. On the other hand, ifS=1, the fourB inputs are applied to the outputs. The outputs have all 0's when E= 1, regardless of the value of S.

Application:

The multiplexer is a very useful MSI function and has various ranges of applications in data communication. Signal routing and data communication are the important applications of a multiplexer. It is used for connecting two or more sources to guide to a single destination among computer units and it is useful for constructing a common bus system. One of the general properties of a multiplexer is that Boolean functions can be implemented by this device.

Implementation of Boolean Function using MUX:

Any Boolean or logical expression can be easily implemented using a multiplexer. If a Boolean expression has (n+1) variables, then _n' of these variables can be connected to the select lines of the multiplexer. The remaining single variable along with constants 1 and 0 is used as the input of the multiplexer. For example, if C is the single variable, then the inputs of the multiplexers are C, C', 1 and 0. By this method any logical expression can be implemented.

In general, a Boolean expression of (n+1) variables can be implemented using a multiplexer with 2_n inputs.

1. **Implement the following boolean function using 4: 1 multiplexer,**
   $F(A, B, C) = \sum m(1, 3, 5, 6)$.

Solution:

Variables, n= 3 (A, B, C)

Select lines= n-1 = 2 (S_1, S_0)

2_{n-1} to MUX i.e., 2_2 to 1 = 4 to 1 MUX

Input lines= $2^{n-1} = 2^2 = 4$ (D_0, D_1, D_2, D_3)

Implementation table:

Apply variables A and B to the select lines. The procedures for implementing the function are:

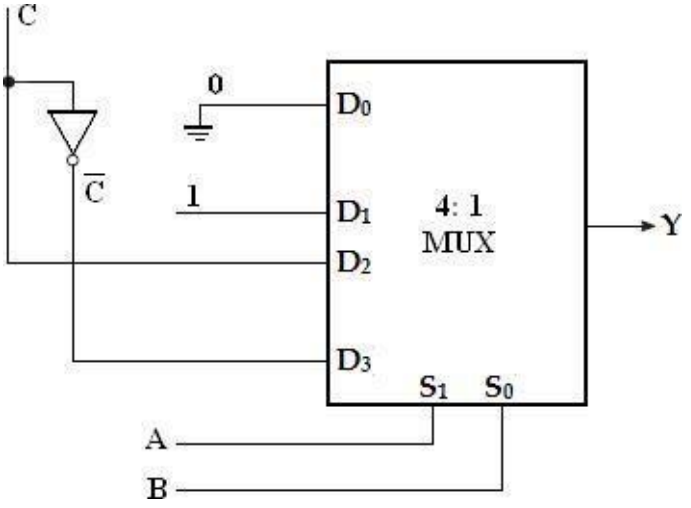
i. List the input of the multiplexer
ii. List under them all the minterms in two rows as shown below.

The first half of the minterms is associated with A‘ and the second half with A. The given function is implemented by circling the minterms of the function and applying the following rules to find the values for the inputs of the multiplexer.

1. If both the minterms in the column are not circled, apply 0 to the corresponding input.
2. If both the minterms in the column are circled, apply 1 to the corresponding input.
3. If the bottom minterm is circled and the top is not circled, apply C to the input.
4. If the top minterm is circled and the bottom is not circled, apply C‘ to the input.

	D_0	D_1	D_2	D_3
$\overline{C}$	0	(1)	2	(3)
C	4	(5)	(6)	7
	0	**1**	**C**	$\overline{C}$

Multiplexer Implementation:



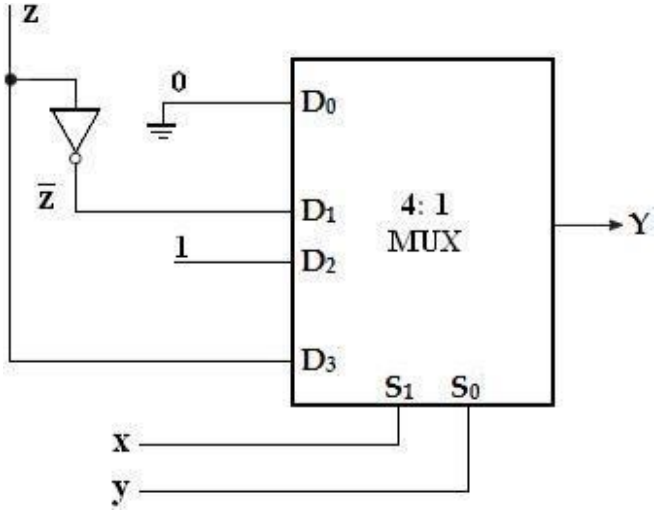
2. F (x, y, z) = ∑m (1, 2, 6, 7)

Solution:

Implementation table:

	D_0	D_1	D_2	D_3
$\bar{z}$	0	(1)	(2)	3
z	4	5	(6)	(7)
	0	$\bar{z}$	**1**	**z**

Multiplexer Implementation:



3. **F (A, B, C) = ∑m (1, 2, 4, 5)**

Solution:

Variables, n= 3 (A, B, C)

Select lines= n-1 = 2 (**S_1, S_0**)

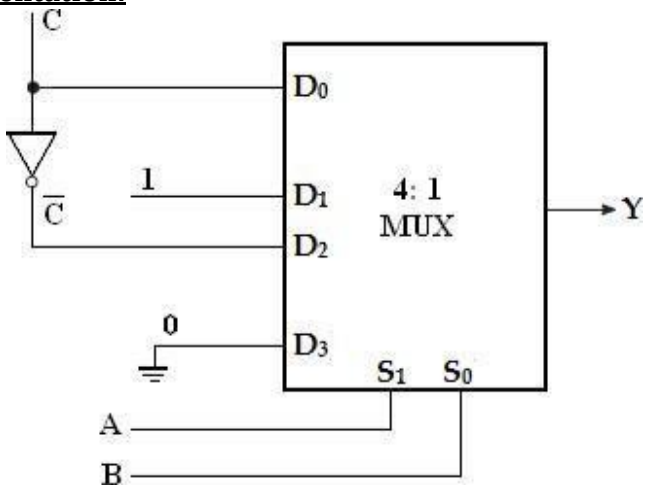
2_{n-1} to MUX i.e., 2_2 to 1 = 4 to 1 MUX

Input lines= $2^{n-1} = 2^2 = 4$ (**D_0, D_1, D_2, D_3**)

Implementation table:

	D_0	D_1	D_2	D_3
$\bar{C}$	0	(1)	(2)	3
C	(4)	(5)	6	7
	C	**1**	$\bar{C}$	**0**

Multiplexer Implementation:



4. F(P, Q, R, S)= ∑m (0, 1, 3, 4, 8, 9, 15)

Solution:

Variables, n= 4 (P, Q, R, S)

Select lines= n-1 = 3 (**S_2, S_1, S_0**)

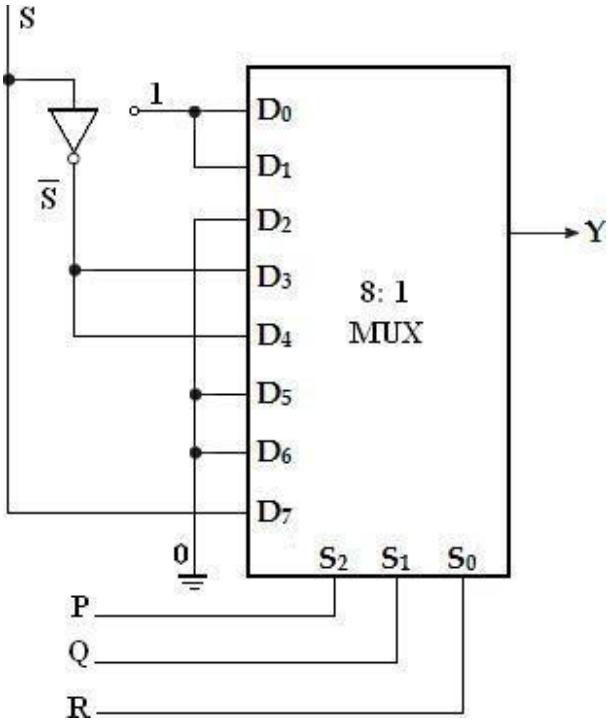
2_{n-1} to MUX i.e., 2_3 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{S}$	(0)	(1)	2	(3)	(4)	5	6	7
S	(8)	(9)	10	11	12	13	14	(15)
	1	**1**	**0**	$\overline{S}$	$\overline{S}$	**0**	**0**	**S**

Multiplexer Implementation:



5. **Implement the Boolean function using 8: 1 and also using 4:1 multiplexer**
   F (A, B, C, D) = ∑m (0, 1, 2, 4, 6, 9, 12, 14)

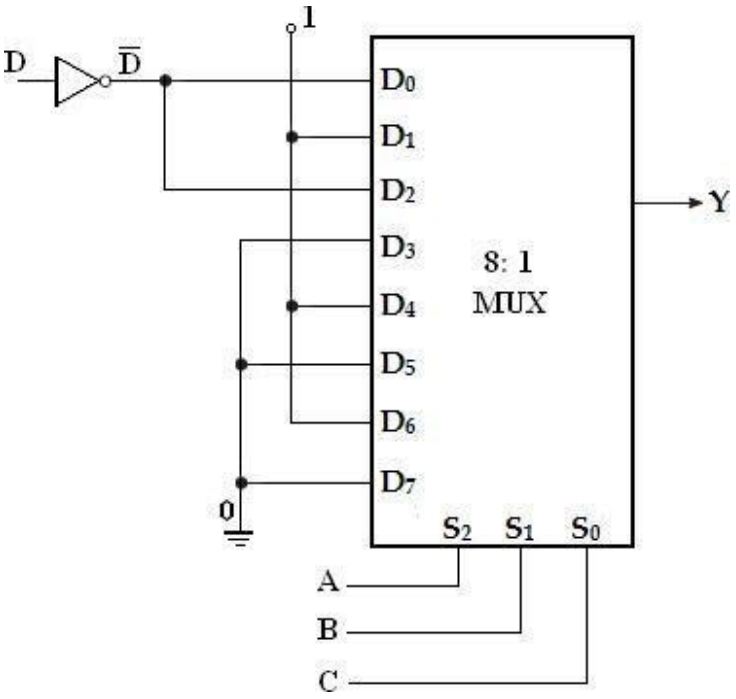
Solution:

Variables, n= 4 (A, B, C, D)
Select lines= n-1 = 3 (**S_2, S_1, S_0**)
2^{n-1} to MUX i.e., 2_3 to 1 = 8 to 1 MUX
Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

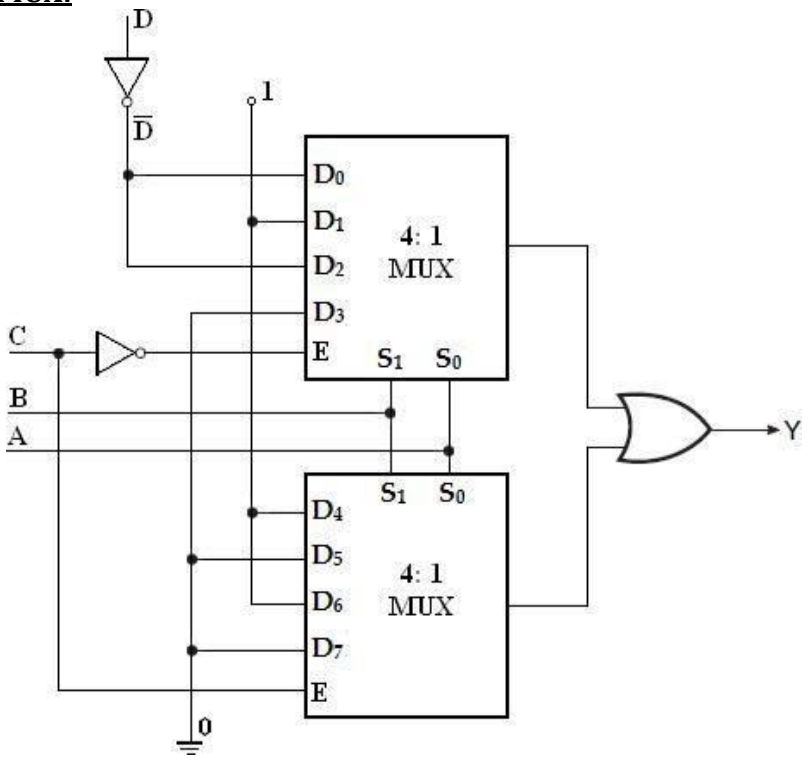
Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	(0)	(1)	(2)	3	(4)	5	(6)	7
D	8	(9)	10	11	(12)	13	(14)	15
	$\overline{D}$	**1**	$\overline{D}$	**0**	**1**	**0**	**1**	**0**

Multiplexer Implementation (Using 8: 1 MUX):



Using 4: 1 MUX:



6. F (A, B, C, D) = ∑m (1, 3, 4, 11, 12, 13, 14, 15)

Solution:

Variables, n= 4 (A, B, C, D)

Select lines= n-1 = 3 (**S_2, S_1, S_0**)

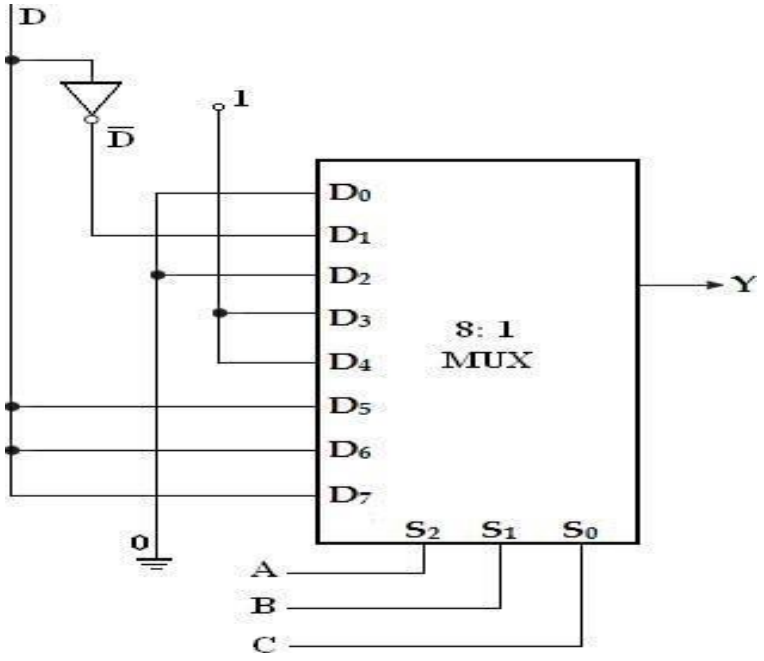
2_{n-1} to MUX i.e., 2_3 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	0	(1)	2	(3)	(4)	5	6	7
D	8	9	10	(11)	(12)	(13)	(14)	(15)
	0	$\overline{D}$	**0**	**1**	**1**	D	D	D

Multiplexer Implementation:



7. Implement the Boolean function using 8: 1 multiplexer.

   F (A, B, C, D) = A'BD' + ACD + B'CD + A'C'D.

Solution:

Convert into standard SOP form,

= A‘BD‘ (C‘+C) + ACD (B‘+B) + B‘CD (A‘+A) + A‘C‘D (B‘+B)

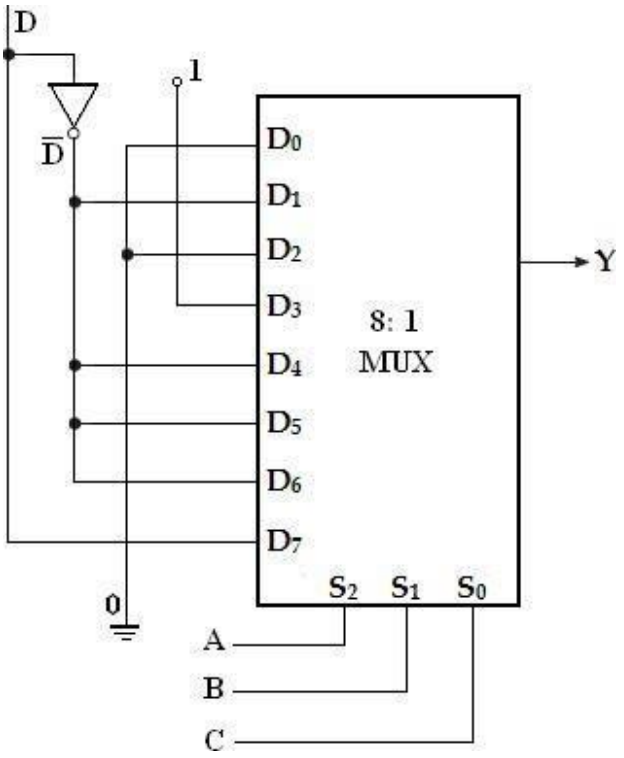
= A‘BC‘D‘ + A‘BCD‘+ AB‘CD + ABCD +A‘B‘CD + AB‘CD +A‘B‘C‘D+ A‘BC‘D

= A‘BC‘D‘ + A‘BCD‘+ AB‘CD + ABCD +A‘B‘CD +A‘B‘C‘D+ A‘BC‘D
= m4+ m6+ m11+ m15+ m3+ m1+ m5
= ∑m (1, 3, 4, 5, 6, 11, 15)

Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	0	(1)	2	(3)	(4)	(5)	(6)	7
D	8	9	10	(11)	12	13	14	(15)
	0	$\overline{D}$	**0**	**1**	$\overline{D}$	$\overline{D}$	$\overline{D}$	D

Multiplexer Implementation:



8. Implement the Boolean function using 8: 1 multiplexer.
F (A, B, C, D) = AB’D + A’C’D + B’CD’ + AC’D.

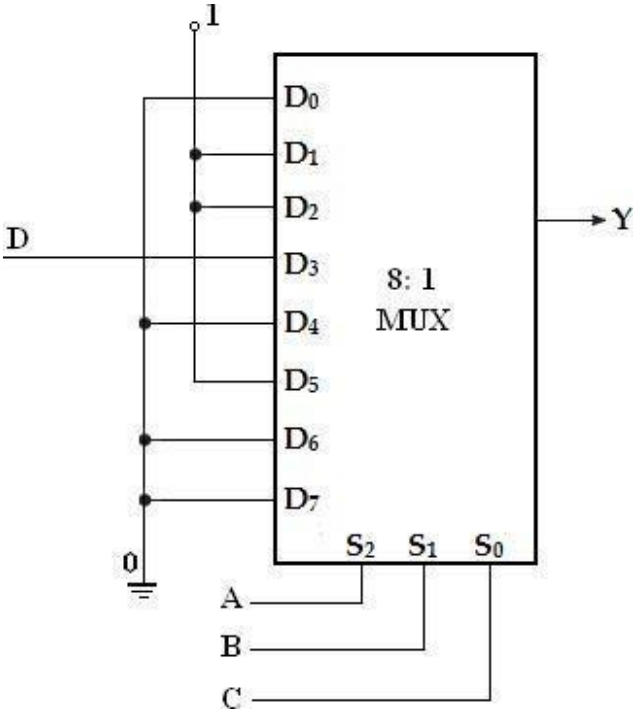
Solution:
Convert into standard SOP form,
= AB‘D (C‘+C) + A‘C‘D (B‘+B) + B‘CD‘ (A‘+A) + AC‘D (B‘+B)

= AB‘C‘D + AB‘CD+ A‘B‘C‘D + A‘BC‘D +A‘B‘CD‘ + AB‘CD‘ +AB‘C‘D+ ABC‘D
= AB‘C‘D + AB‘CD+ A‘B‘C‘D + A‘BC‘D +A‘B‘CD‘ + AB‘CD‘+ ABC‘D
= m9+ m11+ m1+ m5+ m2+ m10+ m13
= ∑m (1, 2, 5, 9, 10, 11, 13).

Implementation Table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	0	(1)	(2)	3	4	(5)	6	7
D	8	(9)	(10)	(11)	12	(13)	14	15
	0	**1**	**1**	D	**0**	**1**	**0**	**0**

Multiplexer Implementation:



9. Implement the Boolean function using 8: 1 and also using 4:1 multiplexer
F (w, x, y, z) = ∑m (1, 2, 3, 6, 7, 8, 11, 12, 14)

Solution:

Variables, n= 4 (w, x, y, z)
Select lines= n-1 = 3 (**S_2, S_1, S_0**)

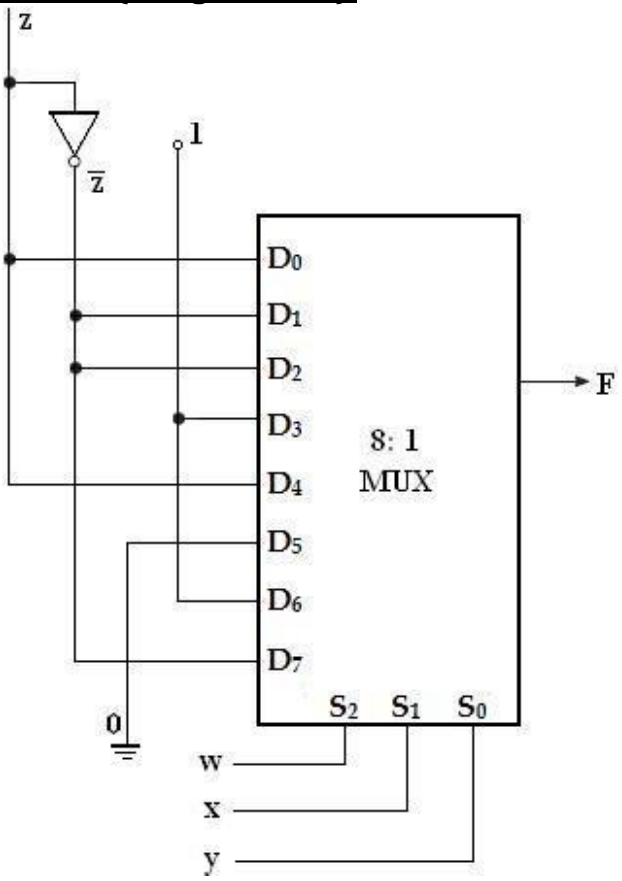
2_{n-1} to MUX i.e., 2_3 to 1 = 8 to 1 MUX

Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

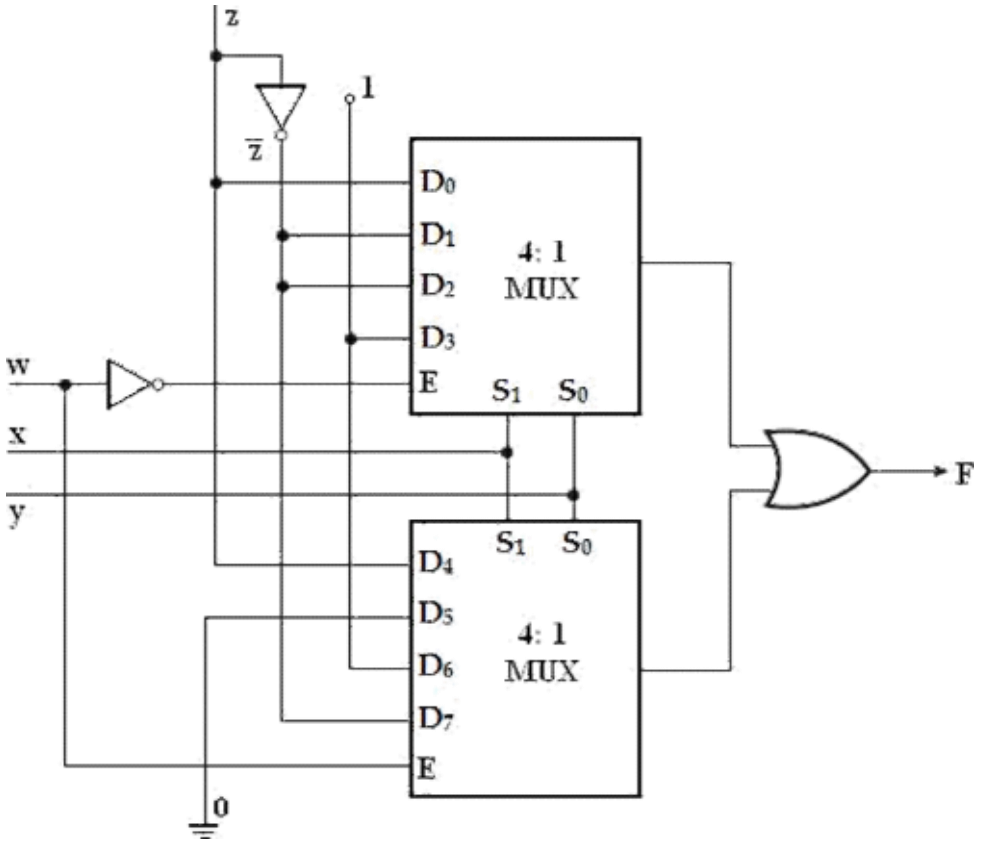
Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\bar{z}$	0	(1)	(2)	(3)	4	5	(6)	(7)
z	(8)	9	10	(11)	(12)	13	(14)	15
	z	$\bar{z}$	$\bar{z}$	**1**	z	**0**	**1**	$\bar{z}$

Multiplexer Implementation (Using 8:1 MUX):



(Using 4:1 MUX):



10. Implement the Boolean function using 8: 1 multiplexer
F (A, B, C, D) = ∏m (0, 3, 5, 8, 9, 10, 12, 14)

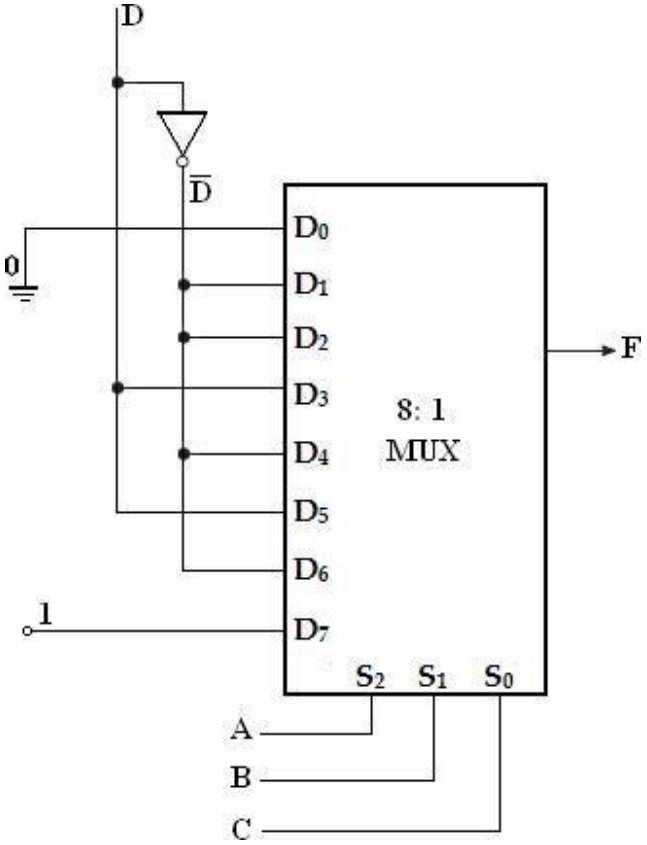
Solution:

Variables, n= 4 (A, B, C, D)
Select lines= n-1 = 3 (**S_2, S_1, S_0**)
2^{n-1} to MUX i.e., 2_3 to 1 = 8 to 1 MUX
Input lines= $2^{n-1} = 2^3 = 8$ (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

Implementation table:

	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	0	(1)	(2)	3	(4)	5	(6)	(7)
D	8	9	10	(11)	12	(13)	14	(15)
	0	$\overline{D}$	$\overline{D}$	D	$\overline{D}$	D	$\overline{D}$	**1**

Multiplexer Implementation:



11. Implement the Boolean function using 8: 1 multiplexer
F (A, B, C, D) = ∑m (0, 2, 6, 10, 11, 12, 13) + d (3, 8, 14)

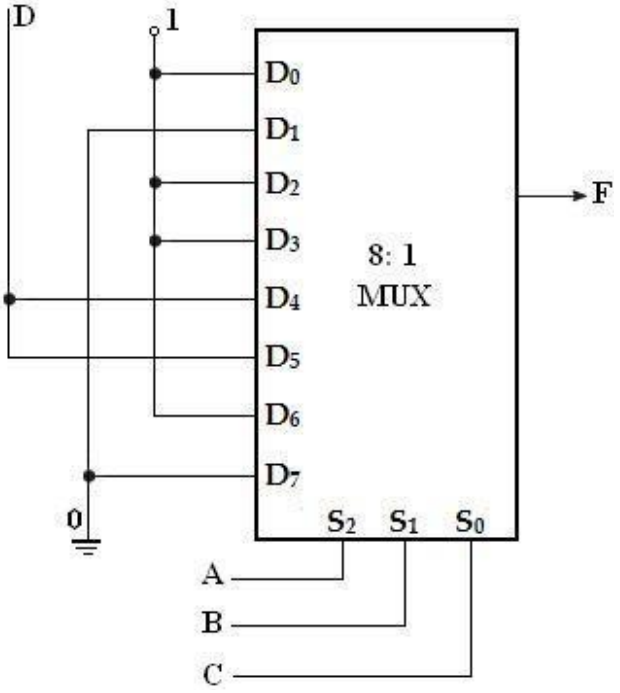
Solution:

Variables, n= 4 (A, B, C, D)
Select lines= n-1 = 3 (**S_2, S_1, S_0**)
2_{n-1} to MUX i.e., 2_3 to 1 = 8 to 1 MUX
Input lines= $2^{n-1} = 2^3$ = 8 (**D_0, D_1, D_2, D_3, D_4, D_5, D_6, D_7**)

Implementation Table:

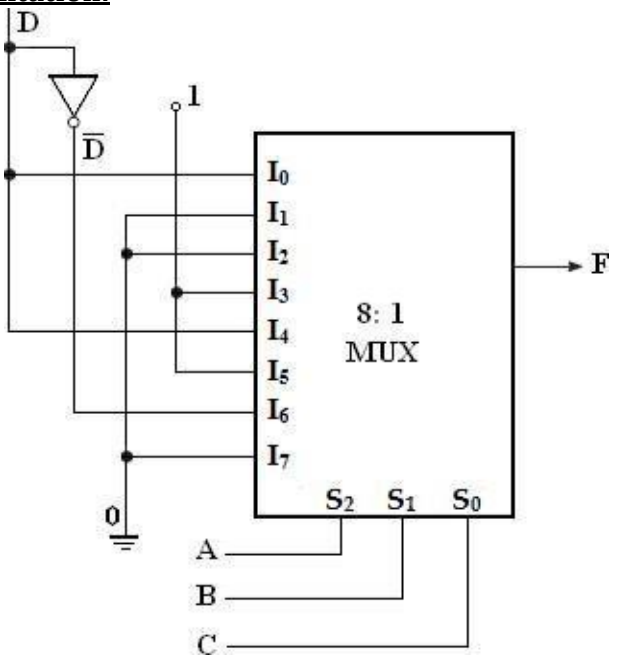
	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
$\overline{D}$	(0)	1	(2)	(3)	4	5	(6)	7
D	(8)	9	(10)	(11)	(12)	(13)	(14)	15
	1	**0**	**1**	**1**	D	D	**1**	**0**

Multiplexer Implementation:



12. An 8×1 multiplexer has inputs A, B and C connected to the selection inputs S_2, S_1, and S_0 respectively. The data inputs I_0 to I_7 are as follows
$I_1=I_2=I_7=0$; $I_3=I_5=1$; $I_0=I_4=D$ and **$I_6=D'$.**
Determine the Boolean function that the multiplexer implements.

Multiplexer Implementation:



Implementation table:

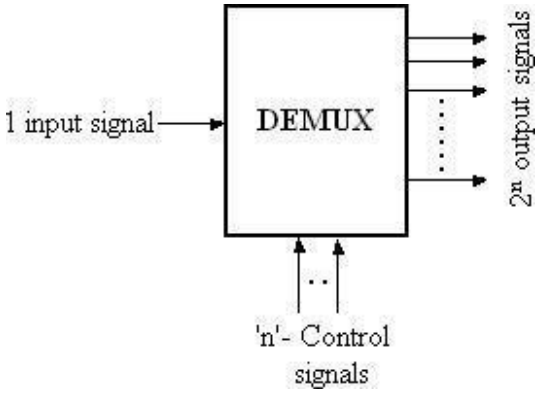
	I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7
$\overline{D}$	0	1	2	(3)	4	(5)	(6)	7
D	(8)	9	10	(11)	(12)	(13)	14	15
	D	0	0	1	D	1	$\overline{D}$	0

F (A, B, C, D) = ∑m (3, 5, 6, 8, 11, 12, 13).

DEMULTIPLEXER:

Demultiplex means one into many. Demultiplexing is the process of taking information from one input and transmitting the same over one of several outputs.

A demultiplexer is a combinational logic circuit that receives information on a single input and transmits the same information over one of several (2^n) output lines.

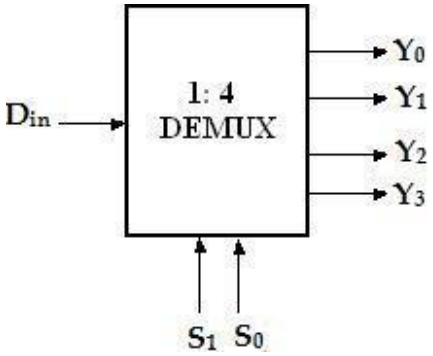


Block diagram of demultiplexer

The block diagram of a demultiplexer which is opposite to a multiplexer in its operation is shown above. The circuit has one input signal, ‗n' select signals and 2^n output signals. The select inputs determine to which output the data input will be connected. As the serial data is changed to parallel data, i.e., the input caused to appear on one of the n output lines, the demultiplexer is also called a ―***data distributer***‖ or a ―***serial-to-parallel converter***‖ .

1-to-4 Demultiplexer:

A 1-to-4 demultiplexer has a single input, D_{in}, four outputs (Y_0 **to** Y_3) and two select inputs (S_1 **and** S_0).



Logic Symbol

The input variable D_{in} has a path to all four outputs, but the input information is directed to only one of the output lines. The truth table of the 1-to-4 demultiplexer is shown below.

Enable	S_1	S_0	D_{in}	Y_0	Y_1	Y_2	Y_3
0	x	x	x	0	0	0	0
1	0	0	0	0	0	0	0
1	0	0	1	1	0	0	0
1	0	1	0	0	0	0	0
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	0
1	1	0	1	0	0	1	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	1

Truth table of 1-to-4 demultiplexer

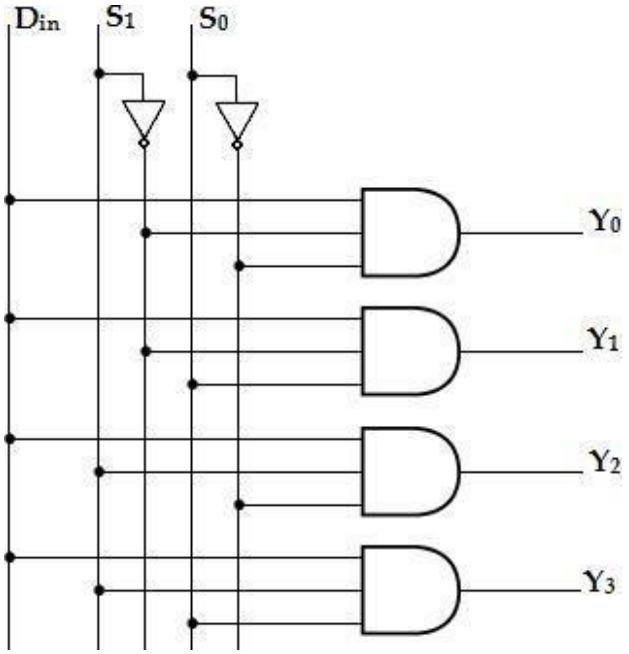
From the truth table, it is clear that the data input, D_{in} is connected to the output Y_0, when S_1= 0 and S_0= 0 and the data input is connected to output Y_1 when S_1= 0 and S_0= 1. Similarly, the data input is connected to output Y_2 and Y_3 when S_1= 1 and S_0= 0 and when S_1= 1 and S_0= 1, respectively. Also, from the truth table, the expression for outputs can be written as follows,

$Y_0 = S_1'S_0'D_{in}$

$Y_1 = S_1'S_0D_{in}$

$Y_2 = S_1S_0'D_{in}$

$Y_3 = S_1S_0D_{in}$



Logic diagram of 1-to-4 demultiplexer

Now, using the above expressions, a 1-to-4 demultiplexer can be implemented using four 3-input AND gates and two NOT gates. Here, the input data line D_{in}, is connected to all the AND gates. The two select lines S_1, S_0 enable only one gate at a time

and the data that appears on the input line passes through the selected gate to the associated output line.

1-to-8 Demultiplexer:

A 1-to-8 demultiplexer has a single input, **D_{in}**, eight outputs (**Y_0 to Y_7**) and three select inputs (**S_2, S_1 and S_0**). It distributes one input line to eight output lines based on the select inputs. The truth table of 1-to-8 demultiplexer is shown below.

D_{in}	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	1
1	0	0	1	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0

1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

Truth table of 1-to-8 demultiplexer

From the above truth table, it is clear that the data input is connected with one of the eight outputs based on the select inputs. Now from this truth table, the expression for eight outputs can be written as follows:

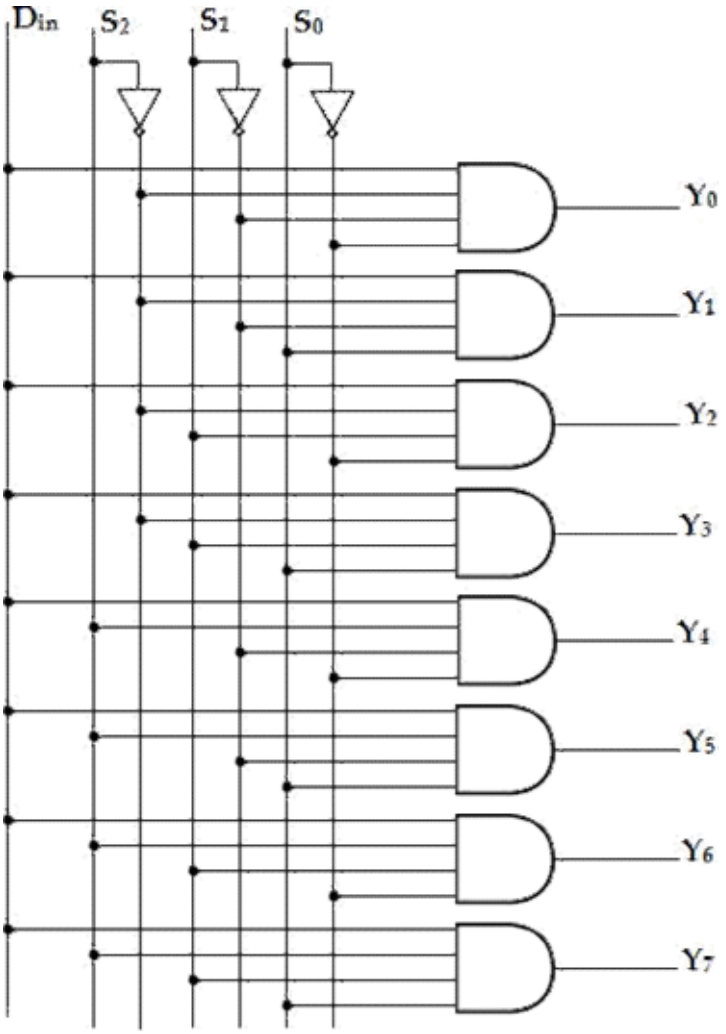
$Y_0 = S_2'S_1'S_0'D_{in}$ $Y_4 = S_2 S_1'S_0'D_{in}$

$Y_1 = S_2'S_1'S_0D_{in}$ $Y_5 = S_2 S_1'S_0D_{in}$

$Y_2 = S_2'S_1S_0'D_{in}$ $Y_6 = S_2 S_1S_0'D_{in}$

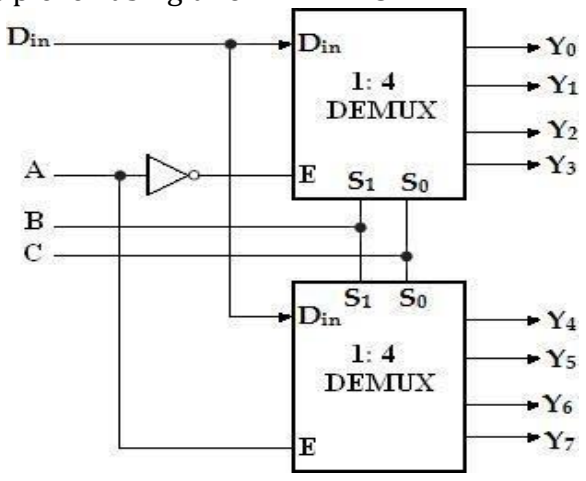
$Y_3 = S_2'S_1S_0D_{in}$ $Y_7 = S_2S_1S_0D_{in}$

Now using the above expressions, the logic diagram of a 1-to-8 demultiplexer can be drawn as shown below. Here, the single data line, D_{in} is connected to all the eight AND gates, but only one of the eight AND gates will be enabled by the select input lines. For example, if $S_2S_1S_0 = 000$, then only AND gate-0 will be enabled and thereby the data input, D_{in} will appear at Y_0. Similarly, the different combinations of the select inputs, the input D_{in} will appear at the respective output.



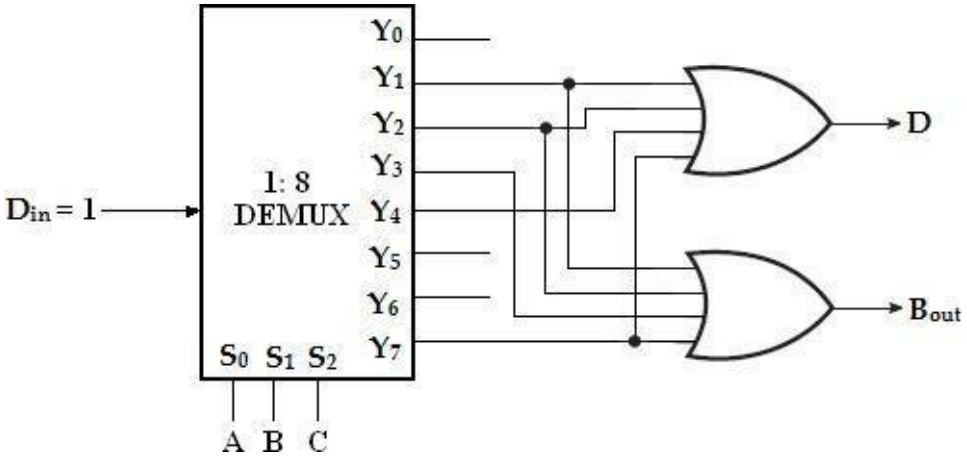
Logic diagram of 1-to-8 demultiplexer

1. Design 1:8 demultiplexer using two 1:4 DEMUX.



2. Implement full subtractor using demultiplexer.

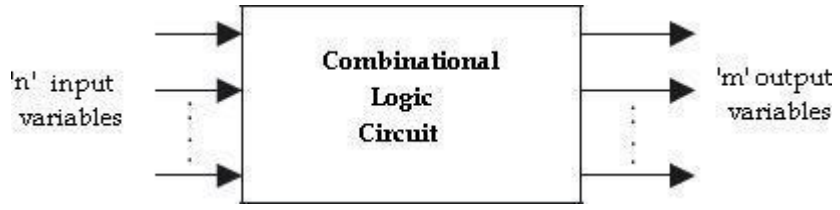
Inputs			Outputs	
A	**B**	**B_{in}**	**Difference(D)**	**Borrow(B_{out})**
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1



UNIT III SYNCHRONOUS SEQUENTIAL CIRCUITS

INTRODUCTION

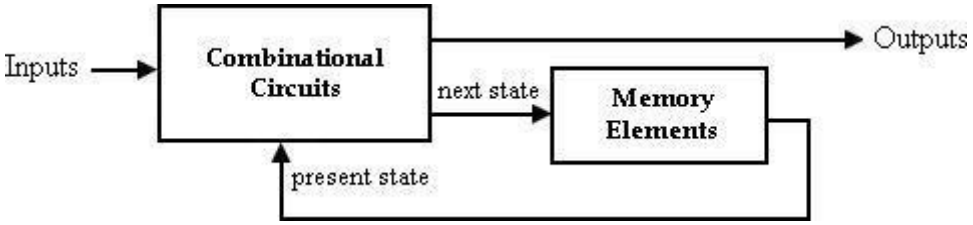
In ***combinational logic circuits***, the outputs at any instant of time depend only on the input signals present at that time. For a change in input, the output occurs immediately.



Combinational Circuit- Block Diagram

In ***sequential logic circuits***, it consists of combinational circuits to which storage elements are connected to form a feedback path. The storage elements are devices capable of storing binary information either 1 or 0.

The information stored in the memory elements at any given time defines the present state of the sequential circuit. The present state and the external circuit determine the output and the next state of sequential circuits.



Sequential Circuit- Block Diagram

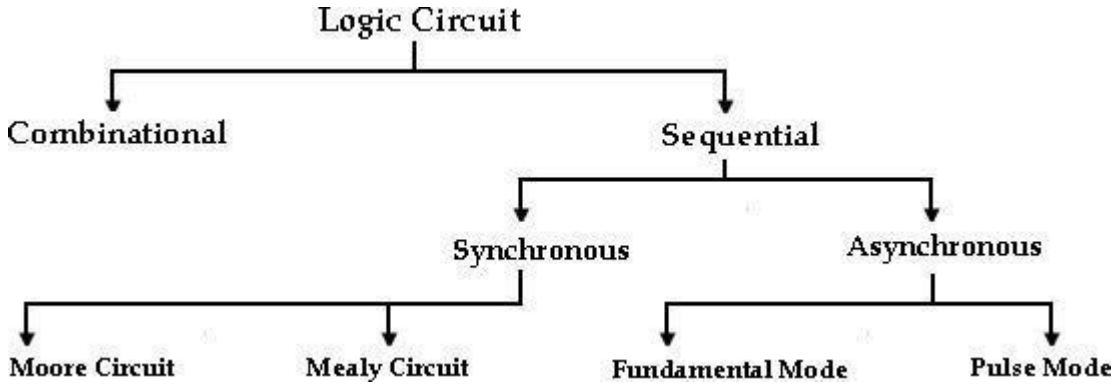
Thus in sequential circuits, the output variables depend not only on the present input variables but also on the past history of input variables.

The rotary channel selected knob on an old-fashioned TV is like a combinational. Its output selects a channel based only on its current input – the position of the knob. The channel-up and channel-down push buttons on a TV is like a sequential circuit. The channel selection depends on the past sequence of up/down pushes.

The comparison between combinational and sequential circuits is given in table below.

S.No	Combinational logic	Sequential logic
1	The output variable, at all times depends on the combination of input variables.	The output variable depends not only on the present input but also depend upon the past history of inputs.
2	Memory unit is not required	Memory unit is required to store the past history of input variables.
3	Faster in speed	Slower than combinational circuits.
4	Easy to design	Comparatively harder to design.
5	Eg. Parallel adder	Eg. Serial adder

3.2 Classification of Logic Circuits



The sequential circuits can be classified depending on the timing of their signals:

- Synchronous sequential circuits
- Asynchronous sequential circuits.

In synchronous sequential circuits, signals can affect the memory elements only at discrete instants of time. In asynchronous sequential circuits change in input signals can affect memory element at any instant of time. The memory elements used in both circuits are Flip-Flops, which are capable of storing 1-bit information.

S.No	Synchronous sequential circuits	Asynchronous sequential circuits
1	Memory elements are clocked Flip-Flops	Memory elements are either unclocked Flip-Flops or time delay elements.
2	The change in input signals can affect memory element upon activation of clock signal.	The change in input signals can affect memory element at any instant of time.
3	The maximum operating speed of clock depends on time delays involved.	Because of the absence of clock, it can operate faster than synchronous circuits.
4	Easier to design	More difficult to design

3.3 LATCHES:

Latches and Flip-Flops are the basic building blocks of the most sequential circuits. Latches are used for a sequential device that checks all of its inputs continuously and changes its outputs accordingly at any time independent of clocking signal. Enable signal is provided with the latch. When enable signal is activeoutput changes occur as the input changes. But when enable signal is not activated input changes do not affect the output.

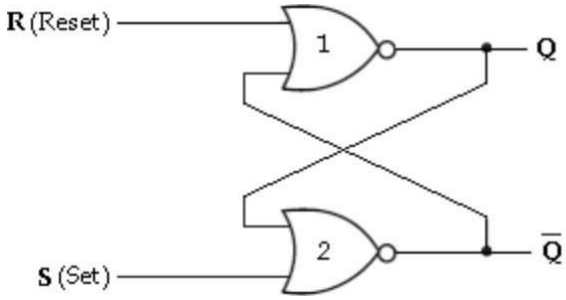
Flip-Flop is used for a sequential device that normally samples its inputs and changes its outputs only at times determined by clocking signal.

3.3.1 SR Latch:

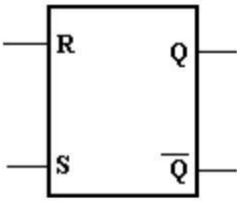
The simplest type of latch is the set-reset (SR) latch. It can be constructed from either two NOR gates or two NAND gates.

SR latch using NOR gates:

The two NOR gates are cross-coupled so that the output of NOR gate 1 is connected to one of the inputs of NOR gate 2 and vice versa. The latch has two outputs Q and Q' and two inputs, set and reset.



SR latch using NOR gates



Logic Symbol

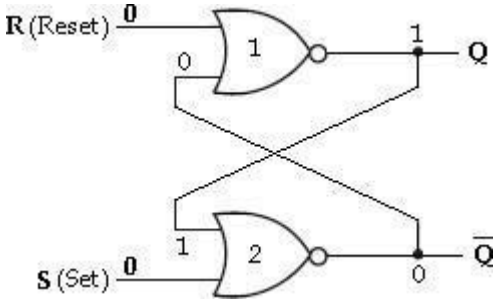
Before going to analyse the SR latch, we recall that a logic 1 at any input of a NOR gate forces its output to a logic 0. Let us understand the operation of this circuit for various input/ output possibilities.

Case 1: S= 0 and R= 0

Initially, Q= 1 and Q'= 0

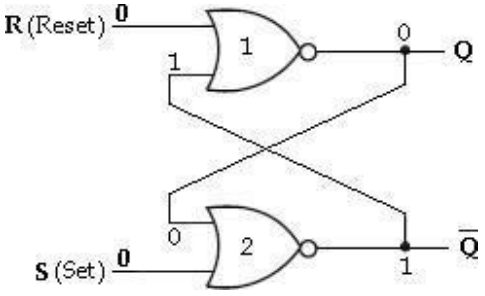
Let us assume that initially Q=1 and Q'=0. With Q'=0, both inputs to NORgate 1 are at logic 0. So, its output, Q is at logic 1. With Q=1, one input of NOR gate 2 is at logic

1. Hence its output, Q' is at logic 0. This shows that when S and R both are low, the output does not change.



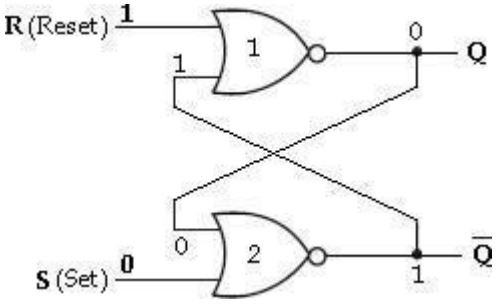
Initially, Q= 0 and Q'= 1

With Q'=1, one input of NOR gate 1 is at logic 1, hence its output, Q is at logic 0. With Q=0, both inputs to NOR gate 2 are at logic 0. So, its output Q' is at logic 1. In this case also there is no change in the output state.



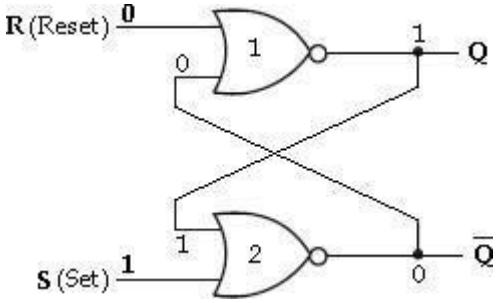
Case 2: S= 0 and R= 1

In this case, R input of the NOR gate 1 is at logic 1, hence its output, Q is at logic 0. Both inputs to NOR gate 2 are now at logic 0. So that its output, Q' is at logic 1.



Case 3: S= 1 and R= 0

In this case, S input of the NOR gate 2 is at logic 1, hence its output, Q is at logic 0. Both inputs to NOR gate 1 are now at logic 0. So that its output, Q is at logic 1.



Case 4: S= 1 and R= 1

When R and S both are at logic 1, they force the outputs of both NOR gates to the low state, i.e., (Q=0 and Q'=0). So, we call this an indeterminate or prohibited state, and represent this condition in the truth table as an asterisk (*). This conditionalso violates the basic definition of a latch that requires Q to be complement of Q'. Thus in normal operation this condition must be avoided by making sure that 1's arenot applied to both the inputs simultaneously.

We can summarize the operation of SR latch as follows:

- When S= 0 and R= 0, the output, Q_{n+1} remains in its present state, Q_n.
- When S= 0 and R= 1, the latch is reset to 0.
- When S= 1 and R= 0, the latch is set to 1.
- When S= 1 and R= 1, the output of both gates will produce 0.

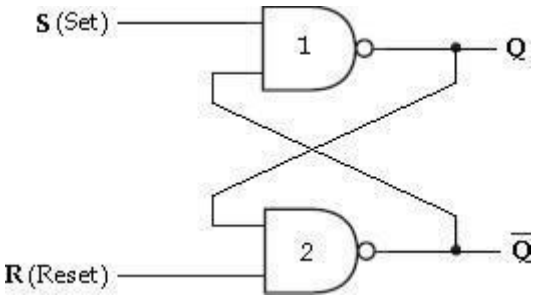
  i.e., $\mathbf{Q_{n+1} = Q_{n+1}' = 0}$.

The truth table of NOR based SR latch is shown below.

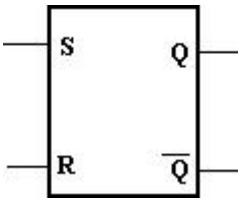
S	R	Q_n	Q_{n+1}	State
0	0	0	0	No Change
0	0	1	1	(NC)
0	1	0	0	Reset
0	1	1	0	
1	0	0	1	Set
1	0	1	1	
1	1	0	x	Indeterminate
1	1	1	x	*

SR latch using NAND gates:

The SR latch can also be implemented using NAND gates. The inputs of this Latch are S and R. To understand how this circuit functions, recall that a low on any input to a NAND gate forces its output high.



SR latch using NAND gates



Logic Symbol

We can summarize the operation of SR latch as follows:

- When S= 0 and R= 0, the output of both gates will produce 0.

  i.e., $\mathbf{Q_{n+1} = Q_{n+1}' = 1.}$
- When S= 0 and R= 1, the latch is reset to 0.
- When S= 1 and R= 0, the latch is set to 1.
- When S= 1 and R= 1, the output, Q_{n+1} remains in its present state, Q_n.

The truth table of NAND based SR latch is shown below.

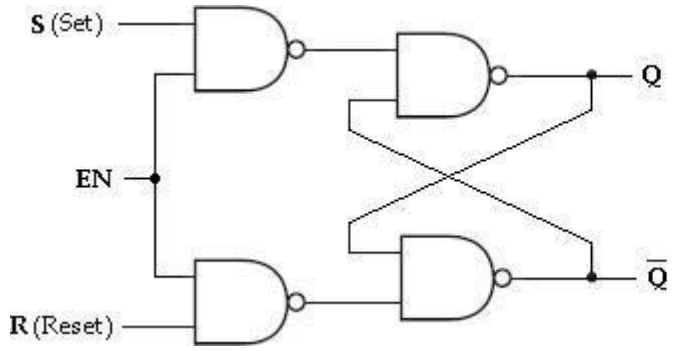
S	R	Q_n	Q_{n+1}	State
0	0	0	x	Indeterminate
0	0	1	x	*
0	1	0	1	Set
0	1	1	1	
1	0	0	0	Reset
1	0	1	0	
1	1	0	0	No Change
1	1	1	1	(NC)

Gated SR Latch:

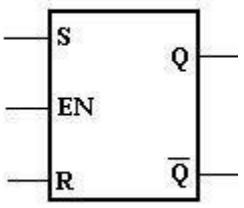
In the SR latch, the output changes occur immediately after the input changes i.e, the latch is sensitive to its S and R inputs all the time.

A latch that is sensitive to the inputs only when an enable input is active. Such a latch with enable input is known as gated SR latch.

- The circuit behaves like SR latch when EN= 1. It retains its previous state when EN= 0



SR Latch with enable input using NAND gates



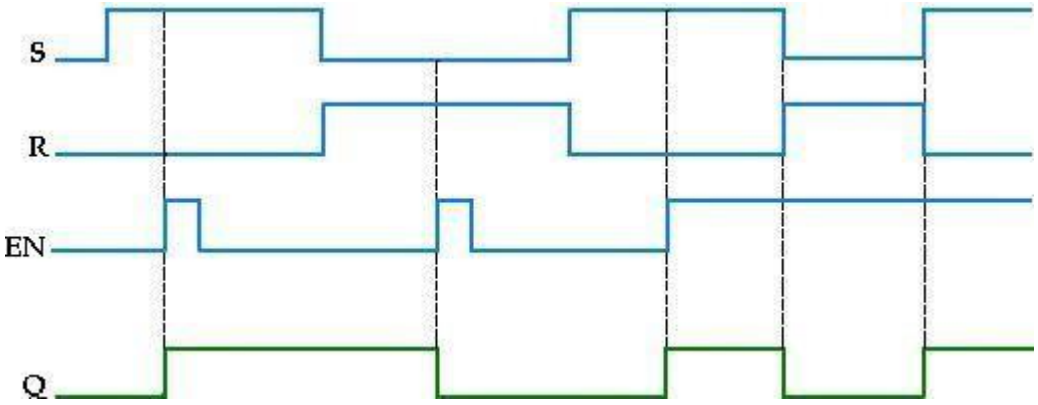
Logic Symbol

The truth table of gated SR latch is show below.

EN	S	R	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	

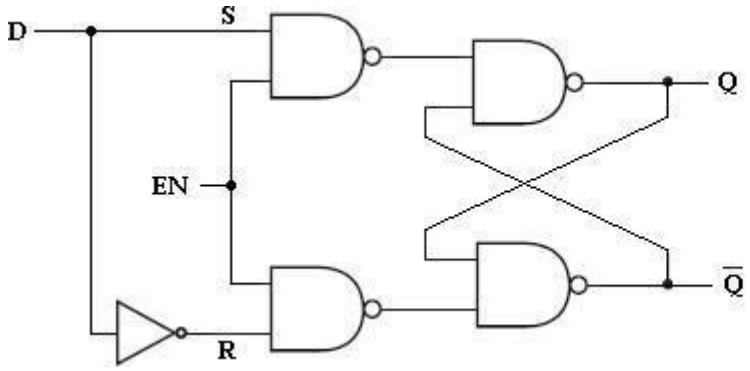
1 1	1 1	0 0	0 1	1 1	Set
1 1	1 1	1 1	0 1	x x	Indeterminate *
0 0	x x	x x	0 1	0 1	No Change (NC)

When S is HIGH and R is LOW, a HIGH on the EN input sets the latch. When S is LOW and R is HIGH, a HIGH on the EN input resets the latch.

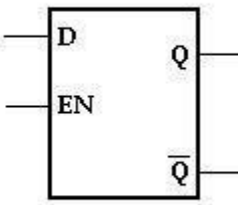


3.3.2 D Latch

In SR latch, when both inputs are same (00 or 11), the output either does not change or it is invalid. In many practical applications, these input conditions are not required. These input conditions can be avoided by making them complement of each other. This modified SR latch is known as **D latch.**



D Latch



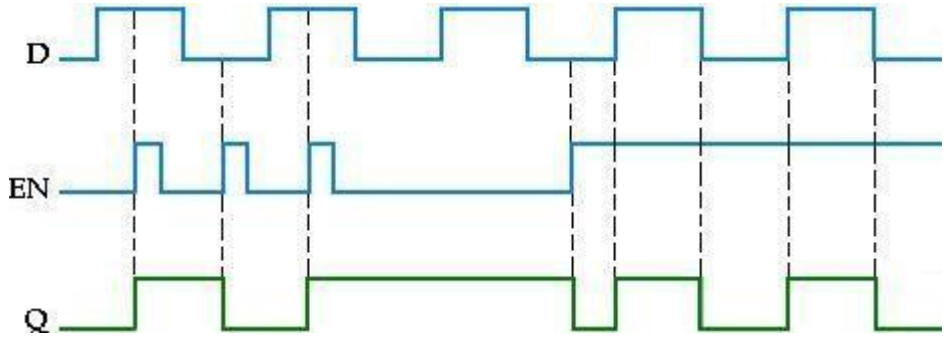
Logic Symbol

As shown in the figure, D input goes directly to the S input, and its complement is applied to the R input. Therefore, only two input conditions exists, either S=0 and R=1 or S=1 and R=0. The truth table for D latch is shown below.

EN	D	Q_n	Q_{n+1}	State
1	0	x	0	Reset
1	1	x	1	Set
0	x	x	Q_n	No Change (NC)

As shown in the truth table, the Q output follows the D input. For this reason, D latch is called ***transparent latch***.

When D is HIGH and EN is HIGH. Q goes HIGH. When D is LOW and EN is HIGH, Q goes LOW. When EN is LOW, the state of the latch is not affected by the D input.

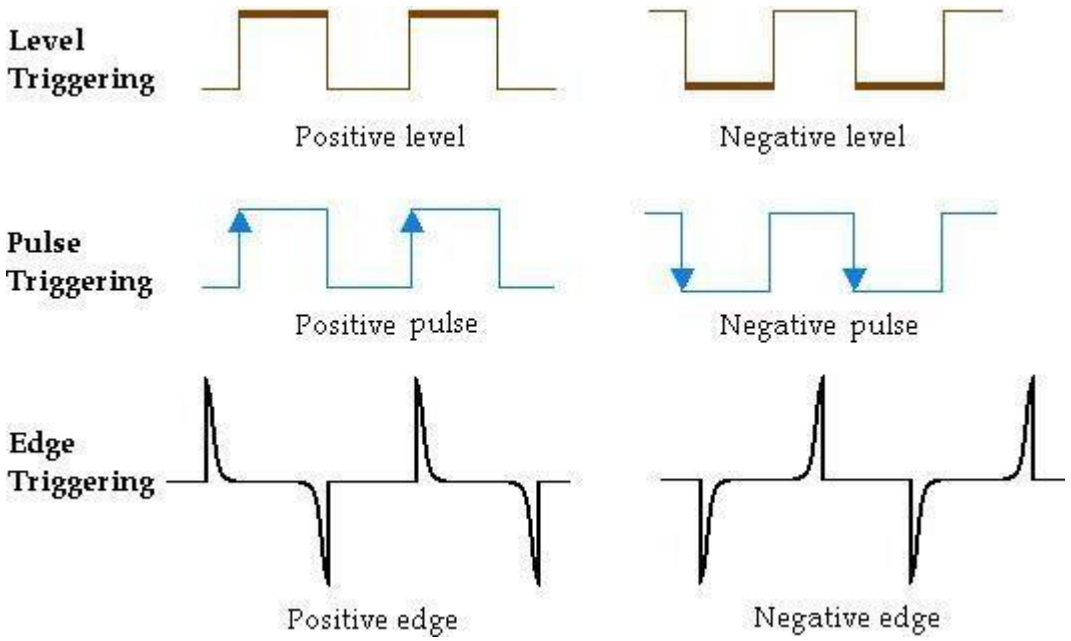


3.4 TRIGGERING OF FLIP-FLOPS

The state of a Flip-Flop is switched by a momentary change in the input signal. This momentary change is called a trigger and the transition it causes is said to trigger the Flip-Flop. Clocked Flip-Flops are triggered by pulses. A clock pulse starts from an initial value of 0, goes momentarily to 1and after a short time, returns to its initial 0 value.

Latches are controlled by enable signal, and they are level triggered, either positive level triggered or negative level triggered. The output is free to change according to the S and R input values, when active level is maintained at the enable input.

Flip-Flops are different from latches. Flip-Flops are pulse or clock edge triggered instead of level triggered.



3.5 EDGE TRIGGERED FLIP-FLOPS

Flip-Flops are synchronous bistable devices (has two outputs Q and Q'). In this case, the term synchronous means that the output changes state only at aspecified point on the triggering input called the clock (CLK), i.e., changes in the output occur in synchronization with the clock.

An ***edge-triggered Flip-Flop*** changes state either at the positive edge (rising edge) or at the negative edge (falling edge) of the clock pulse and is sensitive to its inputs only at this transition of the clock. The different types of edge-triggered Flip-Flops are—

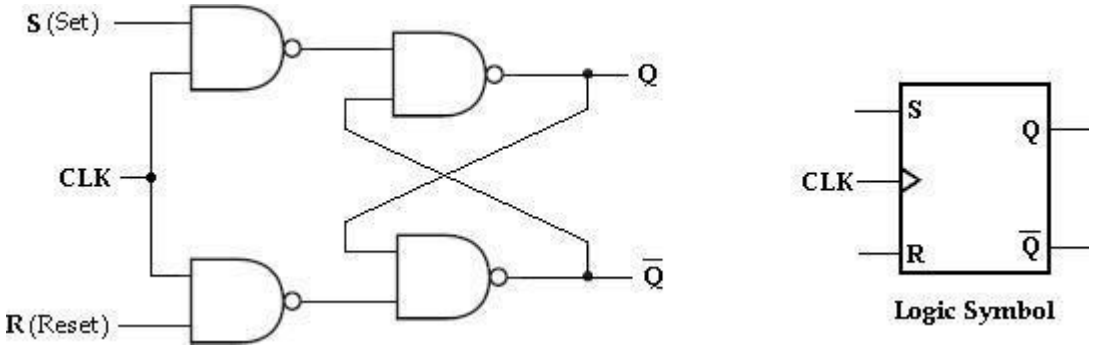
- S-R Flip-Flop,
- J-K Flip-Flop,
- D Flip-Flop,
- T Flip-Flop.

Although the S-R Flip-Flop is not available in IC form, it is the basis for the D and J-K Flip-Flops. Each type can be either positive edge-triggered (no bubble at C

input) or negative edge-triggered (bubble at C input). The key to identifying an edge-triggered Flip-Flop by its logic symbol is the small triangle inside the block at the clock (C) input. This triangle is called the **dynamic input indicator**.

3.5.1 S-R Flip-Flop

The S and R inputs of the S-R Flip-Flop are called ***synchronous*** inputs because data on these inputs are transferred to the Flip-Flop's output only on the triggering edge of the clock pulse. The circuit is similar to SR latch except enable signal is replaced by clock pulse (CLK). On the positive edge of the clock pulse, the circuit responds to the S and R inputs.



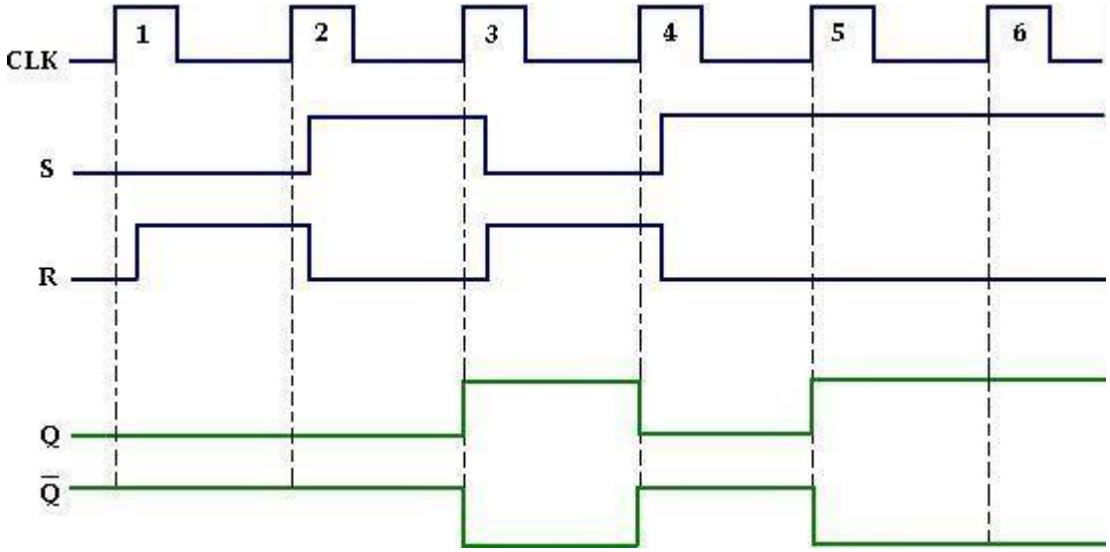
SR Flip-Flop

When S is HIGH and R is LOW, the Q output goes HIGH on the triggering edge of the clock pulse, and the Flip-Flop is SET. When S is LOW and R is HIGH, theQ output goes LOW on the triggering edge of the clock pulse, and the Flip-Flop is RESET.When both S and R are LOW, the output does not change from its prior state. An invalid condition exists when both S and R are HIGH.

CLK	S	R	Q_n	Q_{n+1}	State
1	0	0	0	0	No Change (NC)
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	Set
1	1	0	1	1	

1 1	1 1	1 1	0 1	x x	Indeterminate *
0 0	x x	x x	0 1	0 1	No Change (NC)

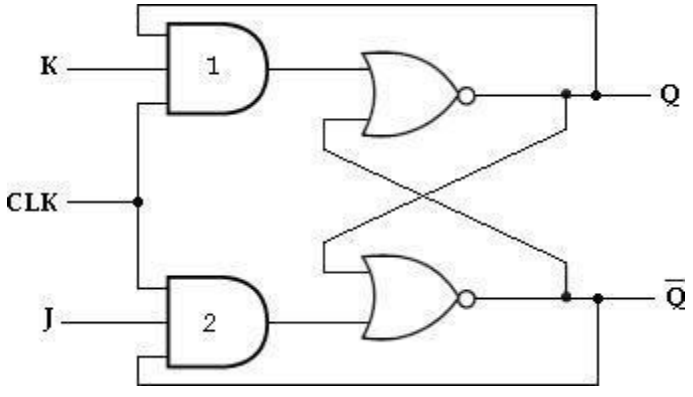
Truth table for SR Flip-Flop



Input and output waveforms of SR Flip-Flop

3.5.2 J-K Flip-Flop:

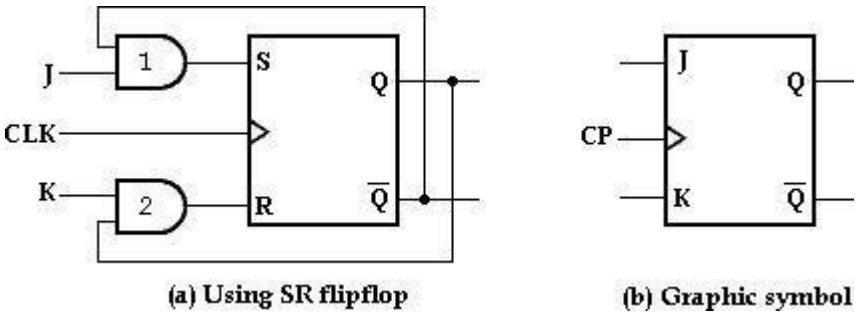
JK means Jack Kilby, Texas Instrument (TI) Engineer, who invented IC in 1958. JK Flip-Flop has two inputs J(set) and K(reset). A JK Flip-Flop can be obtained from the clocked SR Flip-Flop by augmenting two AND gates as shown below.



JK Flip Flop

The data input J and the output Q' are applied o the first AND gate and its output (JQ') is applied to the S input of SR Flip-Flop. Similarly, the data input K and

the output Q are applied to the second AND gate and its output (KQ) is applied to the R input of SR Flip-Flop.



(a) Using SR flipflop (b) Graphic symbol

J=K=0

When J=K= 0, both AND gates are disabled. Therefore clock pulse have no effect, hence the Flip-Flop output is same as the previous output.

J=0,K=1

When J= 0 and K= 1, AND gate 1 is disabled i.e., S= 0 and R= 1. This condition will reset the Flip-Flop to 0.

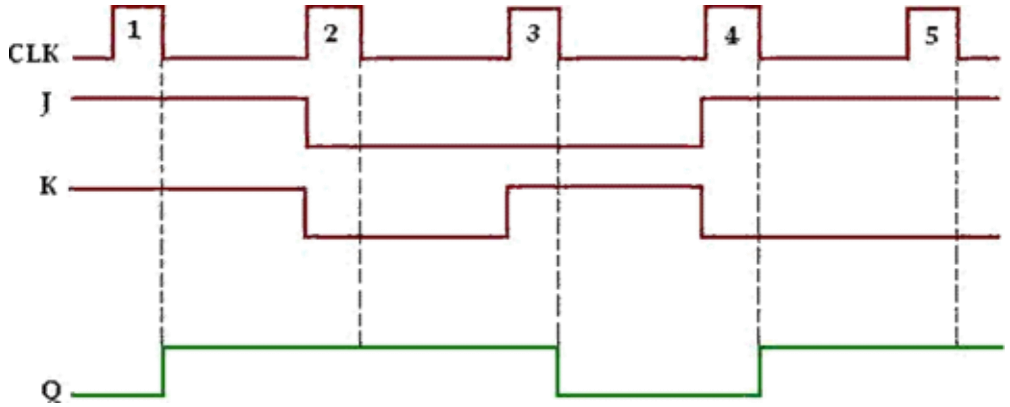
J=1,K=0

When J= 1 and K= 0, AND gate 2 is disabled i.e., S= 1 and R= 0. Therefore the Flip-Flop will set on the application of a clock pulse.

J=K=0

When J=K= 1, it is possible to set or reset the Flip-Flop. If Q is High, AND gate 2 passes on a reset pulse to the next clock. When Q is low, AND gate 1 passes on a set pulse to the next clock. Eitherway, Q changes to the complement of the last state i.e., toggle. Toggle means to switch to the opposite state. The truth table of JK Flip-Flop is given below.

CLK	Inputs		Output	State
	J	K	Q_{n+1}	
1	0	0	Q_n	No Change
1	0	1	0	Reset
1	1	0	1	Set
1	1	1	Q_n'	Toggle



Input and output waveforms of JK Flip-Flop

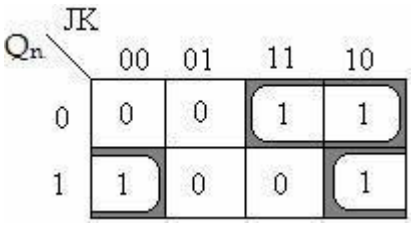
Characteristic table and Characteristic equation:

The characteristic table for JK Flip-Flop is shown in the table below. From the table, K-map for the next state transition (Q_{n+1}) can be drawn and the simplified logic expression which represents the characteristic equation of JK Flip-Flop can be found.

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

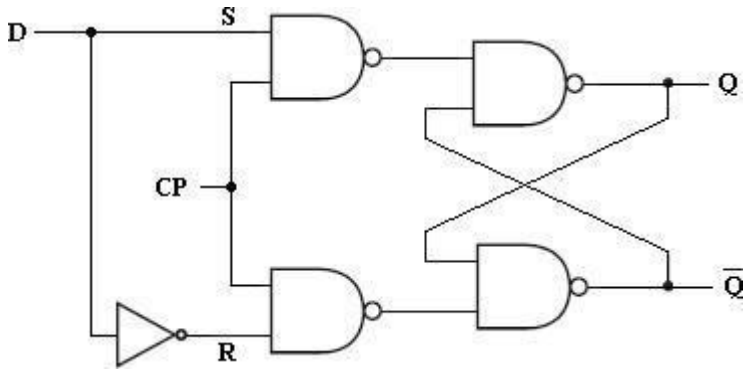
K-map Simplification:



Characteristic equation: **Q_{n+1}= JQ'+ K'Q.**

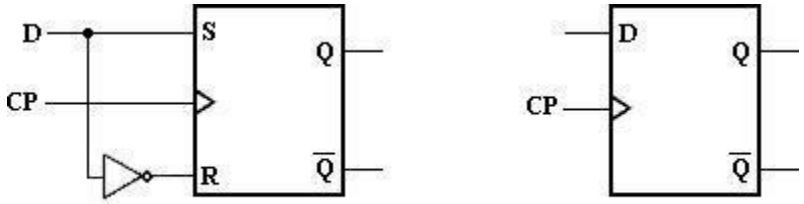
3.5.3 D Flip-Flop:

Like in D latch, in D Flip-Flop the basic SR Flip-Flop is used with complemented inputs. The D Flip-Flop is similar to D-latch except clock pulse is used instead of enable input.



D Flip-Flop

To eliminate the undesirable condition of the indeterminate state in the RS Flip-Flop is to ensure that inputs S and R are never equal to 1 at the same time. This is done by D Flip-Flop. The D (***delay***) Flip-Flop has one input called delay input andclock pulse input. The D Flip-Flop using SR Flip-Flop is shown below.

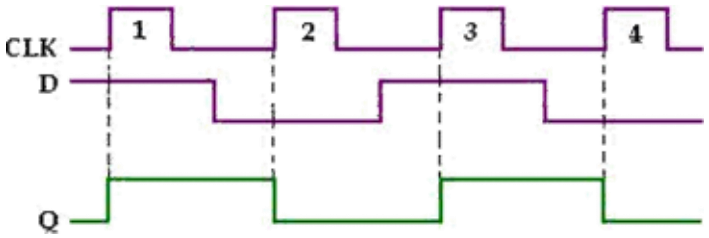


(a) Using SR flipflop (b) Graphic symbol

The truth table of D Flip-Flop is given below.

Clock	D	Q_{n+1}	State
1	0	0	Reset
1	1	1	Set
0	x	Q_n	No Change

Truth table for D Flip-Flop



Input and output waveforms of clocked D Flip-Flop

Looking at the truth table for D Flip-Flop we can realize that Q_{n+1} function follows the D input at the positive going edges of the clock pulses.

Characteristic table and Characteristic equation:

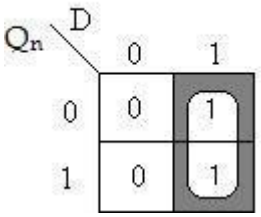
The characteristic table for D Flip-Flop shows that the next state of the Flip- Flop is independent of the present state since Q_{n+1} is equal to D. This means that an inputpulse will transfer the value of input D into the output of the Flip-Flop independent of the value of the output before the pulse was applied.

The characteristic equation is derived from K-map.

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

Characteristic table

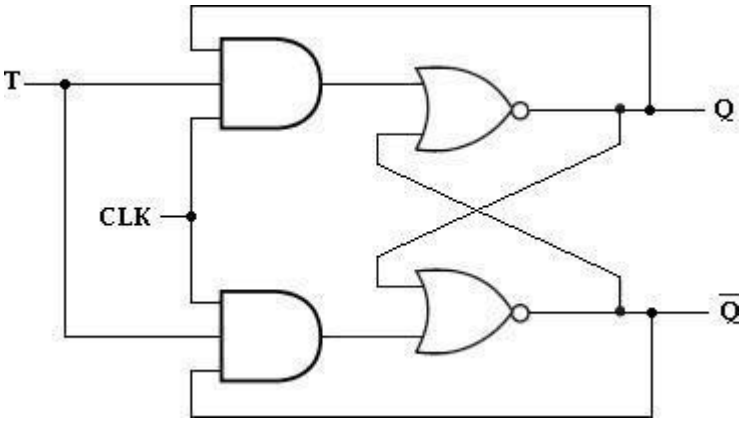
K-map simplification



Characteristic equation: **Q_{n+1}= D.**

3.5.4 T Flip-Flop

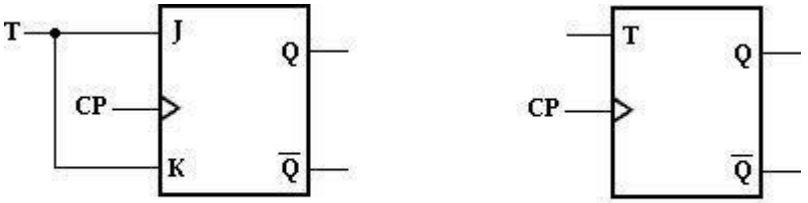
The T (*Toggle*) Flip-Flop is a modification of the JK Flip-Flop. It is obtained from JK Flip-Flop by connecting both inputs J and K together, i.e., single input. Regardless of the present state, the Flip-Flop complements its output when the clock pulse occurs while input T= 1.



T Flip-Flop

When T= 0, $Q_{n+1}= Q_n$, ie., the next state is the sameas the present state and no change occurs.

When T= 1, $Q_{n+1}= Q_n'$,ie., the next state is the complement of the present state.



(a) Using JK flipflop (b) Graphic symbol

The truth table of T Flip-Flop is given below.

T	Q_{n+1}	State
0	Q_n	No Change
1	Q_n'	Toggle

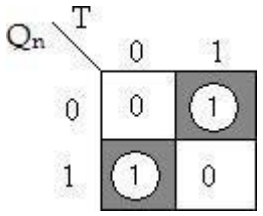
Truth table for T Flip-Flop

Characteristic table and Characteristic equation:

The characteristic table for T Flip-Flop is shown below and characteristic equation is derived using K-map.

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

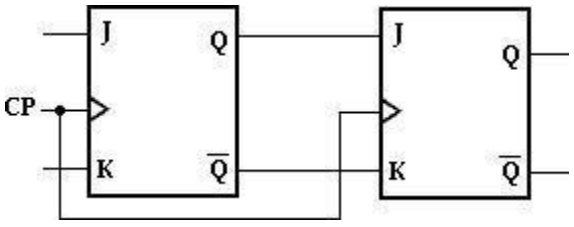
K-map Simplification:



Characteristic equation: $Q_{n+1} = TQ_n' + T'Q_n$.

3.5.5 Master-Slave JK Flip-Flop

A master-slave Flip-Flop is constructed using two separate JK Flip-Flops. The first Flip-Flop is called the master. It is driven by the positive edge of the clock pulse. The second Flip-Flop is called the slave. It is driven by the negative edge of the clock pulse. The logic diagram of a master-slave JK Flip-Flop is shown below.

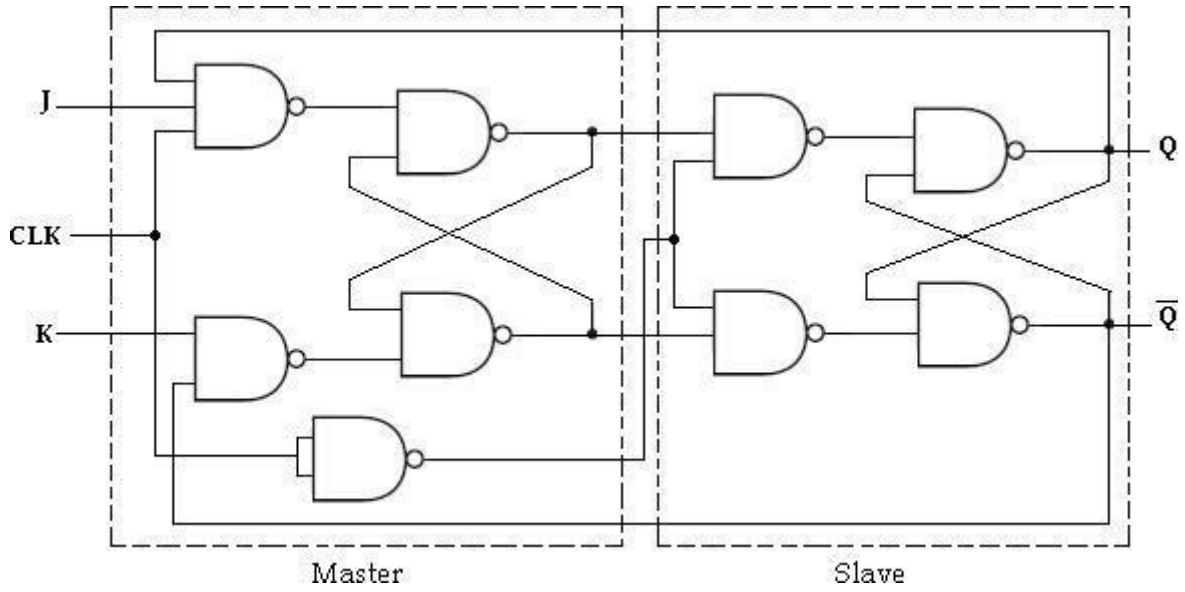


Logic diagram

When the clock pulse has a positive edge, the master acts according to its J-K inputs, but the slave does not respond, since it requires a negative edge at the clock input.

When the clock input has a negative edge, the slave Flip-Flop copies the master outputs. But the master does not respond since it requires a positive edge at its clock input.

The clocked master-slave J-K Flip-Flop using NAND gates is shown below.



Master-Slave JK Flip-Flop

3.6 APPLICATION TABLE (OR) EXCITATION TABLE:

The ***characteristic table*** is useful for **analysis** and for defining the operation of the Flip-Flop. It specifies the next state (Q_{n+1}) when the inputs and present state are known.

The ***excitation or application table*** is useful for **design** process. It is used to find the Flip-Flop input conditions that will cause the required transition, when the present state (Q_n) and the next state (Q_{n+1}) are known.

3.6.1 SR Flip-Flop:

Present State	Inputs		Next State
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

Present State	Next State	Inputs		Inputs	
Q_n	Q_{n+1}	S	R	S	R
0	0	0	0	0	x
0	0	0	1		
0	1	1	0	1	0
1	0	0	1	0	1
1	1	0	0	x	0
1	1	1	0		

Modified Table

Present State	Next State	Inputs	
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

The above table presents the excitation table for SR Flip-Flop. It consists of present state (Q_n), next state (Q_{n+1}) and a column for each input to show how the required transition is achieved.

There are 4 possible transitions from present state to next state. The required Input conditions for each of the four transitions are derived from the information available in the characteristic table. The symbol 'x' denotes the don't care condition, it does not matter whether the input is 0 or 1.

3.6.2 JK Flip-Flop:

Present State	Inputs		Next State
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

Present State	Next State	Inputs		Inputs	
Q_n	Q_{n+1}	J	K	J	K
0	0	0	0	0	x
0	0	0	1		
0	1	1	0	1	x
0	1	1	1		
1	0	0	1	x	1
1	0	1	1		
1	1	0	0	x	0
1	1	1	0		

Modified Table

Present State	Next State	Inputs	
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

3.6.3 D Flip-Flop

Present State	Input	Next State
Q_n	D	Q_{n+1}
0	0	0
0	1	1
1	0	0
1	1	1

Characteristic Table

Present State	Next State	Input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation Table

3.6.4 T Flip-Flop

Present State	Input	Next State
Q_n	T	Q_{n+1}
0	0	0
0	1	1
1	0	1
1	1	0

Characteristic Table

Present State	Next State	Input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Modified Table

3.7 REALIZATION OF ONE FLIP-FLOP USING OTHER FLIP-FLOPS

It is possible to convert one Flip-Flop into another Flip-Flop with some additional gates or simply doing some extra connection. The realization of one Flip-Flop using other Flip-Flops is implemented by the use of characteristic tables and excitation tables. Let us see few conversions among Flip-Flops.

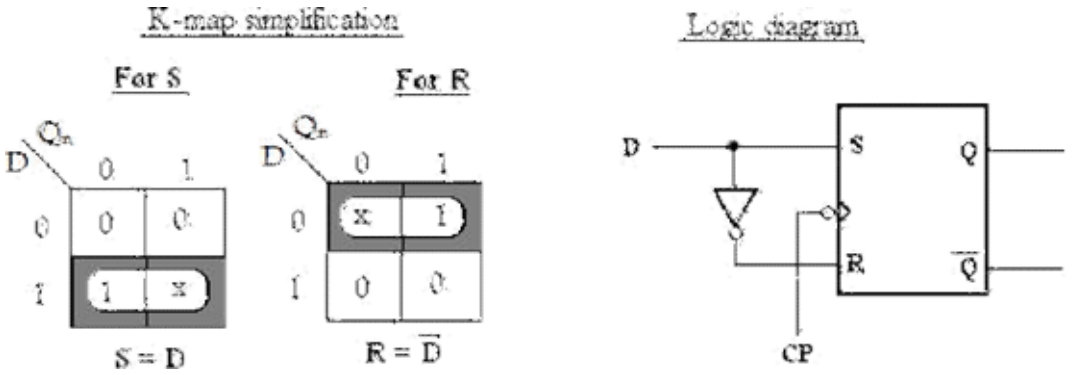
- SR Flip-Flop to D Flip-Flop
- SR Flip-Flop to JK Flip-Flop
- SR Flip-Flop to T Flip-Flop
- JK Flip-Flop to T Flip-Flop
- JK Flip-Flop to D Flip-Flop
- D Flip-Flop to T Flip-Flop
- T Flip-Flop to D Flip-Flop

3.7.1 SR Flip-Flop to D Flip-Flop:

- Write the characteristic table for required Flip-Flop (D Flip-Flop).
- Write the excitation table for given Flip-Flop (SR Flip-Flop).
- Determine the expression for the given Flip-Flop inputs (S and R) by using K- map.
- Draw the Flip-Flop conversion logic diagram to obtain the required Flip-Flop (D Flip-Flop) by using the above obtained expression.

The excitation table for the above conversion is

Required Flip-Flop (D)			**Given Flip-Flop (SR)**	
Input	**Present state**	**Next state**	**Flip-Flop Inputs**	
D	**Q_n**	**Q_{n+1}**	**S**	**R**
0	0	0	0	x
0	1	0	0	1
1	0	1	1	0
1	1	1	x	0

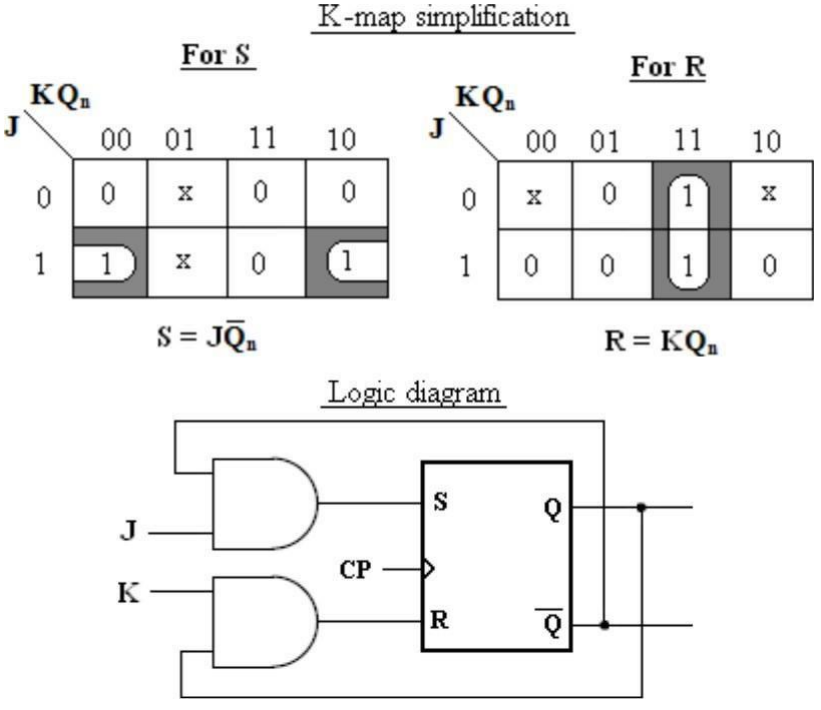


D Flip-Flop

3.7.2 SR Flip-Flop to JK Flip-Flop

The excitation table for the above conversion is,

Inputs		Present state	Next state	Flip-Flop Input	
J	**K**	**Q_n**	**Q_{n+1}**	**S**	**R**
0	0	0	0	0	x
0	0	1	1	x	0
0	1	0	0	0	x
0	1	1	0	0	1
1	0	0	1	1	0
1	0	1	1	x	0
1	1	0	1	1	0
1	1	1	0	0	1



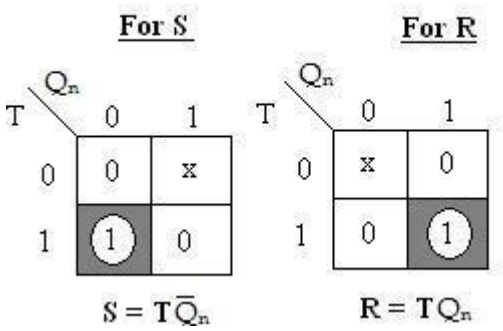
JK Flip-Flop

2.7.3 SR Flip-Flop to T Flip-Flop

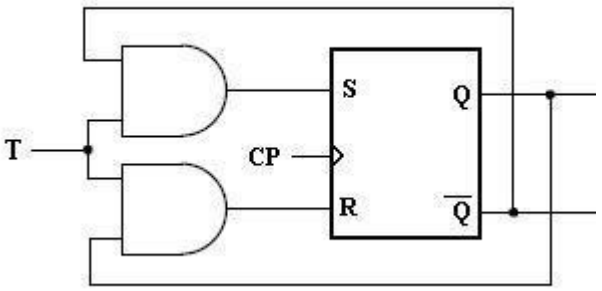
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
T	**Q_n**	**Q_{n+1}**	**S**	**R**
0	0	0	0	x
0	1	1	x	0
1	0	1	1	0
1	1	0	0	1

K-map simplification



Logic diagram

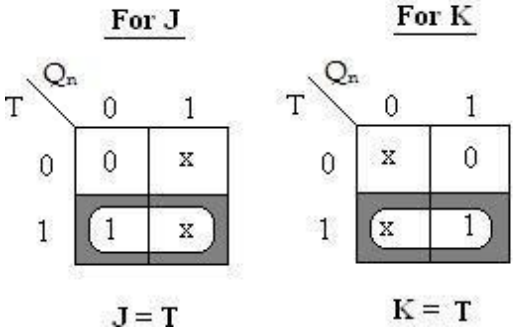


3.7.4 JK Flip-Flop to T Flip-Flop

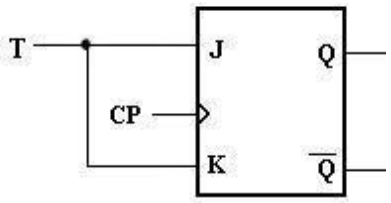
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
T	**Q_n**	**Q_{n+1}**	**J**	**K**
0	0	0	0	x
0	1	1	x	0
1	0	1	1	x
1	1	0	x	1

K-map simplification



Logic diagram

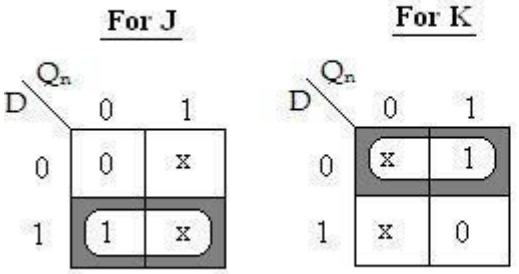


JK Flip-Flop to D Flip-Flop

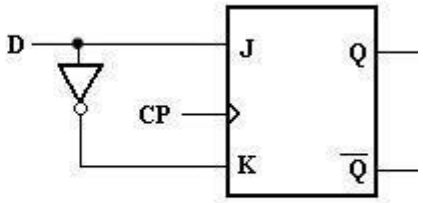
The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Inputs	
D	**Q_n**	**Q_{n+1}**	**J**	**K**
0	0	0	0	x
0	1	0	x	1
1	0	1	1	x
1	1	1	x	0

K-map simplification



Logic diagram

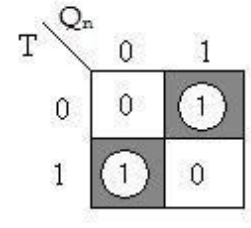


D Flip-Flop to T Flip-Flop

The excitation table for the above conversion is

Input	Present state	Next state	Flip-Flop Input
T	**Q_n**	**Q_{n+1}**	**D**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

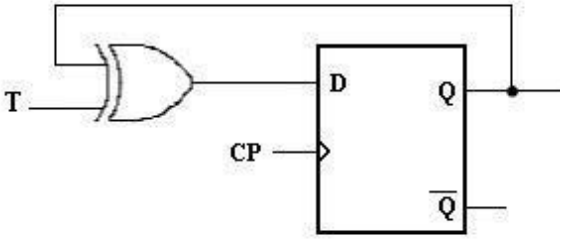
K-map simplification



$D = \overline{T}Q_n + T\overline{Q}_n$

$= T \oplus Q_n$

Logic diagram

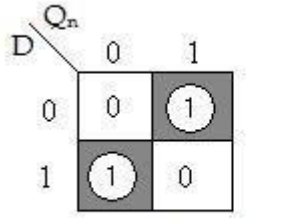


T Flip-Flop to D Flip-Flop

The excitation table for the above conversion is

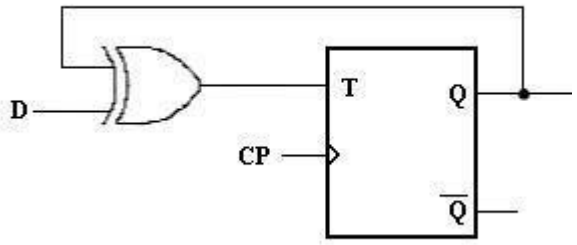
Input	Present state	Next state	Flip-Flop Input
D	Q_n	Q_{n+1}	T
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

K-map simplification



$$T = D\overline{Q}_n + \overline{D}Q_n$$
$$= D \oplus Q_n$$

Logic diagram



3.8 CLASSIFICATION OF SYNCHRONOUS SEQUENTIAL CIRCUIT:

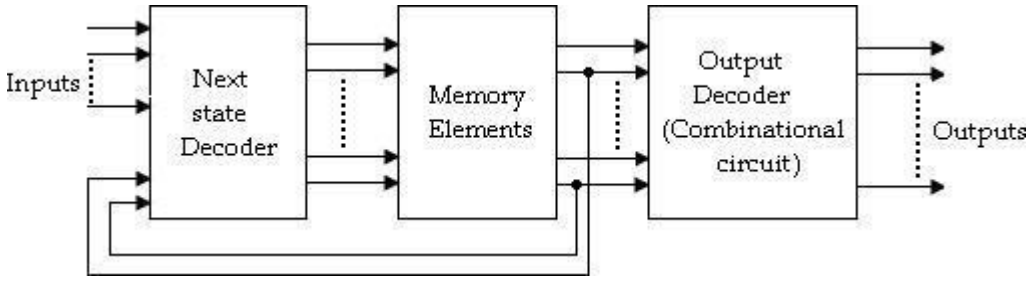
In synchronous or clocked sequential circuits, clocked Flip-Flops are used as memory elements, which change their individual states in synchronism with the periodic clock signal. Therefore, the change in states of Flip-Flop and change in state of the entire circuits occur at the transition of the clock signal.

The synchronous or clocked sequential networks are represented by two models.

- **Moore model:** The output depends only on the present state of the Flip-Flops.
- **Mealy model:** The output depends on both the present state of the Flip-Flops and on the inputs.

3.8.1 Moore model:

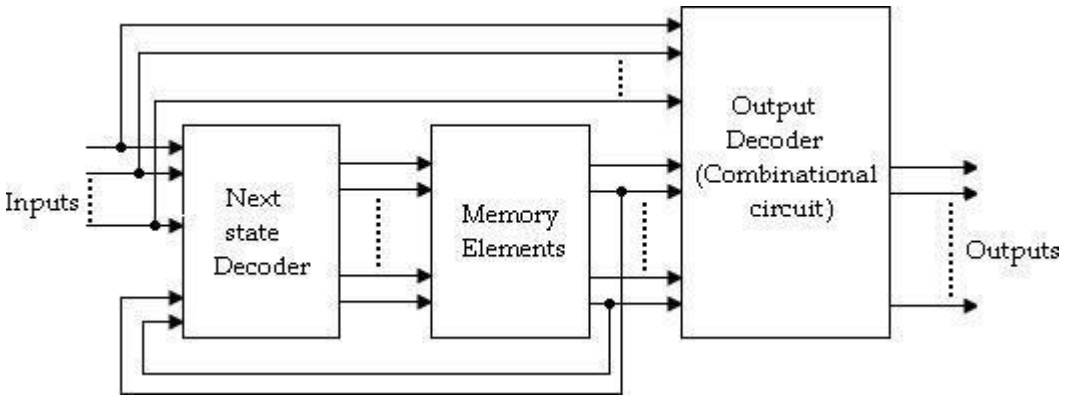
In the Moore model, the outputs are a function of the present state of the Flip-Flops only. The output depends only on present state of Flip-Flops, it appears only after the clock pulse is applied, i.e., it varies in synchronism with the clock input.



Moore model

3.8.2 Mealy model:

In the Mealy model, the outputs are functions of both the present state of the Flip-Flops and inputs.



Mealy model

3.8.3 Difference between Moore and Mealy model

Sl.No	Moore model	Mealy model
1	Its output is a function of present state only.	Its output is a function of present state as well as present input.
2	Input changes does not affect the output.	Input changes may affect the output of the circuit.
3	It requires more number of states for implementing same function.	It requires less number of states for implementing same function.

3.9 ANALYSIS OF SYNCHRONOUS SEQUENTIAL CIRCUIT:

The behavior of a sequential circuit is determined from the inputs, outputs and the state of its Flip-Flops. The outputs and the next state are both a function of the inputs and the present state. The analysis of a sequential circuit consists of obtaining a table or diagram from the time sequence of inputs, outputs and internal states.

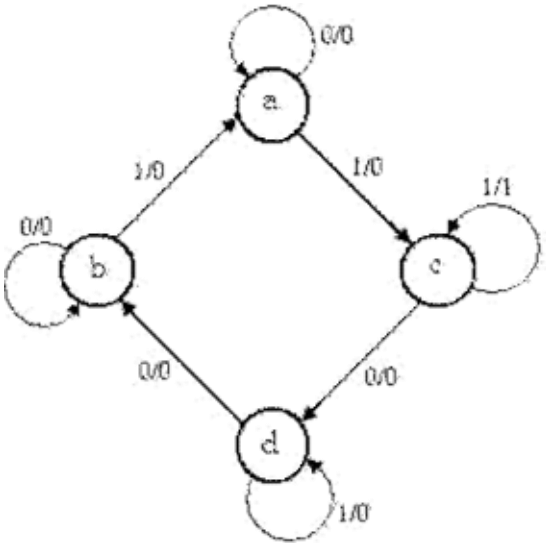
Before going to see the analysis and design examples, we first understand the state diagram, state table.

3.9.1 State Diagram

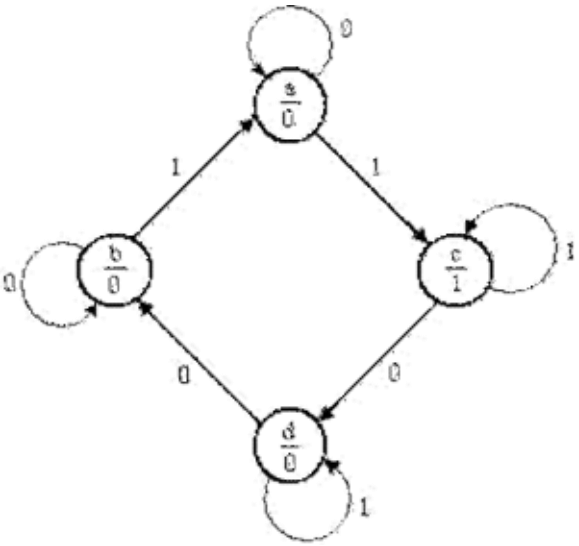
State diagram is a pictorial representation of a behavior of a sequential circuit.

- In the state diagram, a state is represented by a circle and the transition between states is indicated by directed lines connecting the circles.
- A directed line connecting a circle with circle with itself indicates that next state is same as present state.
- The binary number inside each circle identifies the state represented by the circle.
- The directed lines are labeled with two binary numbers separated by a symbol '/'. The input value that causes the state transition is labeled first and the output value during the present state is labeled after the symbol '/'.

In case of Moore circuit, the directed lines are labeled with only one binary number representing the state of the input that causes the state transition. The output state is indicated within the circle, below the present state because output state depends only on present state and not on the input.



State diagram for Mealy circuit



State diagram for Moore circuit

3.9.2 State Table

State table represents relationship between input, output and Flip-Flop states.

- It consists of three sections labeled present state, next state and output.
  - The present state designates the state of Flip-Flops before the occurrence of a clock pulse, and the output section gives the values of the output variables during the present state.
  - Both the next state and output sections have two columns representing two possible input conditions: X= 0 and X=1.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
AB	AB	AB	Y	Y
a	a	c	0	0
b	b	a	0	0
c	d	c	0	1
d	b	d	0	0

- In case of Moore circuit, the output section has only one column since output does not depend on input.

Present state	Next state		Output
	X= 0	X= 1	Y
AB	AB	AB	
a	a	c	0
b	b	a	0
c	d	c	1
d	b	d	0

2.9.3 State Equation

It is an algebraic expression that specifies the condition for a Flip-Flop state transition.

The Flip-Flops may be of any type and the logic diagram may or may not include combinational circuit gates.

3.9.4 ANALYSIS PROCEDURE

The synchronous sequential circuit analysis is summarizes as given below:

1. Assign a state variable to each Flip-Flop in the synchronous sequential circuit.
2. Write the excitation input functions for each Flip-Flop and also write the Moore/ Mealy output equations.
3. Substitute the excitation input functions into the bistable equations for the Flip-Flops to obtain the next state output equations.
4. Obtain the state table and reduced form of the state table.
5. Draw the state diagram by using the second form of the state table.

3.9.5 Analysis of Mealy Model

1. A sequential circuit has two JK Flip-Flops A and B, one input (x) and one output (y). the Flip-Flop input functions are,

   $J_A = B + x$ $J_B = A' + x'$

   $K_A = 1$ $K_B = 1$

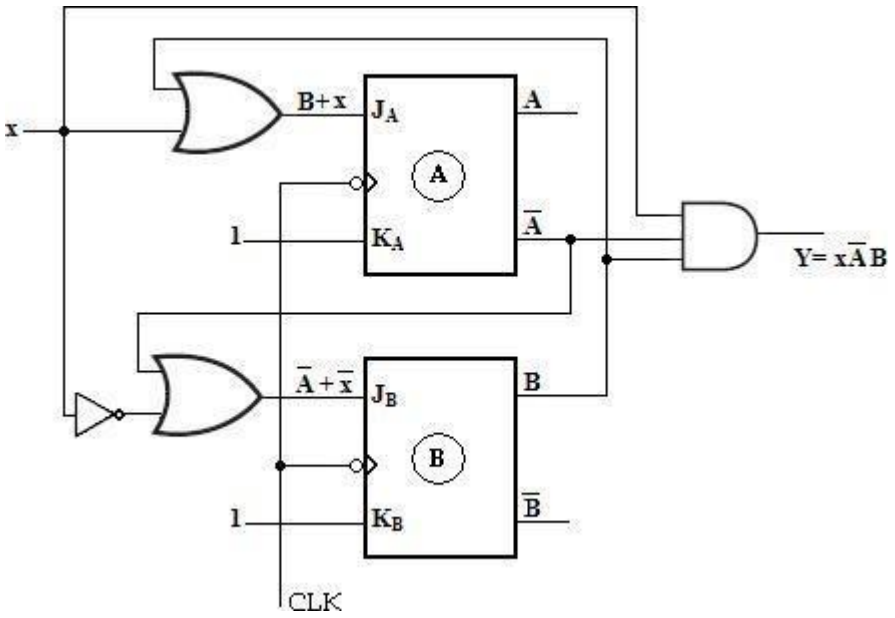
   and the circuit output function, **Y= xA'B**.

a) Draw the logic diagram of the Mealy circuit,

b) Tabulate the state table,

c) Draw the state diagram.

Soln:



State table:

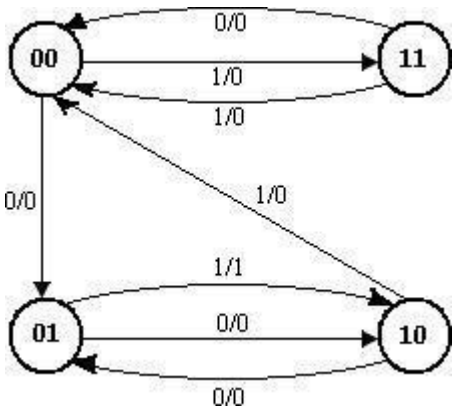
To obtain the next-state values of a sequential circuit with JK Flip-Flops, use the JK Flip-Flop characteristics table.

Present state		Input	Flip-Flop Inputs				Next state		Output
A	B	x	$J_A = B + x$	$K_A = 1$	$J_B = A' + x'$	$K_B = 1$	A(t+1)	B(t+1)	Y= xA'B
0	0	0	0	1	1	1	0	1	0
0	0	1	1	1	1	1	1	1	0
0	1	0	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	1	1	0	1	0
1	0	1	1	1	0	1	0	0	0
1	1	0	1	1	1	1	0	0	0
1	1	1	1	1	0	1	0	0	0

Present state		Next state				Output	
		x= 0		x= 1		x= 0	x= 1
A	B	A	B	A	B	y	y
0	0	0	1	1	1	0	0
0	1	1	0	1	0	0	1
1	0	0	1	0	0	0	0
1	1	0	0	0	0	0	0

Second form of state table

State Diagram:



State Diagram

2. A sequential circuit with two 'D' Flip-Flops A and B, one input (x) and one output (y). the Flip-Flop input functions are:

   $D_A = Ax + Bx$

   $D_B = A'x$ and the circuit output function is,

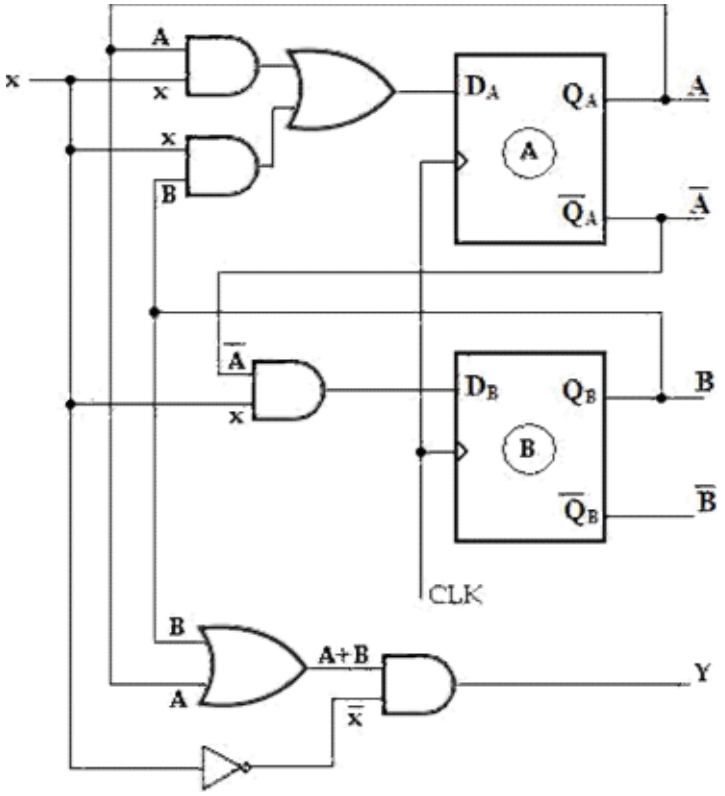
   $Y = (A + B)x'$.

   (a) Draw the logic diagram of the circuit,

   (b) Tabulate the state table,

   (c) Draw the state diagram.

Soln:



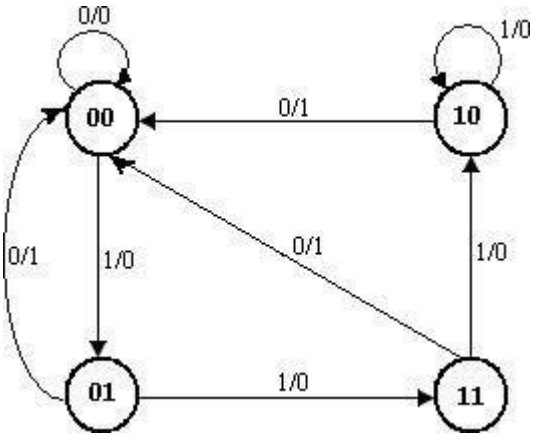
State Table:

Present state		Input	Flip-Flop Inputs		Next state		Output
A	**B**	**x**	**D_A= Ax+Bx**	**D_B= A'x**	**A(t+1)**	**B(t+1)**	**Y= (A+B)x'**
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	0	0	0	1
0	1	1	1	1	1	1	0
1	0	0	0	0	0	0	1
1	0	1	1	0	1	0	0
1	1	0	0	0	0	0	1
1	1	1	1	0	1	0	0

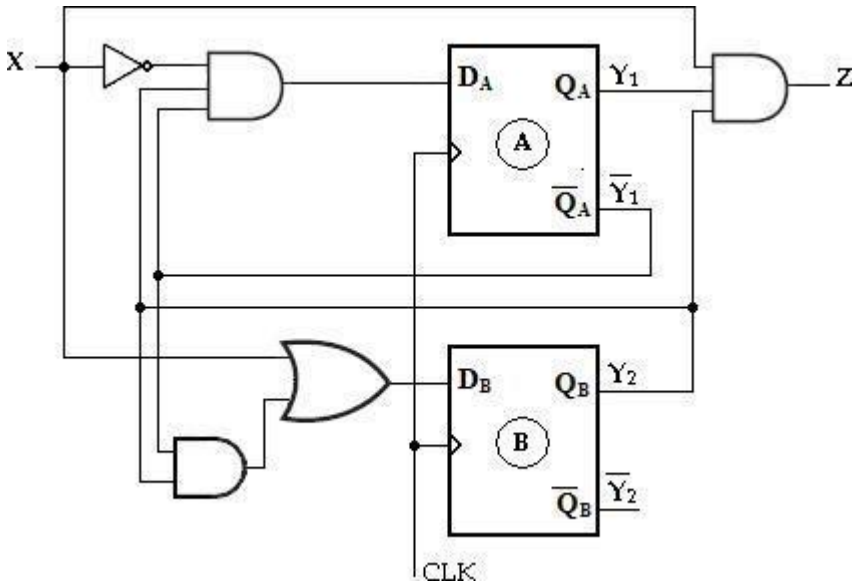
Present state		Next state				Output	
		x= 0		x= 1		x= 0	x= 1
A	**B**	**A**	**B**	**A**	**B**	**Y**	**Y**
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

Second form of state table

State Diagram:



3. Analyze the synchronous Mealy machine and obtain its state diagram.



Soln:

The given synchronous Mealy machine consists of two D Flip-Flops, one inputs and one output.

The Flip-Flop input functions are,

$$D_A = Y_1'Y_2X'$$

$$D_B = X + Y_1'Y_2$$

The circuit output function is, $Z = Y_1Y_2X$

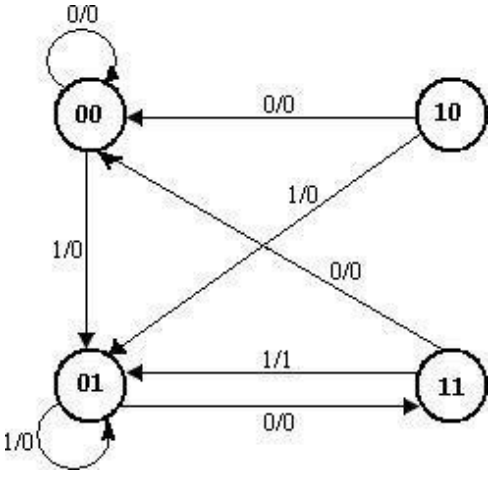
State Table:

Present state		Input	Flip-Flop Inputs		Next state		Output
Y_1	Y_2	X	$D_A = Y_1'Y_2X'$	$D_B = X + Y_1'Y_2$	$Y_1(t+1)$	$Y_2(t+1)$	$Z = Y_1Y_2X$
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	1	1	1	1	0
0	1	1	0	1	0	1	0
1	0	0	0	0	0	0	0
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	0
1	1	1	0	1	0	1	1

Present state		Next state				Output	
		X= 0		X= 1		X= 0	X= 1
Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	Z	Z
0	0	0	0	0	1	0	0
0	1	1	1	0	1	0	0
1	0	0	0	0	1	0	0
1	1	0	0	0	1	0	1

Second form of state table

State Diagram:



4. A sequential circuit has two JK Flop-Flops A and B, two inputs x and y and one output z. The Flip-Flop input equation and circuit output equations are

$J_A = Bx + B'y'$ $K_A = B'xy'$

$J_B = A'x$ $K_B = A + xy'$

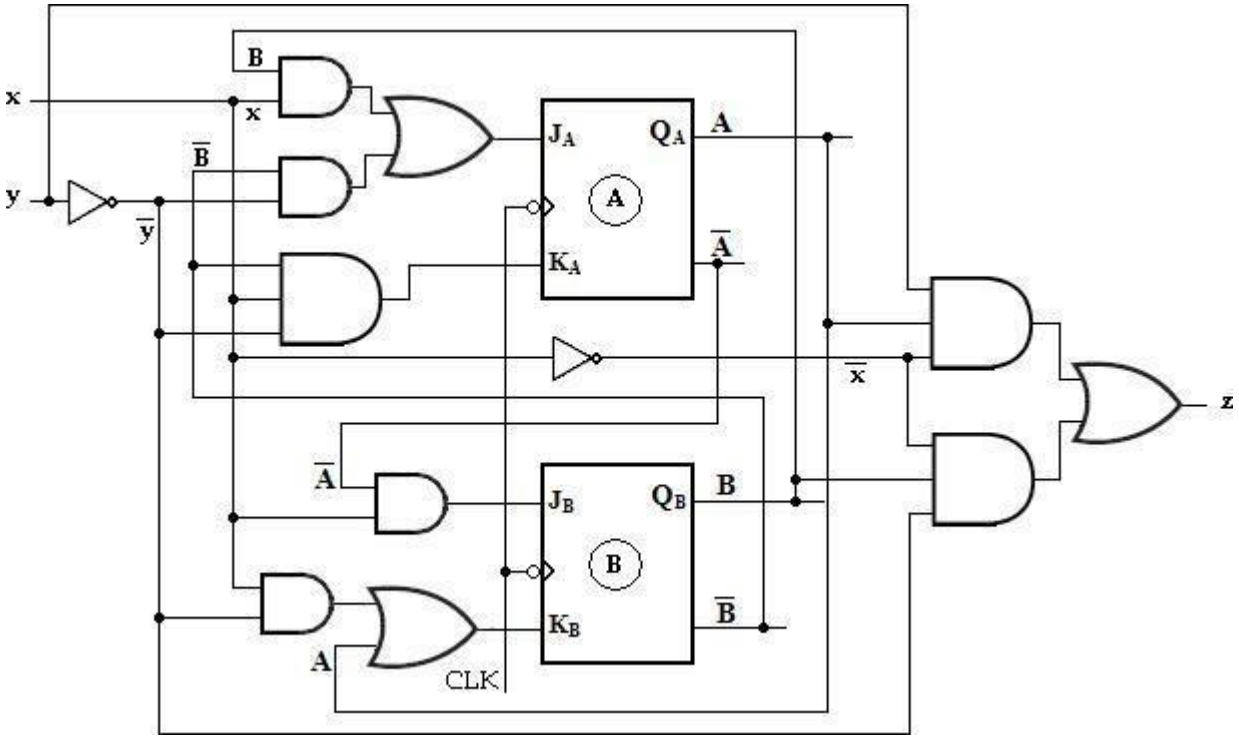
$z = Ax'y' + Bx'y'$

(a) Draw the logic diagram of the circuit

(b) Tabulate the state table.

(c) Derive the state equation.

State diagram:



State table:

To obtain the next-state values of a sequential circuit with JK Flip-Flop, use the JK Flip-Flop characteristic table,

Present state		Input		Flip-Flop Inputs				Next state		Output
A	B	x	y	J_A= Bx+B'y'	K_A= B'xy'	J_B= A'x	K_B= A+xy'	A(t+1)	B(t+1)	z
0	0	0	0	1	0	0	0	1	0	0
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1	1	1	0
0	0	1	1	0	0	1	0	0	1	0
0	1	0	0	0	0	0	0	0	0	1
0	1	0	1	0	0	0	0	0	0	0
0	1	1	0	1	0	1	1	1	1	0
0	1	1	1	1	0	1	0	1	1	0
1	0	0	0	1	0	0	1	1	0	1
1	0	0	1	0	0	0	1	1	0	0
1	0	1	0	1	1	0	1	0	0	0
1	0	1	1	0	0	0	1	1	0	0
1	1	0	0	0	0	0	1	1	0	1
1	1	0	1	0	0	0	1	1	0	0
1	1	1	0	1	0	0	1	1	0	0
1	1	1	1	1	0	0	1	1	0	0

State Equation:

For A(t+1)

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

A (t+1) = Ax' + Ay + Bx + A'B'y'

For B(t+1)

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

B (t+1) = A'x

5. A sequential circuit has two JK Flip-Flop A and B. the Flip-Flop input functions are: $J_A = B$ $J_B = x'$

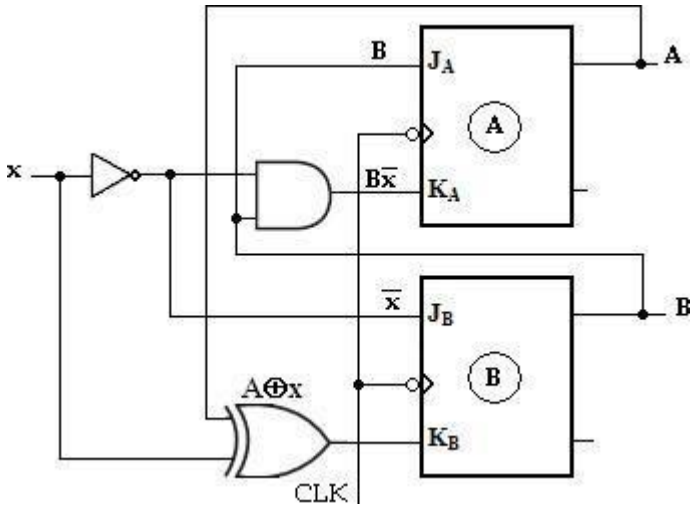
$K_A = Bx'$ $K_B = A \oplus x$.

(a) Draw the logic diagram of the circuit,

(b) Tabulate the state table,

(c) Draw the state diagram.

Logic diagram:



The output function is not given in the problem. The output of the Flip-Flops may be considered as the output of the circuit.

State table:

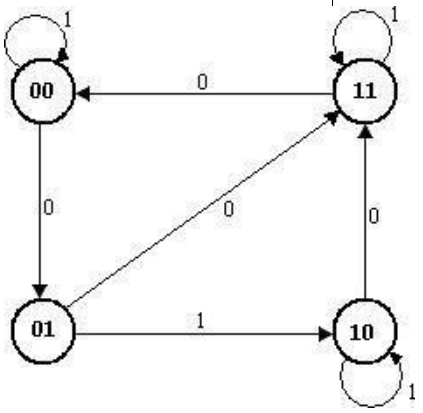
To obtain the next-state values of a sequential circuit with JK Flip-Flop, use the JK Flip-Flop characteristic table.

Present state		Input	Flip-Flop Inputs				Next state	
A	B	x	$J_A = B$	$K_A = Bx'$	$J_B = x'$	$K_B = A\,x$	A(t+1)	B(t+1)
0	0	0	0	0	1	0	0	1
0	0	1	0	0	0	1	0	0
0	1	0	1	1	1	0	1	1
0	1	1	1	0	0	1	1	0
1	0	0	0	0	1	1	1	1
1	0	1	0	0	0	0	1	0
1	1	0	1	1	1	1	0	0
1	1	1	1	0	0	0	1	1

		Next state			
		X= 0		X= 1	
A	B	A	B	A	B
0	0	0	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	0	0	1	1

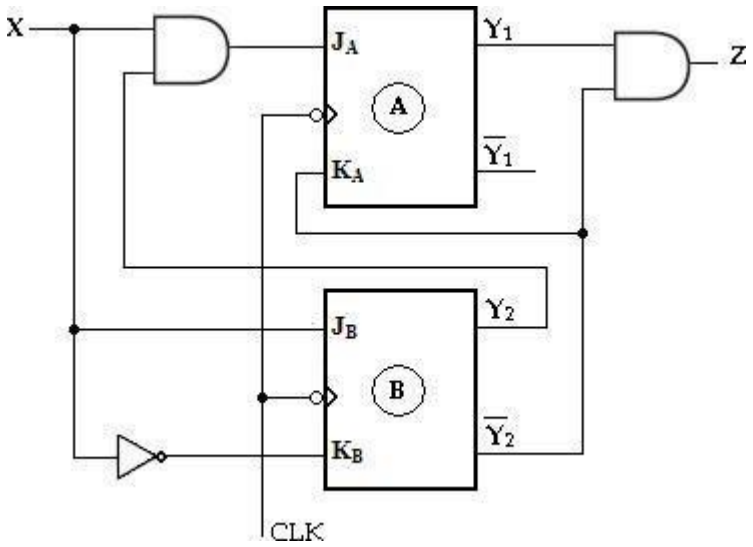
Second form of state table

State Diagram:



3.9.6 Analysis of Moore Model

6. Analyze the synchronous Moore circuit and obtain its state diagram.



Soln:

Using the assigned variable Y_1 and Y_2 for the two JK Flip-Flops, we can write the four excitation input equations and the Moore output equation as follows:

$J_A = Y_2X$; $K_A = Y_2'$

$J_B = X$; $K_B = X'$ and output function, $Z = Y_1Y_2'$

State table:

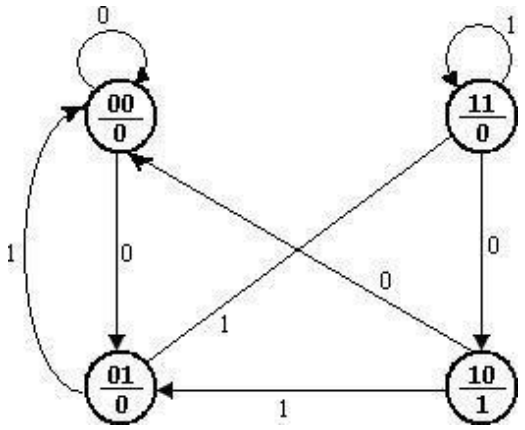
Present state		Input	Flip-Flop Inputs				Next state		Output
Y_1	Y_2	X	$J_A = Y_2X$	$K_A = Y_2'$	$J_B = X$	$K_B = X'$	$Y_1(t+1)$	$Y_2(t+1)$	$Z = Y_1Y_2'$
0	0	0	0	1	0	1	0	0	0
0	0	1	0	1	1	0	0	1	0
0	1	0	0	0	0	1	0	0	0
0	1	1	1	0	1	0	1	1	0
1	0	0	0	1	0	1	0	0	1
1	0	1	0	1	1	0	0	1	1
1	1	0	0	0	0	1	1	0	0
1	1	1	1	0	1	0	1	1	0

Present state		Next state				Output
		X= 0		X= 1		Y
Y_1	Y_2	Y_1	Y_2	Y_1	Y_2	
0	0	0	0	0	1	0
0	1	0	0	1	1	0
1	0	0	0	0	1	1
1	1	1	0	1	1	0

Second form of state table

State Diagram:

Here the output depends on the present state only and is independent of the input. The two values inside each circle separated by a slash are for the present state and output.



7. A sequential circuit has two T Flip-Flop A and B. The Flip-Flop input functions are:

   $T_A = Bx$ $T_B = x$

   $y = AB$

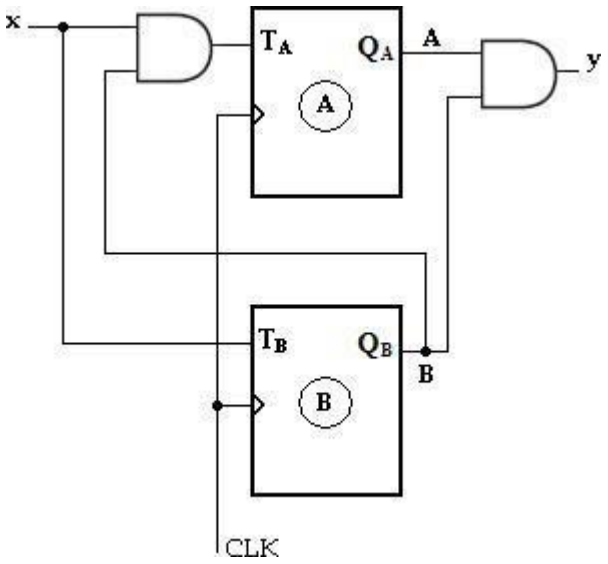
   (a) Draw the logic diagram of the circuit,

   (b) Tabulate the state table,

   (c) Draw the state diagram.

Soln:

Logic diagram:



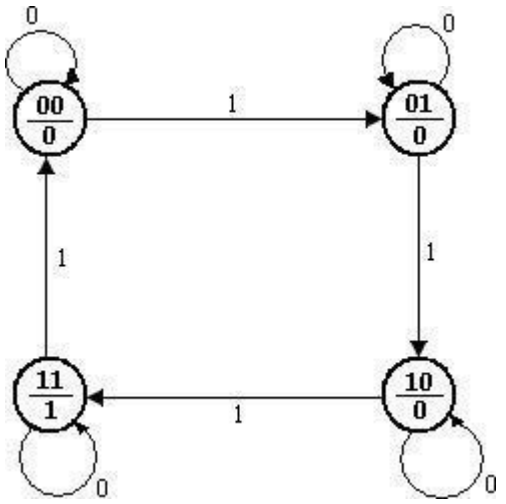
State table

Present state		Input	Flip-Flop Inputs		Next state		Output
A	B	x	$T_A = Bx$	$T_B = x$	A (t+1)	B (t+1)	y= AB
0	0	0	0	0	0	0	0
0	0	1	0	1	0	1	0
0	1	0	0	0	0	1	0
0	1	1	1	1	1	0	0
1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	0
1	1	0	0	0	1	1	1
1	1	1	1	1	0	0	1

Present state		Next state				Output	
		x= 0		x= 1		x= 0	x= 1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	1	1	0	0	0
1	0	1	0	1	1	0	0
1	1	1	1	0	0	1	1

Second form of state table

State Diagram:



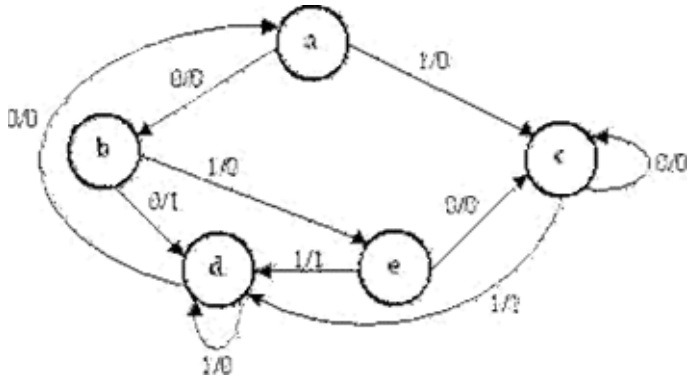
3.10 STATE REDUCTION/ MINIMIZATION

The state reduction is used to avoid the redundant states in the sequential circuits. The reduction in redundant states reduces the number of required Flip-Flops and logic gates, reducing the cost of the final circuit.

The two states are said to be redundant or equivalent, if every possible set of inputs generate exactly same output and same next state. When two states are equivalent, one of them can be removed without altering the input-output relationship.

Since 'n' Flip-Flops produced 2_n state, a reduction in the number of states may result in a reduction in the number of Flip-Flops.

The need for state reduction or state minimization is explained with one example.



State diagram

Step 1: Determine the state table for given state diagram

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	b	c	0	0
b	d	e	1	0
c	c	d	0	1
d	a	d	0	0
e	c	d	0	1

State table

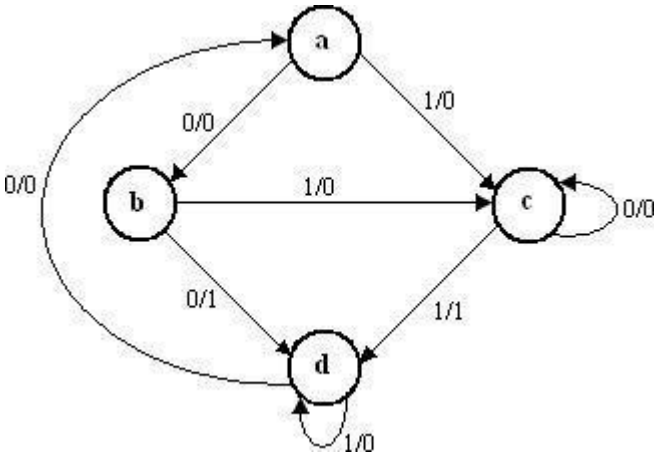
Step 2: Find equivalent states

From the above state table **c** and **e** generate exactly same next state and same output for every possible set of inputs. The state **c** and **e** go to next states **c** and **d** and have outputs 0 and 1 for x=0 and x=1 respectively. Therefore state **e** can be removed and replaced by **c**. The final reduced state table is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	b	c	0	0
b	d	c	1	0
c	c	d	0	1
d	a	d	0	0

Reduced state table

The state diagram for the reduced table consists of only four states and is shown below.



Reduced state diagram

1. Reduce the number of states in the following state table and tabulate the reduced state table.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

Soln:

From the above state table **e** and **g** generate exactly same next state and same output for every possible set of inputs. The state **e** and **g** go to next states **a** and **f** and have outputs 0 and 1 for x=0 and x=1 respectively. Therefore state **g** can be removed and replaced by **e**.

The reduced state table-1 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	**e**	f	0	1

Reduced state table-1

Now states d and f are equivalent. Both states go to the same next state (e, f) and have same output (0, 1). Therefore one state can be removed; **f** is replaced by **d.** The final reduced state table-2 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	**d**	0	0
d	e	**d**	0	1
e	a	**d**	0	1

Reduced state table-2

Thus 7 states are reduced into 5 states.

2. Determine a minimal state table equivalent furnished below

Present state	Next state	
	X= 0	X= 1
1	1, 0	1, 0
2	1, 1	6, 1
3	4, 0	5, 0
4	1, 1	7, 0
5	2, 0	3, 0
6	4, 0	5, 0
7	2, 0	3, 0

Soln:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
1	1	1	0	0
2	1	6	1	1
3	4	5	0	0
4	1	7	1	0
5	2	3	0	0
6	4	5	0	0
7	2	3	0	0

From the above state table, **5** and **7** generate exactly same next state and same output for every possible set of inputs. The state **5** and **7** go to next states **2** and **3** and have outputs 0 and 0 for x=0 and x=1 respectively. Therefore state **7** can be removed and replaced by **5**.

Similarly, **3** and **6** generate exactly same next state and same output for every possible set of inputs. The state **3** and **6** go to next states **4** and **5** and have outputs 0 and 0 for x=0 and x=1 respectively. Therefore state **6** can be removed and replaced by **3**.

The final reduced state table is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
1	1	1	0	0
2	1	3	1	1
3	4	5	0	0
4	1	5	1	0
5	2	3	0	0

Reduced state table

Thus 7 states are reduced into 5 states.

3. Minimize the following state table.

Present state	Next state	
	X= 0	X= 1
A	D, 0	C, 1
B	E, 1	A, 1
C	H, 1	D, 1
D	D, 0	C, 1
E	B, 0	G, 1
F	H, 1	D, 1
G	A, 0	F, 1
H	C, 0	A, 1
I	G, 1	H,1

Soln:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
A	D	C	0	1
B	E	A	1	1
C	H	D	1	1
D	D	C	0	1
E	B	G	0	1
F	H	D	1	1
G	A	F	0	1
H	C	A	0	1
I	G	H	1	1

From the above state table, **A** and **D** generate exactly same next state and same output for every possible set of inputs. The state **A** and **D** go to next states **D** and **C** and have outputs 0 and 1 for x=0 and x=1 respectively. Therefore state **D** can be removed and replaced by **A**. Similarly, **C** and **F** generate exactly same next state and same output for every possible set of inputs. The state **C** and **F** go to next states **H** and **D** and have outputs 1 and 1 for x=0 and x=1 respectively. Therefore state **F** can be removed and replaced by **C**.

The reduced state table-1 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
A	A	C	0	1
B	E	A	1	1
C	H	A	1	1
E	B	G	0	1
G	A	C	0	1
H	C	A	0	1
I	G	H	1	1

Reduced state table-1

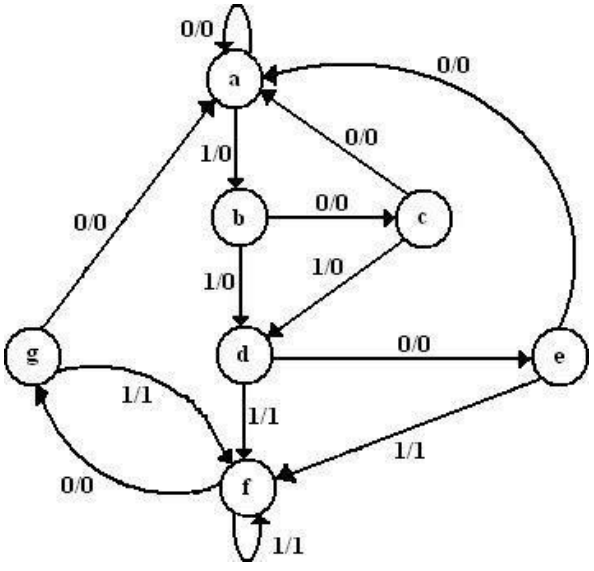
From the above reduced state table-1, **A** and **G** generate exactly same next state and same output for every possible set of inputs. The state **A** and **G** go to next states **A** and **C** and have outputs 0 and 1 for x=0 and x=1 respectively. Therefore state **G** can be removed and replaced by **A**. The final reduced state table-2 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
A	A	C	0	1
B	E	A	1	1
C	H	A	1	1
E	B	A	0	1
H	C	A	0	1
I	A	H	1	1

Reduced state table-2

Thus 9 states are reduced into 6 states.

4. Reduce the following state diagram.



Soln:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

State table

From the above state table **e** and **g** generate exactly same next state and same output for every possible set of inputs. The state **e** and **g** go to next states **a** and **f** and have outputs 0 and 1 for x=0 and x=1 respectively. Therefore state **g** can be removed and replaced by **e**. The reduced state table-1 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	**e**	f	0	1

Reduced state table-1

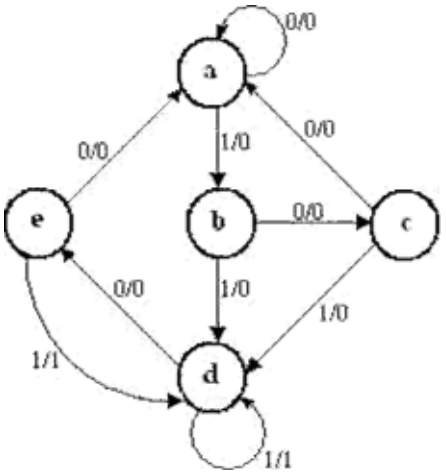
Now states d and f are equivalent. Both states go to the same next state (e, f) and have same output (0, 1). Therefore one state can be removed; **f** is replaced by **d.** The final reduced state table-2 is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	d	0	0
c	a	**d**	0	0
d	e	**d**	0	1
e	a	**d**	0	1

Reduced state table-2

Thus 7 states are reduced into 5 states.

The state diagram for the reduced state table-2 is,



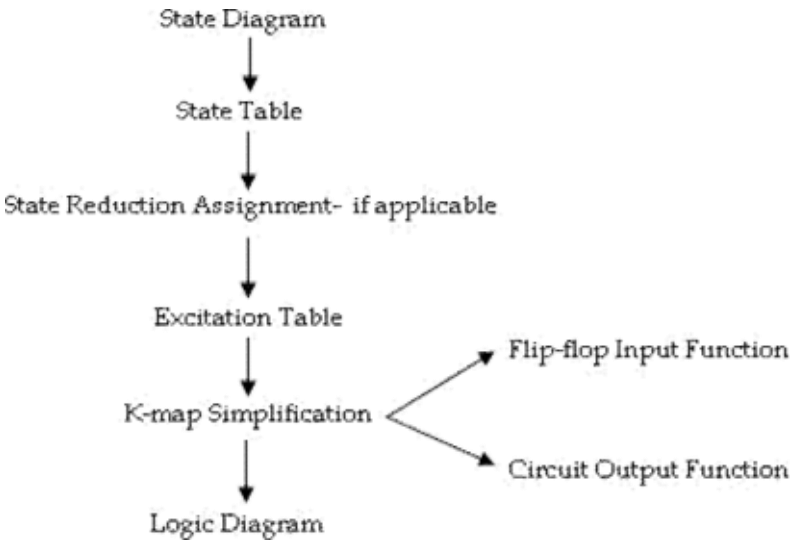
Reduced state diagram

3.11 DESIGN OF SYNCHRONOUS SEQUENTIAL CIRCUITS:

A synchronous sequential circuit is made up of number of Flip-Flops and combinational gates. The design of circuit consists of choosing the Flip-Flops and then finding a combinational gate structure together with the Flip-Flops. The number of Flip-Flops is determined from the number of states needed in the circuit. The combinational circuit is derived from the state table.

3.11.1 Design procedure:

1. The given problem is determined with a state diagram.
2. From the state diagram, obtain the state table.
3. The number of states may be reduced by state reduction methods (if applicable).
4. Assign binary values to each state (Binary Assignment) if the state table contains letter symbols.
5. Determine the number of Flip-Flops and assign a letter symbol (A, B, C,…) to each.
6. Choose the type of Flip-Flop (SR, JK, D, T) to be used.
7. From the state table, circuit excitation and output tables.
8. Using K-map or any other simplification method, derive the circuit output functions and the Flip-Flop input functions.
9. Draw the logic diagram.



The type of Flip-Flop to be used may be included in the design specifications or may depend what is available to the designer. Many digital systems are constructed with JK Flip-Flops because they are the most versatile available. The selection of inputs is given as follows.

Flip-Flop	Application
JK	General Applications
D	Applications requiring transfer of data
T	(Ex: Shift Registers) Application involving complementation (Ex: Binary Counters)

3.11.2 Excitation Tables:

Before going to the design examples for the clocked synchronous sequential circuits we revise Flip-Flop excitation tables.

Present State	Next	Inputs	
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 for SR Flip-Flop

Present State	Next State	Inputs	
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 for JK Flip-Flop

Present State	Next State	Input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

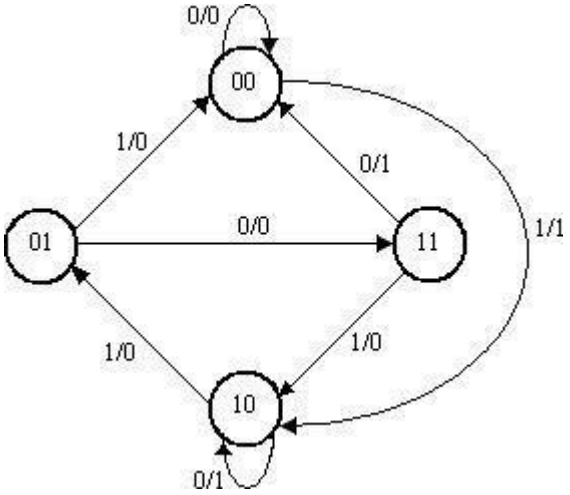
Excitation table for T Flip-Flop

Present State	Next State	Input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table for D Flip-Flop

3.11.3 Problems

1. A sequential circuit has one input and one output. The state diagram is shown below. Design the sequential circuit with a) D-Flip-Flops, b) T Flip-Flops, c) RS Flip-Flops and d) JK Flip-Flops.



Solution:

State Table:

The state table for the state diagram is,

Present state		Next state		Output	
		X= 0	X= 1	X= 0	X= 1
A	B	AB	AB	Y	Y
0	0	00	10	0	1
0	1	11	00	0	0
1	0	10	01	1	0
1	1	00	10	1	0

State reduction:

As seen from the state table there is no equivalent states. Therefore, no reduction in the state diagram.

The state table shows that circuit goes through four states, therefore we require 2 Flip-Flops (number of states= 2^m, where m= number of Flip-Flops). Since two Flip-Flops are required first is denoted as A and second is denoted as B.

i) Design using D Flip-Flops:

Excitation table:

Using the excitation table for T Flip-Flop, we can determine the excitation table for the

given circuit as,

Present State	Next State	Input
Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

Excitation table for D Flip-Flop

Present state		Input	Next state		Flip-Flop Inputs		Output
A	B	X	A	B	D_A	D_B	Y
0	0	0	0	0	0	0	0
0	0	1	1	0	1	0	1
0	1	0	1	1	1	1	0
0	1	1	0	0	0	0	0
1	0	0	1	0	1	0	1
1	0	1	0	1	0	1	0
1	1	0	0	0	0	0	1
1	1	1	1	0	1	0	0

Circuit excitation table

K-map Simplification:

For Flip-flop A

A \ BX	00	01	11	10
0	0	(1)	0	(1)
1	(1)	0	(1)	0

$D_A = A'B'X + A'BX' + ABX + AB'X'$

$= A \oplus (B \oplus x)$

For Flip-flop B

A \ BX	00	01	11	10
0	0	0	0	(1)
1	0	(1)	0	0

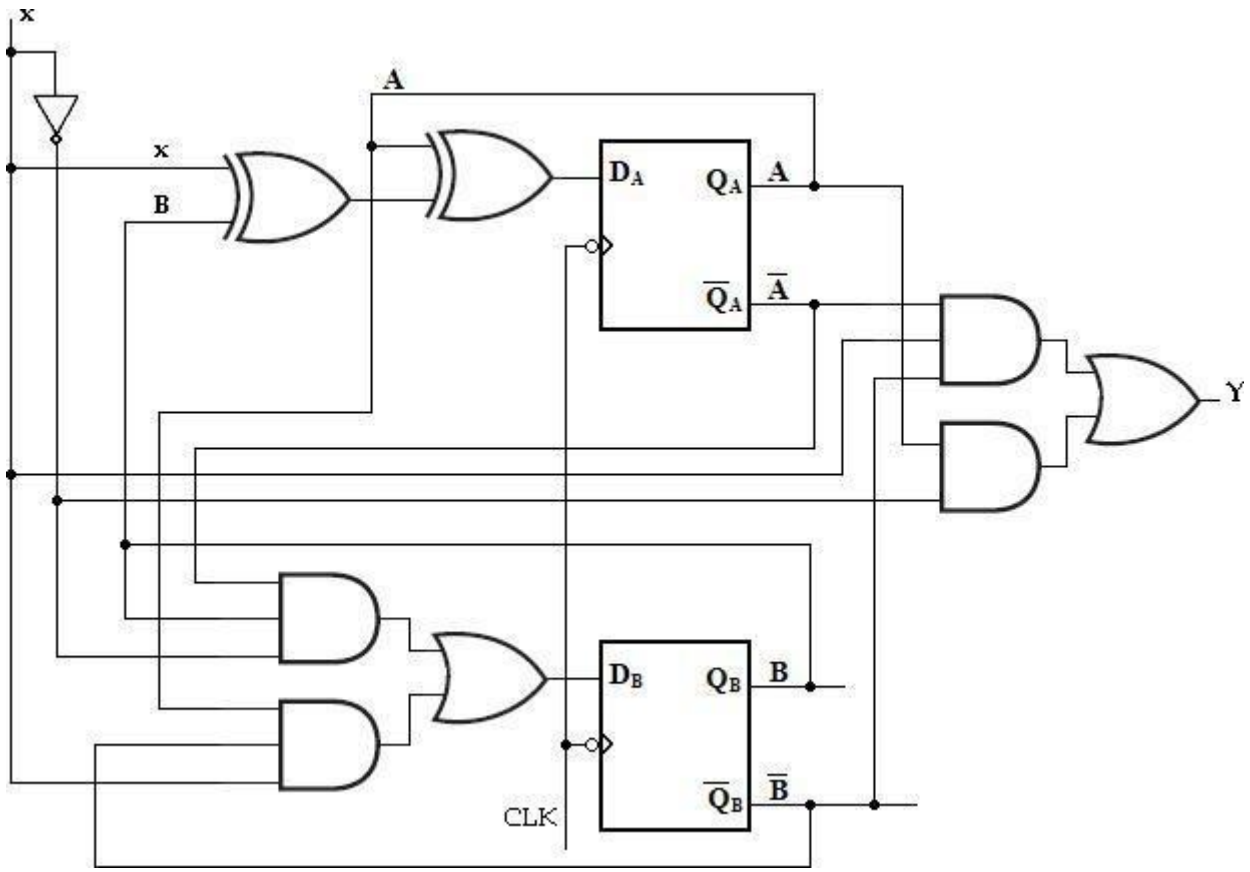
$D_B = A'BX' + AB'X$

For Output

A \ BX	00	01	11	10
0	0	(1)	0	0
1	(1)	0	0	(1)

$Y = A'B'X + AX'$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



Logic diagram of given sequential circuit using D Flip-Flop

ii) Design using T Flip-Flops:

Using the excitation table for T Flip-Flop, we can determine the excitation table for the given circuit as,

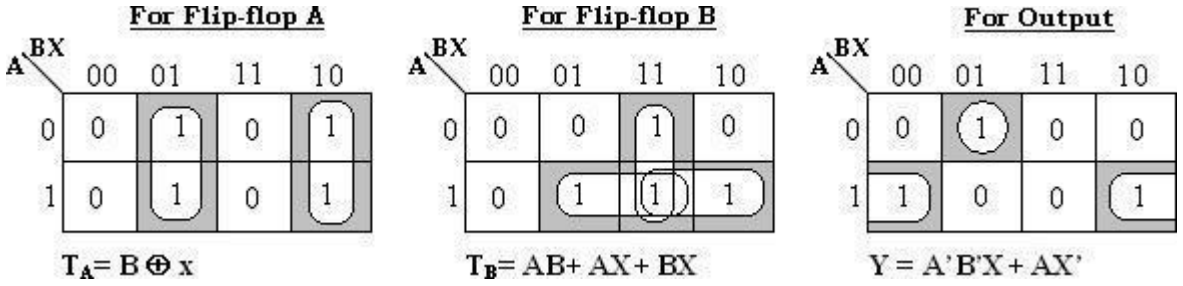
Present State	Next State	Input
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Excitation table for T Flip-Flop

Present state		Input	Next state		Flip-Flop Inputs		Output
A	B	X	A	B	T_A	T_B	Y
0	0	0	0	0	0	0	0
0	0	1	1	0	1	0	1
0	1	0	1	1	1	0	0
0	1	1	0	0	0	1	0
1	0	0	1	0	0	0	1
1	0	1	0	1	1	1	0
1	1	0	0	0	1	1	1
1	1	1		0	0	1	0

Circuit excitation table

K-map Simplification:



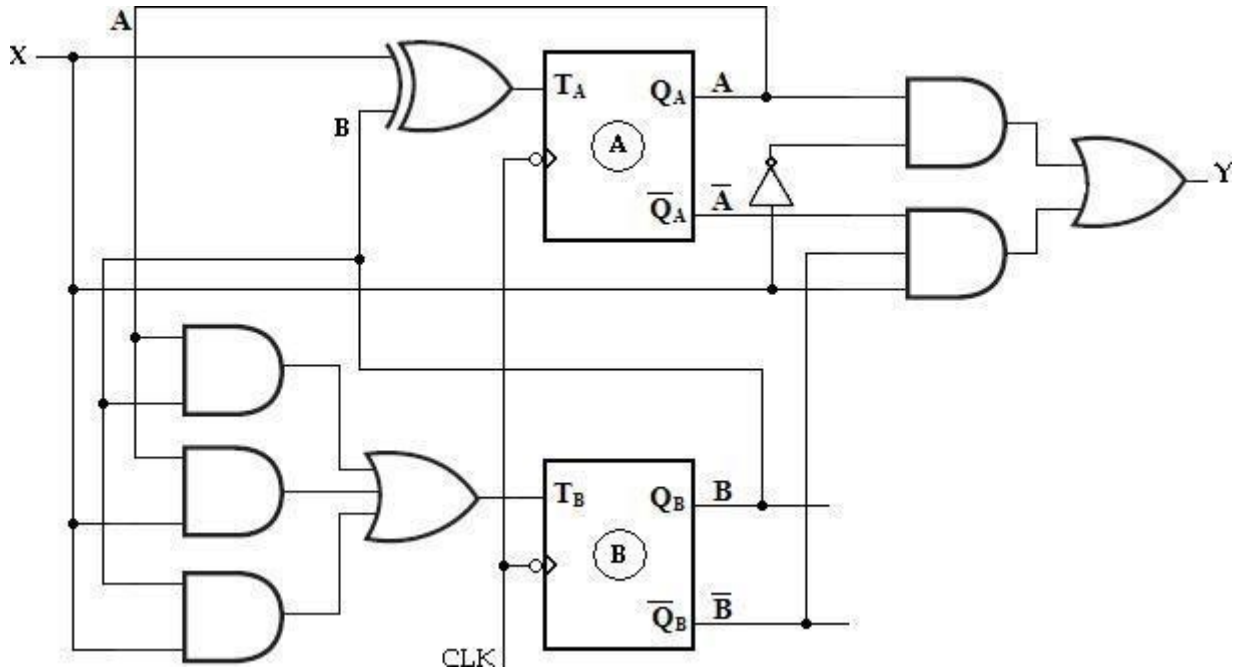
Therefore, input functions for,

T_A= B ⊕ x and

T_B= AB+ AX+ BX

Circuit output function, **Y = XA'B'+ X'A**

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



Logic diagram of given sequential circuit using T Flip-Flop

iii) Design using SR Flip-Flops:

Using the excitation table for RS Flip-Flop, we can determine the excitation table for the given circuit as,

Present State	Next State	Inputs	
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 for SR Flip-Flop

Present state		Input	Next state		Flip-Flop Inputs				Output
A	B	X	A	B	S_A	R_A	S_B	R_B	Y
0	0	0	0	0	0	x	0	x	0
0	0	1	1	0	1	0	0	x	1
0	1	0	1	1	1	0	x	0	0
0	1	1	0	0	0	x	0	1	0
1	0	0	1	0	x	0	0	x	1
1	0	1	0	1	0	1	1	0	0
1	1	0	0	0	0	1	0	1	1
1	1	1	1	0	x	0	0	1	0

Circuit excitation table

K-map Simplification:

For Flip-flop A

For S_A

A \ BX	00	01	11	10
0	0	1	0	1
1	x	0	x	0

$S_A = A'B'X + A'BX'$
$= A'(B \oplus X)$

For R_A

A \ BX	00	01	11	10
0	x	0	x	0
1	0	1	0	1

$R_A = ABX' + AB'X'$

For Output

A \ BX	00	01	11	10
0	0	1	0	0
1	1	0	0	1

$Y = A'B'X + AX'$

For Flip-flop B

For S_B

A \ BX	00	01	11	10
0	0	0	0	x
1	0	1	0	0

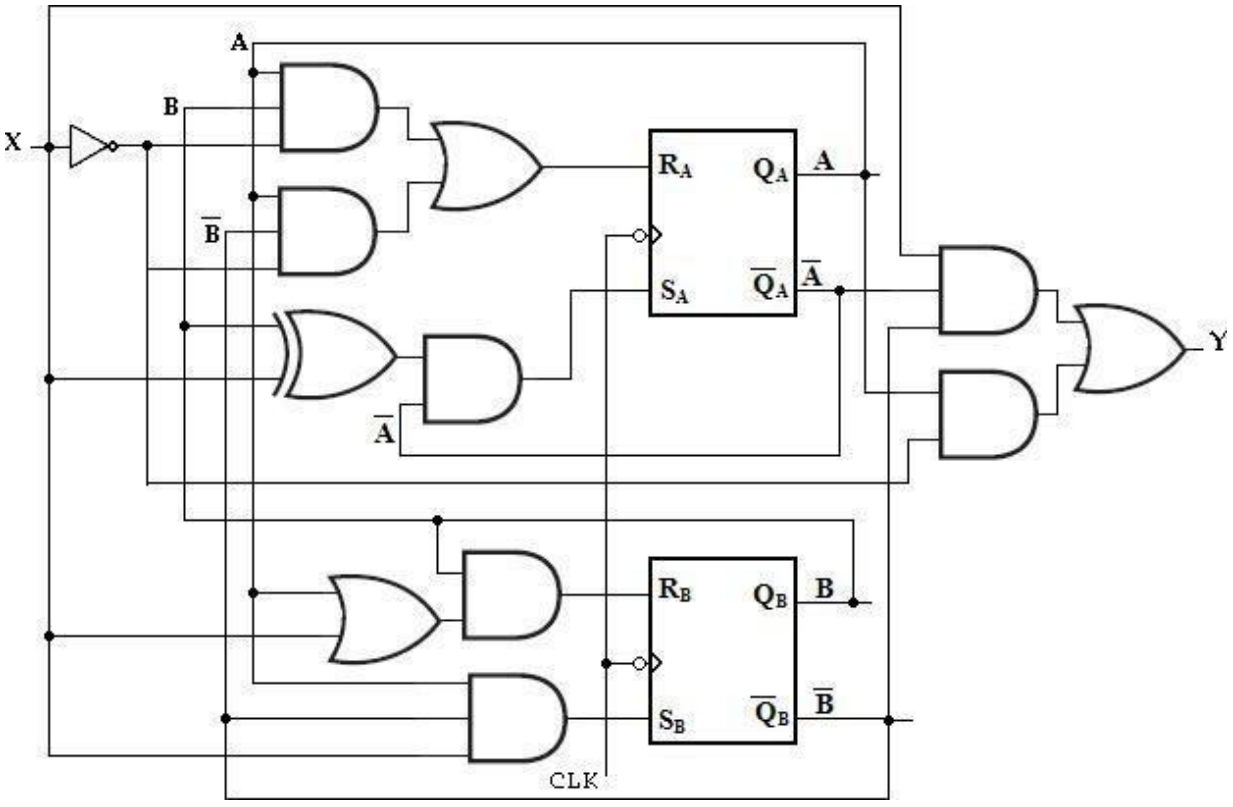
$S_B = AB'X$

For R_B

A \ BX	00	01	11	10
0	x	x	1	0
1	x	0	1	1

$R_B = AB + BX$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



iii) Design using JK Flip-Flops:

Using the excitation table for JK Flip-Flop, we can determine the excitation table for the given circuit as,

Present State	Next State	Inputs	
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 for JK Flip-Flop

Present state		Input	Next state		Flip-Flop Inputs				Output
A	B	X	A	B	J_A	K_A	J_B	K_B	Y
0	0	0	0	0	0	x	0	x	0
0	0	1	1	0	1	x	0	x	1
0	1	0	1	1	1	x	x	0	0
0	1	1	0	0	0	x	x	1	0
1	0	0	1	0	x	0	0	x	1
1	0	1	0	1	x	1	1	x	0
1	1	0	0	0	x	1	x	1	1
1	1	1	1	0	x	0	x	1	0

Circuit excitation table

K-map Simplification:

For Flip-flop A

For J_A

A \ BX	00	01	11	10
0	0	1	0	1
1	x	x	x	x

$J_A = BX' + B'X$
$= B \oplus X$

For K_A

A \ BX	00	01	11	10
0	x	x	x	x
1	0	1	0	1

$K_A = BX' + B'X$
$= B \oplus X$

For Output

A \ BX	00	01	11	10
0	0	1	0	0
1	1	0	0	1

$Y = A'B'X + AX'$

For Flip-flop B

For J_B

A \ BX	00	01	11	10
0	0	0	x	x
1	0	1	x	x

$J_B = AX$

For K_B

A \ BX	00	01	11	10
0	x	x	1	0
1	x	x	1	1

$K_B = A + X$

The input functions for,

$J_A = BX' + B'X$ **$J_B = AX$**

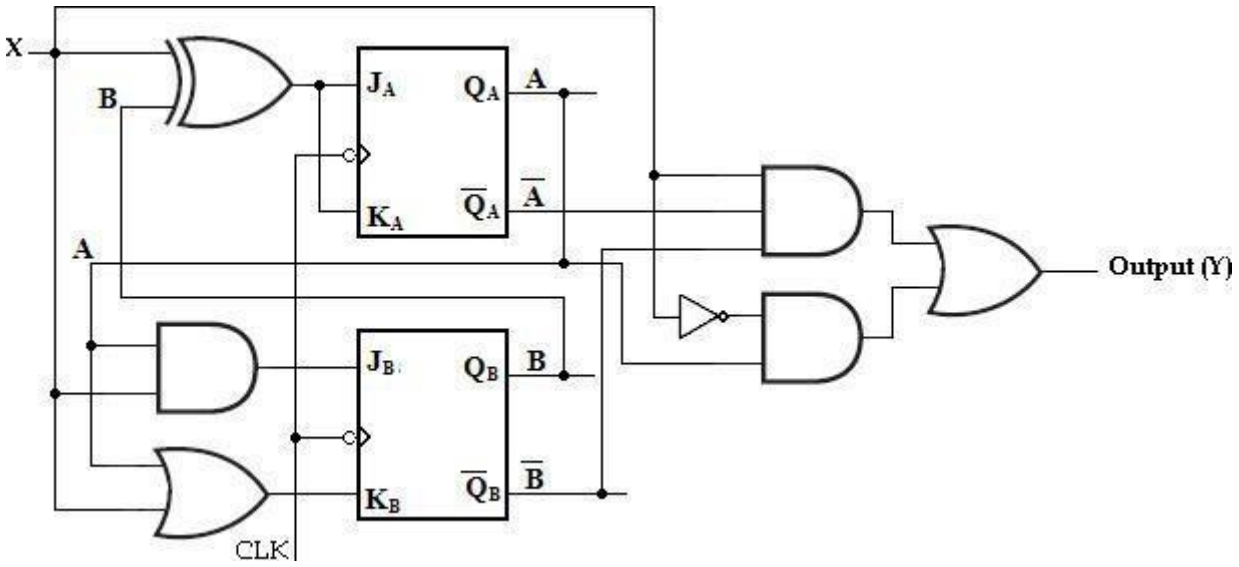
$= B \oplus X$

$K_A = BX' + B'X$ **$K_B = A + X$**

$= B \oplus X$

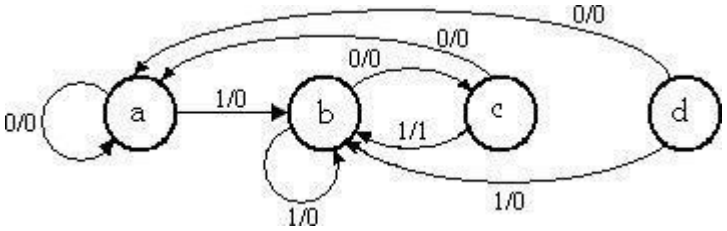
Circuit output function, **$Y = AX' + A'B'X$**

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



Logic diagram of given sequential circuit using JK Flip-Flop

2. Design a clocked sequential machine using JK Flip-Flops for the state diagram shown in the figure. Use state reduction if possible. Make proper state assignment.



Soln:

State Table:

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	b	0	0
c	a	b	0	1
d	a	b	0	0

From the above state table **a** and **d** generate exactly same next state and same output for every possible set of inputs. The state **a** and **d** go to next states **a** and **b** and have outputs 0 and 0 for x=0 and x=1 respectively. Therefore state **d** can be removed and replaced by **a**. The final reduced state table is shown below.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	c	b	0	0
c	a	b	0	1

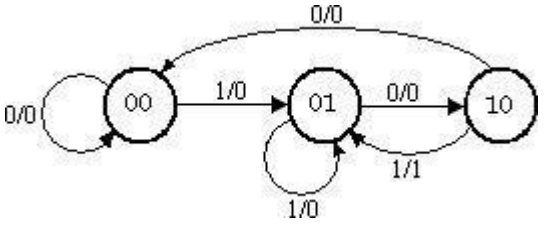
Reduced State table

Binary Assignment:

Now each state is assigned with binary values. Since there are three states, number of Flip-Flops required is two and 2 binary numbers are assigned to the states.

a= 00; b= 0; and c= 10

The reduced state diagram is drawn as,



Reduced State Diagram

Excitation Table:

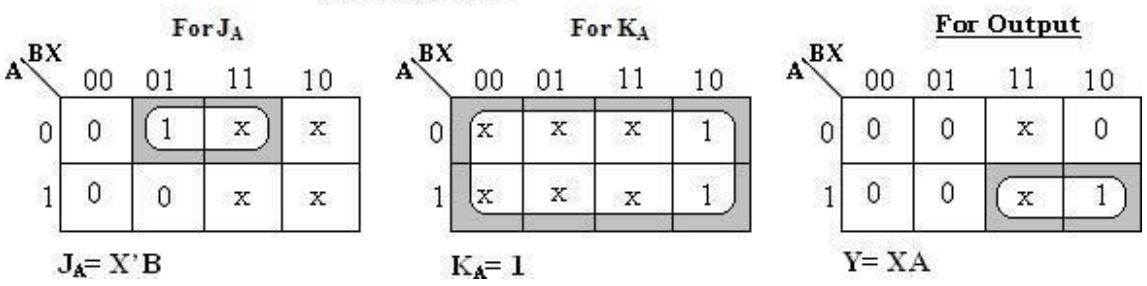
Present State	Next State	Inputs	
Q_n	Q_{n+1}	J	K
0	0	0	x
0	1	1	x
1	0	x	1
1	1	x	0

Excitation table for JK Flip-Flop

Input	Present state		Next state		Flip-Flop Inputs				Output
X	A	B	A	B	J_A	K_A	J_B	K_B	Y
0	0	0	0	0	0	x	0	x	0
1	0	0	0	1	0	x	1	x	0
0	0	1	1	0	1	x	x	1	0
1	0	1	0	1	0	x	x	0	0
0	1	0	0	0	x	1	0	x	0
1	1	0	0	1	x	1	1	x	1
0	1	1	x	x	x	x	x	x	x
1	1	1	x	x	x	x	x	x	x

K-map Simplification:

For Flip-flop A



For Flip-flop B

For J_B

A \ BX	00	01	11	10
0	0	x	x	0
1	1	x	x	1

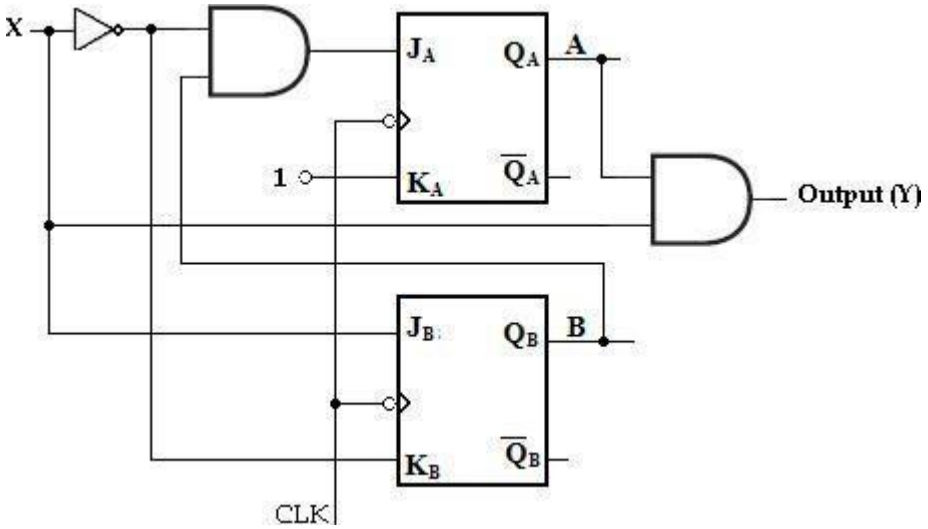
$J_B = X$

For K_B

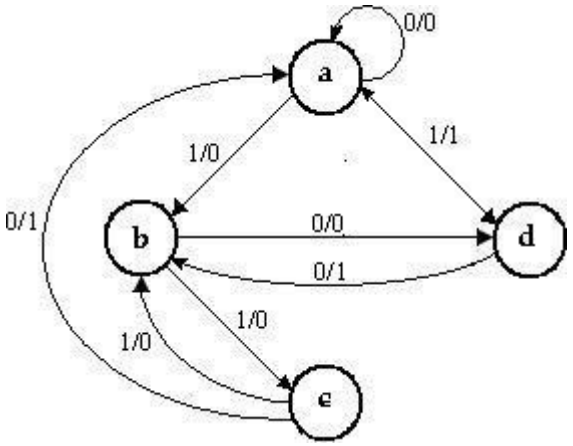
A \ BX	00	01	11	10
0	x	1	x	x
1	x	0	x	x

$K_B = X'$

With these Flip-Flop input functions and circuit output function we can draw the logic diagram as follows.



3. Design a clocked sequential machine using T Flip-Flops for the following state diagram. Use state reduction if possible. Also use straight binary state assignment.



Soln:

State Table:

State table for the given state diagram is,

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
a	a	b	0	0
b	d	c	0	0
c	a	b	1	0
d	b	a	1	1

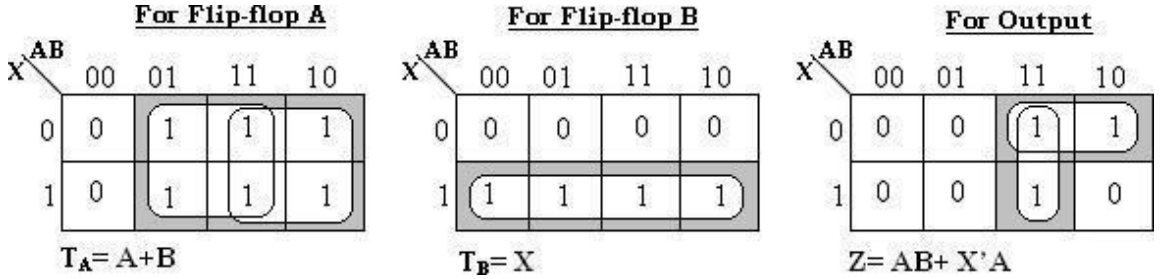
Even though a and c are having same next states for input X=0 and X=1, as the outputs are not same state reduction is not possible.

State Assignment:

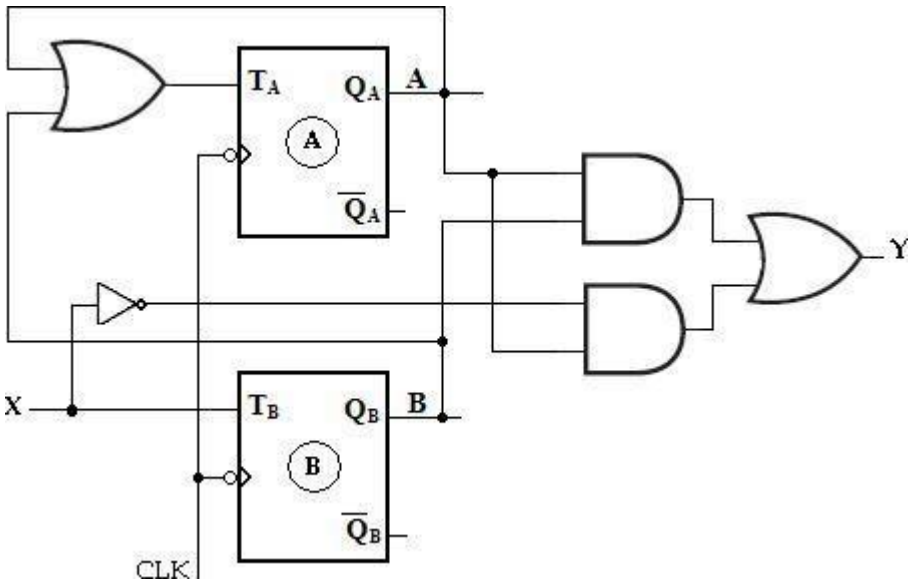
Use straight binary assignments as a= 00, b= 01, c= 10 and d= 11, the transition table is,

Input	Present state		Next state		Flip-Flop Inputs		Output
X	A	B	A	B	T_A	T_B	Y
0	0	0	0	0	0	0	0
0	0	1	1	1	1	0	0
0	1	0	0	0	1	0	1
0	1	1	0	1	1	0	1
1	0	0	0	1	0	1	0
1	0	1	1	0	1	1	0
1	1	0	0	1	1	1	0
1	1	1	0	0	1	1	1

K-map simplification:



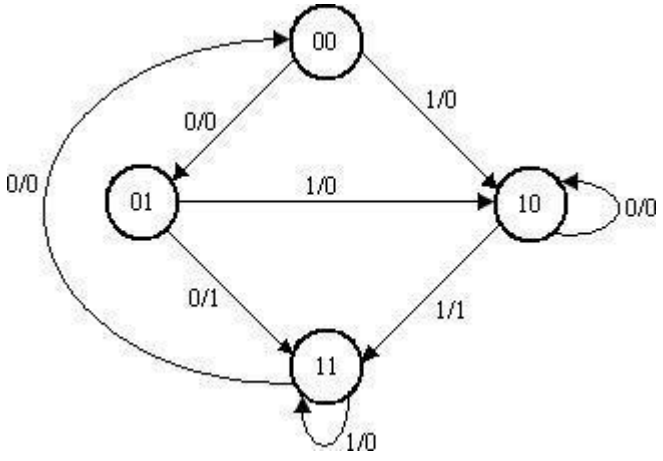
Logic Diagram:



3.12 STATE ASSIGNMENT:

In sequential circuits, the behavior of the circuit is defined in terms of its inputs, present states, next states and outputs. To generate desired next state at particular present state and inputs, it is necessary to have specific Flip-Flop inputs. These Flip-Flop inputs are described by a set of Boolean functions called Flip-Flopinput functions.

To determine the Flip-Flop functions, it is necessary to represent states in the state diagram using binary values instead of alphabets. This procedure is known as ***state assignment***.



Reduced state diagram with binary states

3.15.1 Rules for state assignments

There are two basic rules for making state assignments.

Rule 1:

States having the **same** NEXT STATES for a given input condition should have assignments which can be grouped into logically adjacent cells in a K-map.

Rule 2:

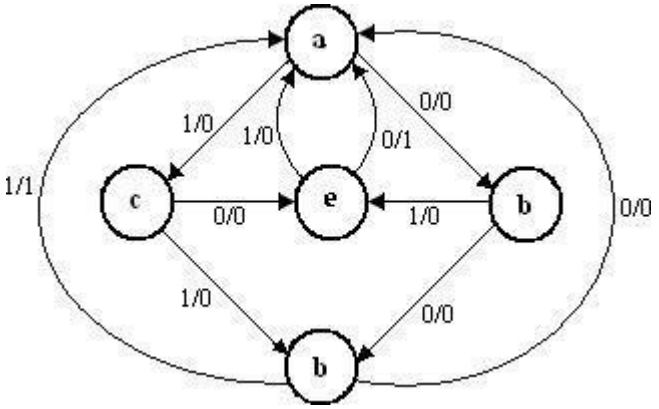
States that are the NEXT STATES of a single state should have assignment which can be grouped into logically adjacent cells in a K-map.

Present state	Next state		Output	
	X= 0	X= 1	X= 0	X= 1
00	01	10	0	0
01	11	10	1	0
10	10	11	0	1
11	00	11	0	0

State table with assignment states

3.15.2 State Assignment Problem:

1. Design a sequential circuit for a state diagram shown below. Use state assignment rules for assigning states and compare the required combinational circuit with random state assignment.



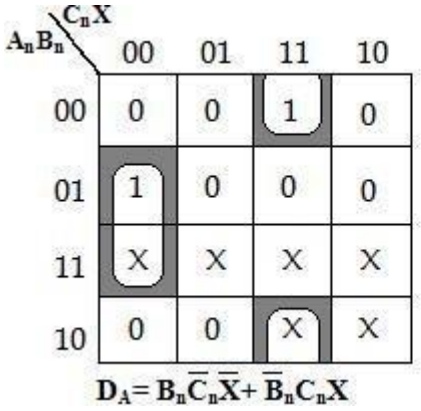
Using random state assignment we assign,

a= 000, b= 001, c= 010, d= 011 and e= 100.

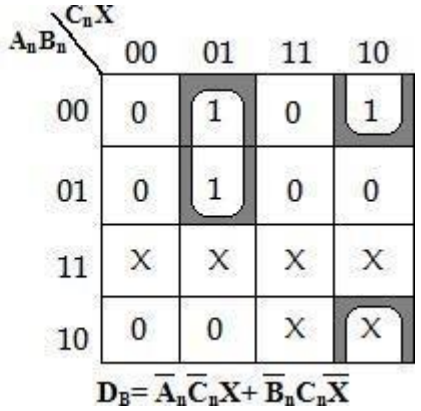
The excitation table with these assignments is given as,

Present state			Input	Next state			Output
A_n	B_n	C_n	X	A_{n+1}	B_{n+1}	C_{n+1}	Z
0	0	0	0	0	0	1	0
0	0	0	1	0	1	0	0
0	0	1	0	0	1	1	0
0	0	1	1	1	0	0	0
0	1	0	0	1	0	0	0
0	1	0	1	0	1	1	0
0	1	1	0	0	0	0	0
0	1	1	1	0	0	0	1
1	0	0	0	0	0	0	1
1	0	0	1	0	0	0	0
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	x	x	x	x
1	1	1	1	x	x	x	x

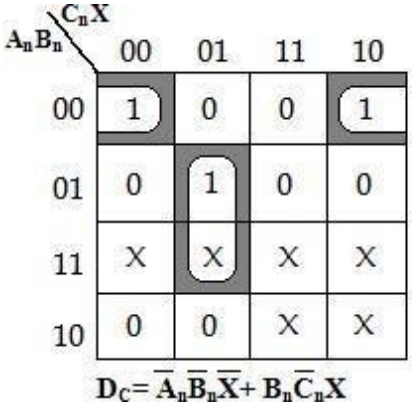
K-map Simplification:



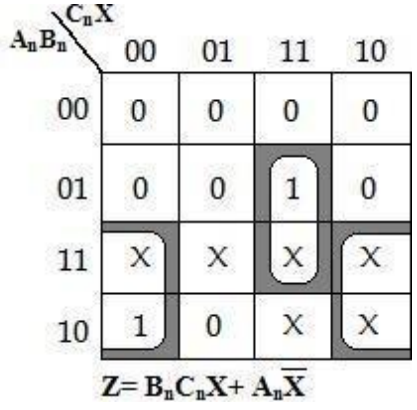
$D_A = B_n\overline{C}_n\overline{X} + \overline{B}_nC_nX$



$D_B = \overline{A}_n\overline{C}_nX + \overline{B}_nC_n\overline{X}$



$D_C = \overline{A}_n\overline{B}_n\overline{X} + B_n\overline{C}_nX$



$Z = B_nC_nX + A_n\overline{X}$

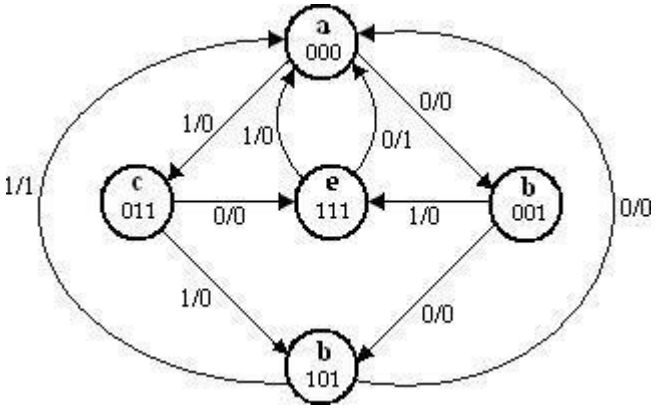
The random assignments require:

7 three input AND functions
1 two input AND function
4 two input OR functions

---

12 gates with 31 inputs

Now, we will apply the state assignment rules and compare the results.



State diagram after applying Rules 1 and 2

Rule 1 says that: e and d must be adjacent, and

b and c must be adjacent.

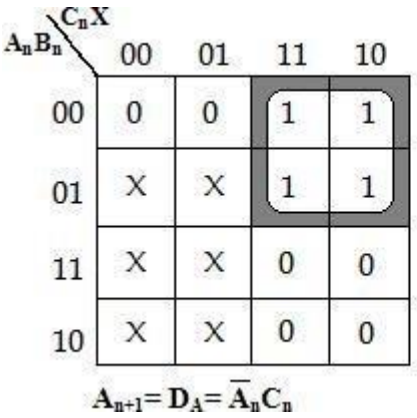
Rule 2 says that: e and d must be adjacent, and

b and c must be adjacent.

Applying Rule 1, Rule 2 to the state diagram we get the state assignment as,

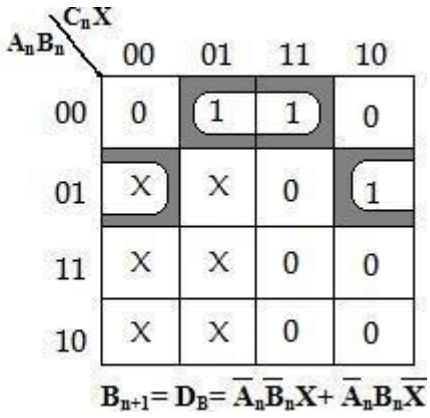
Present state			Input	Next state			Output
A_n	B_n	C_n	X	A_{n+1}	B_{n+1}	C_{n+1}	Z
0	0	0	0	0	0	1	0
0	0	0	1	0	1	1	0
0	0	1	0	1	0	1	0
0	0	1	1	1	1	1	0
0	1	0	0	x	x	x	x
0	1	0	1	x	x	x	x
0	1	1	0	1	1	1	0
0	1	1	1	1	0	1	0
1	0	0	0	x	x	x	x
1	0	0	1	x	x	x	x
1	0	1	0	0	0	0	0
1	0	1	1	0	0	0	1
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x
1	1	1	0	0	0	0	1
1	1	1	1	0	0	0	0

K-map Simplification:



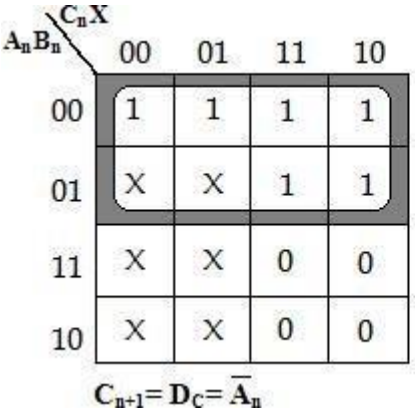
A_nB_n \ C_nX	00	01	11	10
00	0	0	1	1
01	X	X	1	1
11	X	X	0	0
10	X	X	0	0

$A_{n+1} = D_A = \overline{A}_n C_n$



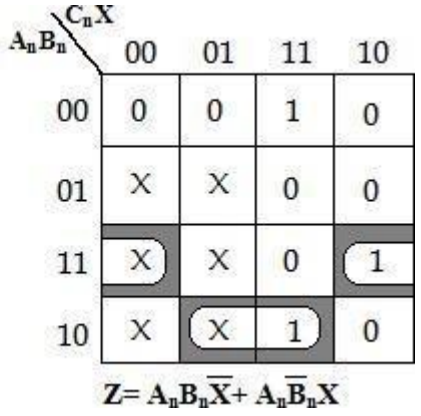
A_nB_n \ C_nX	00	01	11	10
00	0	1	1	0
01	X	X	0	1
11	X	X	0	0
10	X	X	0	0

$B_{n+1} = D_B = \overline{A}_n \overline{B}_n X + \overline{A}_n B_n \overline{X}$



A_nB_n \ C_nX	00	01	11	10
00	1	1	1	1
01	X	X	1	1
11	X	X	0	0
10	X	X	0	0

$C_{n+1} = D_C = \overline{A}_n$



A_nB_n \ C_nX	00	01	11	10
00	0	0	1	0
01	X	X	0	0
11	X	X	0	1
10	X	X	1	0

$Z = A_n B_n \overline{X} + A_n \overline{B}_n X$

The state assignments using Rule 1 and 2 require:

4 three input AND functions

1 two input AND function

2 two input OR functions

---

7 gates with 18 inputs

Thus by simply applying Rules 1 and 2 good results have been achieved.

3.14 SYNCHRONOUS COUNTERS

Flip-Flops can be connected together to perform counting operations. Such a group of Flip- Flops is a **counter**. The number of Flip-Flops used and the way in which they are connected determine the number of states (called the modulus) and also the specific sequence of states that the counter goes through during each complete cycle.

Counters are classified into two broad categories according to the way they are clocked:

- Asynchronous counters,
- Synchronous counters.

In asynchronous (ripple) counters, the first Flip-Flop is clocked by the external clock pulse and then each successive Flip-Flop is clocked by the output of the preceding Flip-Flop.

In synchronous counters, the clock input is connected to all of the Flip-Flops so that they are clocked simultaneously. Within each of these two categories, counters are classified primarily by the type of sequence, the number of states, or the number of Flip-Flops in the counter.

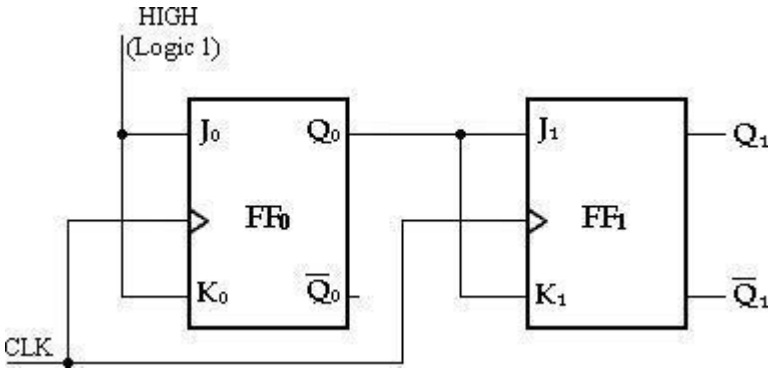
The term 'synchronous' refers to events that have a fixed time relationship with each other. In synchronous counter, the clock pulses are applied to all Flip-Flops simultaneously. Hence there is minimum propagation delay.

S.No	Asynchronous (ripple) counter	Synchronous counter
1	All the Flip-Flops are not clocked simultaneously.	All the Flip-Flops are clocked simultaneously.
2	The delay times of all Flip-Flops are added. Therefore there is considerable propagation delay.	There is minimum propagation delay.
3	Speed of operation is low	Speed of operation is high.
4	Logic circuit is very simple	Design involves complex logic circuit

	even for more number of states.	as number of state increases.
5	Minimum numbers of logic devices are needed.	The number of logic devices is more than ripple counters.
6	Cheaper than synchronous counters.	Costlier than ripple counters.

3.14.1 2-Bit Synchronous Binary Counter

In this counter the clock signal is connected in parallel to clock inputs of both the Flip-Flops (FF_0 and FF_1). The output of FF_0 is connected to J_1 and K_1 inputs of the second Flip-Flop (FF_1).



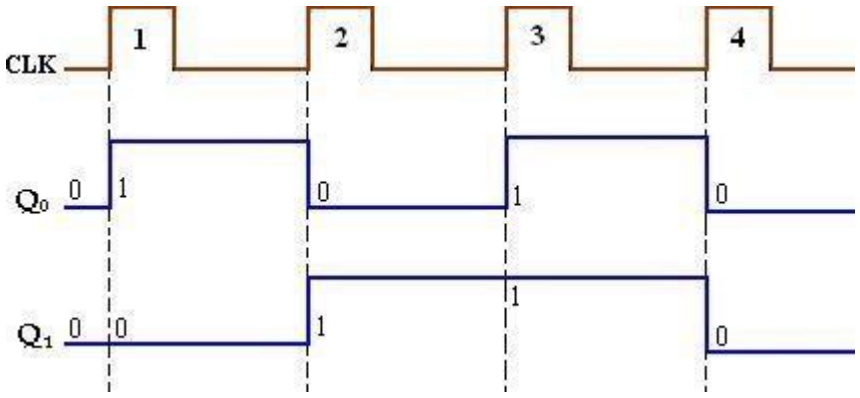
2-Bit Synchronous Binary Counter

Assume that the counter is initially in the binary 0 state: i.e., both Flip-Flops are RESET. When the positive edge of the first clock pulse is applied, FF_0 will toggle because $J_0 = k_0 = 1$, whereas FF_1 output will remain 0 because $J_1 = k_1 = 0$. After the first clock pulse $Q_0=1$ and $Q_1=0$.

When the leading edge of CLK2 occurs, FF_0 will toggle and Q_0 will go LOW. Since FF_1 has a HIGH ($Q_0 = 1$) on its J_1 and K_1 inputs at the triggering edge of this clock pulse, the Flip-Flop toggles and Q_1 goes HIGH. Thus, after CLK2, $Q_0 = 0$ and $Q_1 = 1$.

When the leading edge of CLK3 occurs, FF_0 again toggles to the SET state ($Q_0 = 1$), and FF_1 remains SET ($Q_1 = 1$) because its J_1 and K_1 inputs are both LOW ($Q_0 = 0$). After this triggering edge, $Q_0 = 1$ and $Q_1 = 1$.

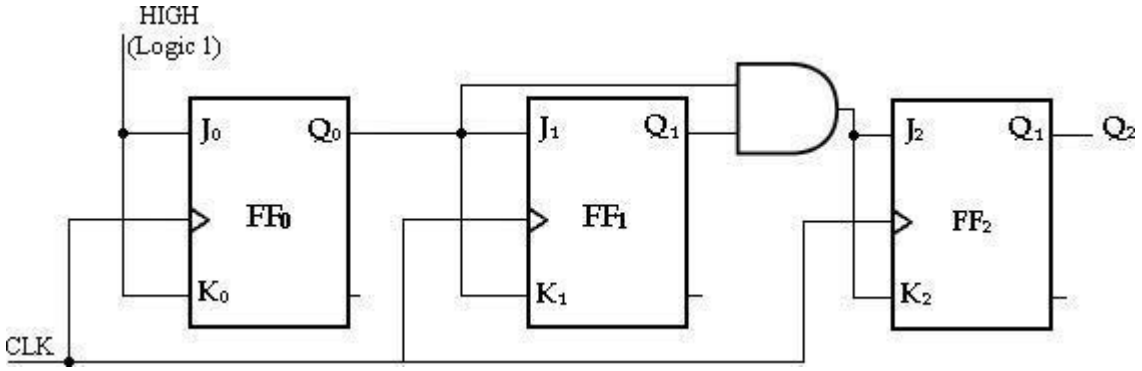
Finally, at the leading edge of CLK4, Q_0 and Q_1 go LOW because they both have a toggle condition on their J_1 and K_1 inputs. The counter has now recycled to its original state, $Q_0 = Q_1 = 0$.



Timing diagram

3.14.2 3-Bit Synchronous Binary Counter

A 3 bit synchronous binary counter is constructed with three JK Flip-Flops and an AND gate. The output of FF_0 (Q_0) changes on each clock pulse as the counter progresses from its original state to its final state and then back to its original state. To produce this operation, FF_0 must be held in the toggle mode by constant HIGH, on its J_0 and K_0 inputs.



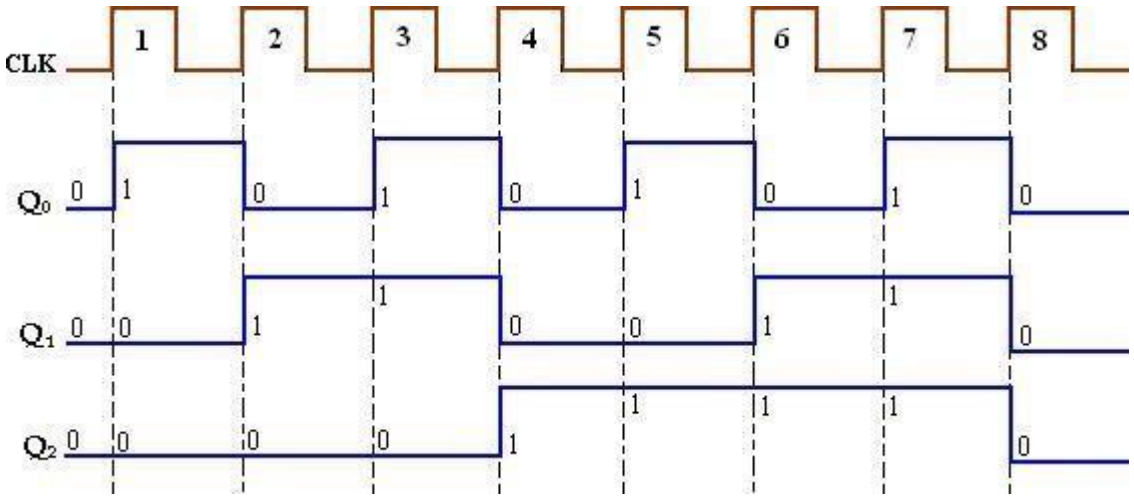
3-Bit Synchronous Binary Counter

The output of FF_1 (Q_1) goes to the opposite state following each time $Q_0 = 1$. This change occurs at CLK2, CLK4, CLK6, and CLK8. The CLK8 pulse causes the counter to recycle. To produce this operation, Q_0 is connected to the J_1 and K_1 inputs of FF_1. When $Q_0 = 1$ and a clock pulse occurs, FF_1 is in the toggle mode and therefore changes state. When $Q_0 = 0$, FF_1 is in the no-change mode and remains in its present state.

The output of FF_2 (Q_2) changes state both times; it is preceded by the unique condition in which both Q_0 and Q_1 are HIGH. This condition is detected by the AND gate and applied to the J_2 and K_2 inputs of FF_3. Whenever both outputs $Q_0 = Q_1 = 1$,

the output of the AND gate makes the $J_2 = K_2 = 1$ and FF_2 toggles on the following clock pulse. Otherwise, the J_2 and K_2 inputs of FF2 are held LOW by the AND gate output, FF_2 does not change state.

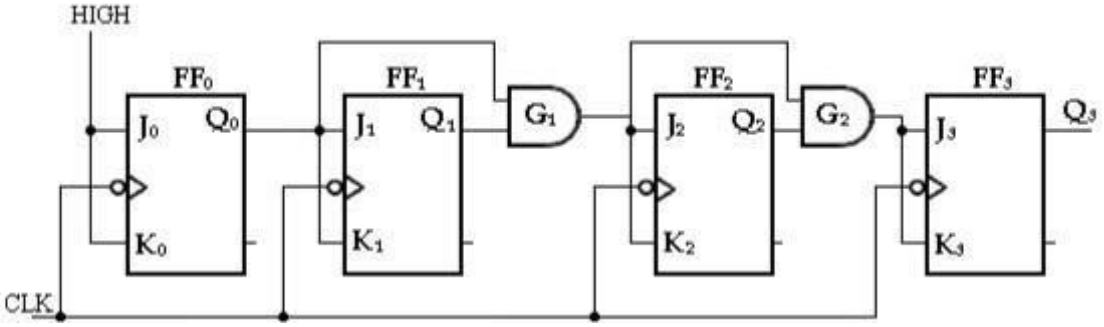
CLOCK Pulse	Q_2	Q_1	Q_0
Initially	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1
8 (recycles)	0	0	0



Timing diagram

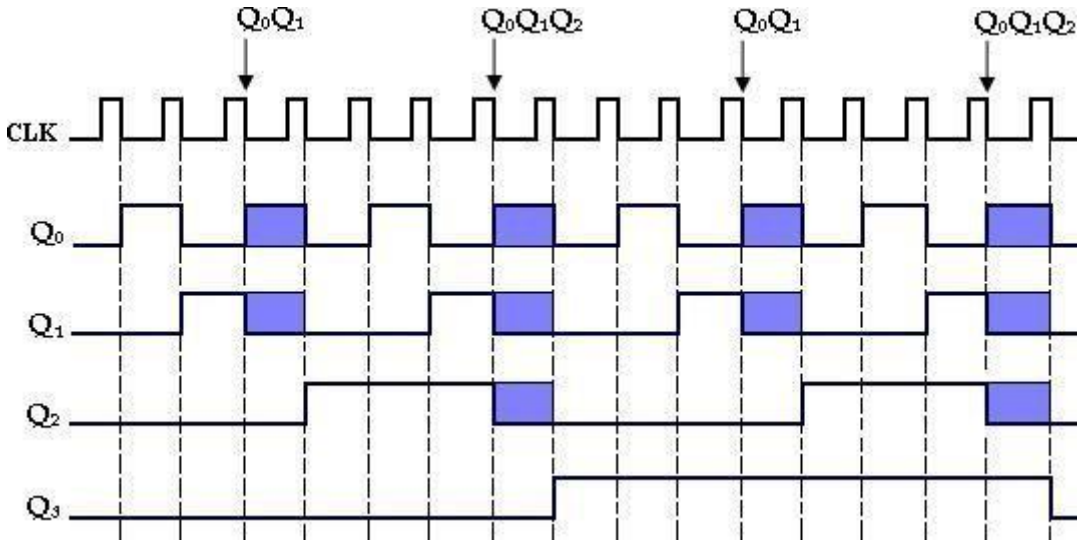
3.14.3 4-Bit Synchronous Binary Counter

This particular counter is implemented with negative edge-triggered Flip-Flops. The reasoning behind the J and K input control for the first three Flip- Flops is the same as previously discussed for the 3-bit counter. For the fourth stage, the Flip-Flop has to change the state when $Q_0 = Q_1 = Q_2 = 1$. This condition is decoded by AND gate G_3.



4-Bit Synchronous Binary Counter

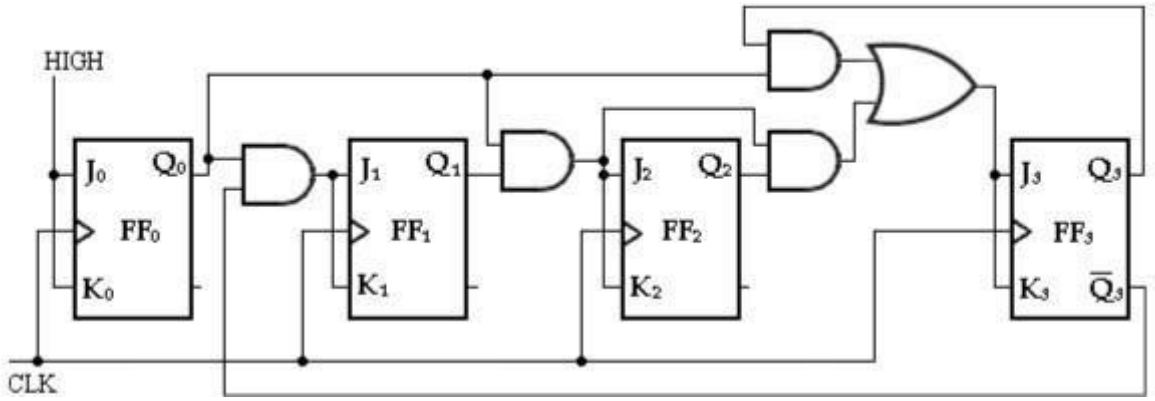
Therefore, when $Q_0 = Q_1 = Q_2 = 1$, Flip-Flop FF_3 toggles and for all other times it is in a no-change condition. Points where the AND gate outputs are HIGH are indicated by the shaded areas.



Timing diagram

3.14.4 4-Bit Synchronous Decade Counter: (BCD Counter):

BCD decade counter has a sequence from 0000 to 1001 (9). After 1001 state it must recycle back to 0000 state. This counter requires four Flip-Flops and AND/OR logic as shown below.



4-Bit Synchronous Decade Counter

CLOCK Pulse	Q_3	Q_2	Q_1	Q_0
Initially	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10(recycles)	0	0	0	0

1. First, notice that FF_0 (Q_0) toggles on each clock pulse, so the logic equation for its J_0 and K_0 inputs is

$$\mathbf{J_0 = K_0 = 1}$$

This equation is implemented by connecting J_0 and K_0 to a constant HIGH level.

2. Next, notice from table, that FF_1 (Q_1) changes on the next clock pulse each time Q0 = 1 and Q3 = 0, so the logic equation for the J_1 and K_1 inputs is

$$\mathbf{J_1 = K_1 = Q_0 Q_3'}$$

This equation is implemented by ANDing Q_0 and Q_3 and connecting the gate output to the J_1 and K_1 inputs of FF_1.

3. Flip-Flop 2 (Q_2) changes on the next clock pulse each time both $Q_0 = Q_1 =$ 1. This requires an input logic equation as follows:

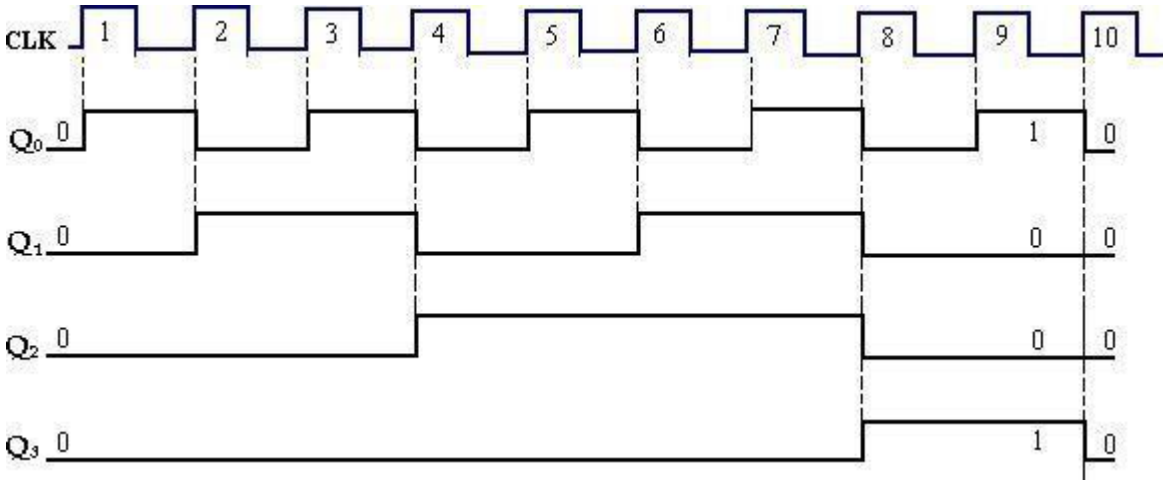
$$\mathbf{J_2 = K_2 = Q_0 Q_1}$$

This equation is implemented by ANDing Q_0 and Q_1 and connecting the gate output to the J_2 and K_2 inputs of FF_3.

4. Finally, FF_3 (Q_3) changes to the opposite state on the next clock pulse each time $Q_0 = 1$, $Q_1 = 1$, and $Q_2 = 1$ (state 7), or when $Q_0 = 1$ and $Q_1 = 1$ (state 9).The equation for this is as follows:

$$J_3 = K_3 = Q_0Q_1Q_2 + Q_0Q_3$$

This function is implemented with the AND/OR logic connected to the J_3 and K_3 inputs of FF_3.



Timing diagram

3.14.5 Synchronous UP/DOWN Counter

An up/down counter is a bidirectional counter, capable of progressing in either direction through a certain sequence. A 3-bit binary counter that advances upward through its sequence (0, 1, 2, 3, 4, 5, 6, 7) and then can be reversed so that it goes through the sequence in the opposite direction (7, 6, 5, 4, 3, 2, 1,0) is an illustration of up/down sequential operation.

The complete up/down sequence for a 3-bit binary counter is shown in table below. The arrows indicate the state-to-state movement of the counter for both its UP and its DOWN modes of operation. An examination of Q_0 for both the up and down sequences shows that FF_0 toggles on each clock pulse. Thus, the J_0 and K_0 inputs of FF_0 are,

$$J_0 = K_0 = 1$$

CLOCK PULSE	UP	Q_2	Q_1	Q_0	DOWN
0		0	0	0	
1		0	0	1	
2		0	1	0	
3		0	1	1	
4		1	0	0	
5		1	0	1	
6		1	1	0	
7		1	1	1	

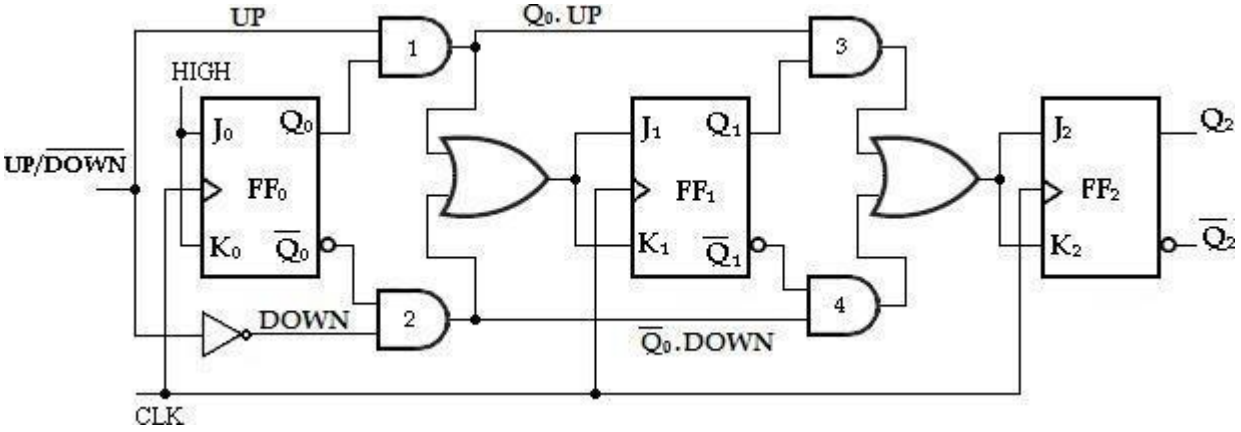
To form a synchronous UP/DOWN counter, the control input (UP/DOWN)is used to allow either the normal output or the inverted output of one Flip-Flop to the J and K inputs of the next Flip-Flop. When UP/DOWN= 1, the MOD 8 counter will count from 000 to 111 and UP/DOWN= 0, it will count from 111 to 000.

When UP/DOWN= 1, it will enable AND gates 1 and 3 and disable AND gates 2 and 4. This allows the Q0 and Q1 outputs through the AND gates to the J and K inputs of the following Flip-Flops, so the counter counts up as pulses are applied.

When UP/DOWN= 0, the reverse action takes place.

$J_1 = K_1 = (Q_0.UP) + (Q_0'.DOWN)$

$J_2 = K_2 = (Q_0.Q_1.UP) + (Q_0'.Q_1'.DOWN)$



3-bit UP/DOWN Synchronous Counter

3.14.6 MODULUS-N-COUNTERS

The counter with 'n' Flip-Flops has maximum MOD number 2_n. Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation,

$$\mathbf{2_n \geq N}$$

(i) For example, a 3 bit binary counter is a MOD 8 counter. The basic counter can be modified to produce MOD numbers less than 2_n by allowing the counter to skin those are normally part of counting sequence.

n= 3

N= 8

$2_n = 2_3 = 8 = N$

(ii) MOD 5 Counter:

$2_n = N$

$2_n = 5$

$2_2 = 4$ less than N.

$2_3 = 8 > N(5)$

Therefore, 3 Flip-Flops are required.

(iii) MOD 10 Counter:

$2_n = N = 10$

$2_3 = 8$ less than N.

$2_4 = 16 > N(10)$.

To construct any MOD-N counter, the following methods can be used.

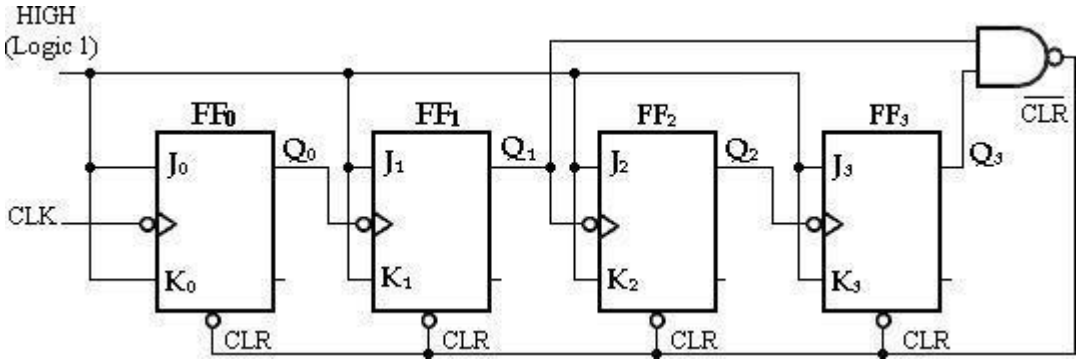
1. Find the number of Flip-Flops (n) required for the desired MOD number (N) using the equation,

   $\mathbf{2_n \geq N.}$
2. Connect all the Flip-Flops as a required counter.
3. Find the binary number for N.
4. Connect all Flip-Flop outputs for which Q= 1 when the count is N, as inputs to NAND gate.
5. Connect the NAND gate output to the CLR input of each Flip-Flop.

When the counter reaches N_{th} state, the output of the NAND gate goes LOW, resetting all Flip-Flops to 0. Therefore the counter counts from 0 through N-1.

For example, MOD-10 counter reaches state 10 (1010). i.e., $Q_3Q_2Q_1Q_0$= 1 0 1 0. The outputs Q_3 and Q_1 are connected to the NAND gate and the output of the NAND gate goes LOW and resetting all Flip-Flops to zero. Therefore MOD-10 counter counts from 0000 to 1001. And then recycles to the zero value.

The MOD-10 counter circuit is shown below.



MOD-10 (Decade) Counter

3.15 SHIFT REGISTERS:

A register is simply a group of Flip-Flops that can be used to store a binary number. There must be one Flip-Flop for each bit in the binary number. For instance, a register used to store an 8-bit binary number must have 8 Flip-Flops.

The Flip-Flops must be connected such that the binary number can be entered (shifted) into the register and possibly shifted out. A group of Flip-Flops connected to provide either or both of these functions is called a ***shift register***.

The bits in a binary number (data) can be removed from one place to another in either of two ways. The first method involves shifting the data one bit at a time in a serial fashion, beginning with either the most significant bit (MSB) or the least significant bit (LSB). This technique is referred to as *serial shifting*. The second method involves shifting all the data bits simultaneously and is referred to as *parallel shifting*.

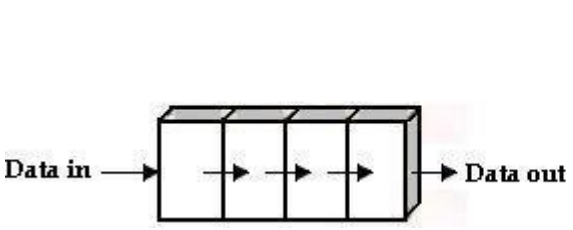
There are two ways to shift into a register (serial or parallel) and similarly two ways to shift the data out of the register. This leads to the construction of four basic register types—

i. Serial in- serial out,

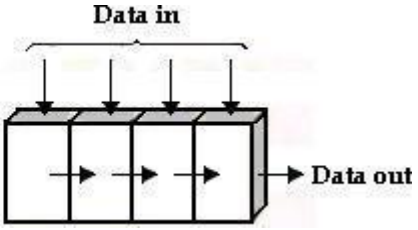
ii. Serial in- parallel out,

iii. Parallel in- serial out,

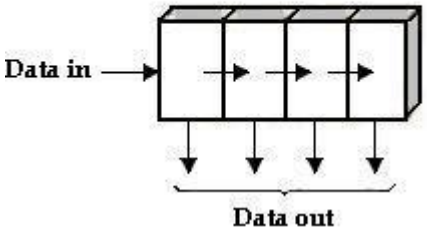
iv. Parallel in- parallel out.



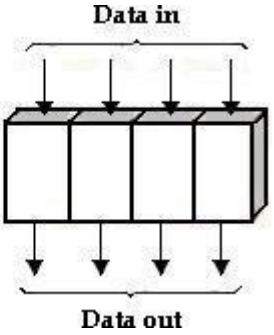
(i) Serial in- serial out



(iii) Parallel in- serial out



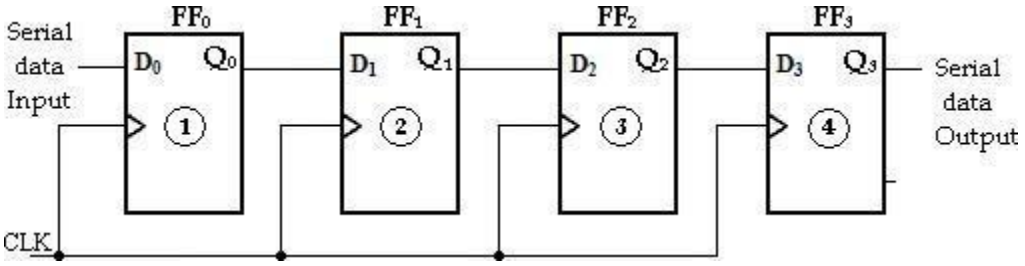
(iii) Serial in- parallel out



(iv) Parallel in- parallel out

3.15.1 Serial-In Serial-Out Shift Register:

The serial in/serial out shift register accepts data serially, i.e., one bit at a time on a single line. It produces the stored information on its output also in serial form.



Serial-In Serial-Out Shift Register

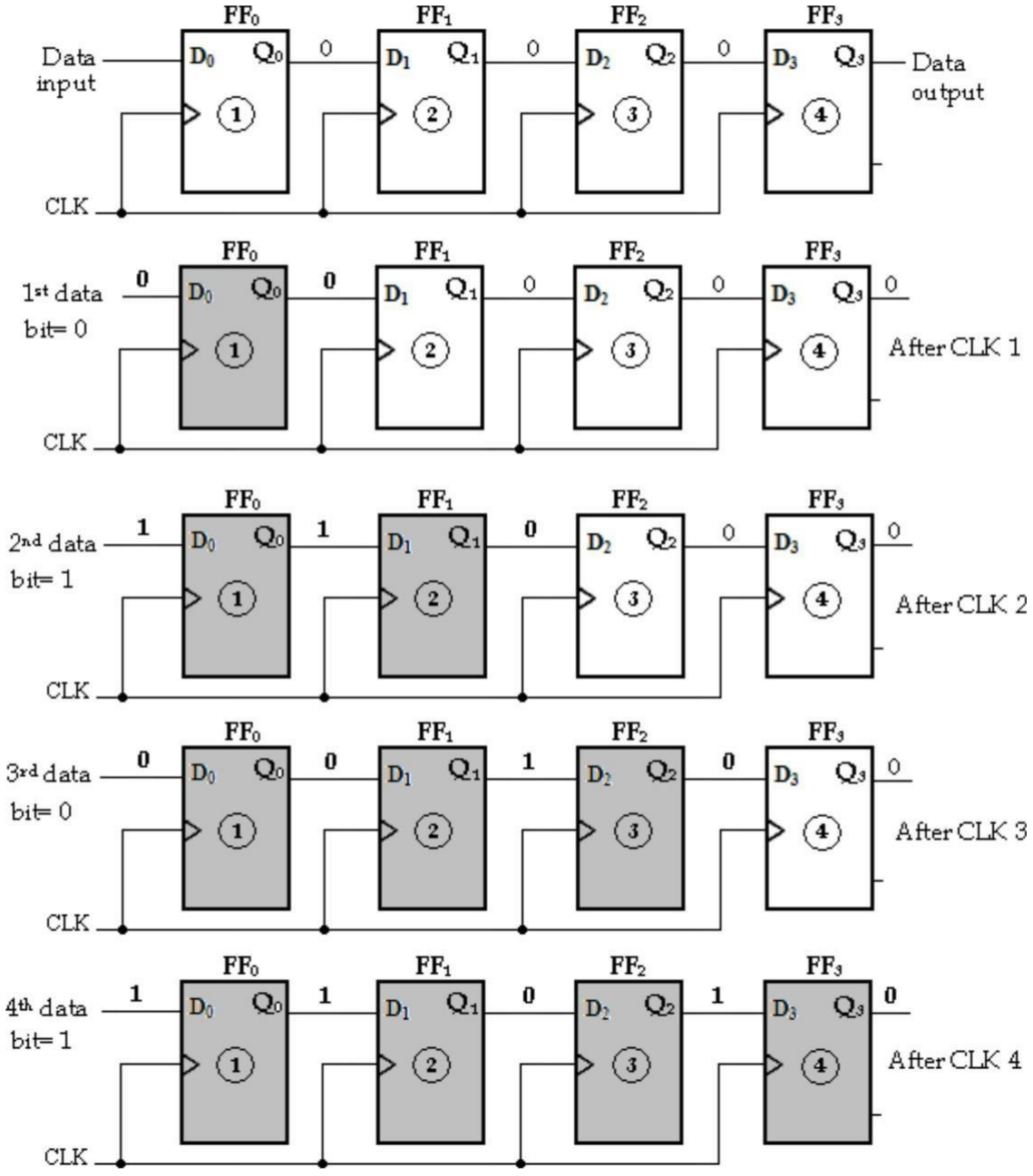
The entry of the four bits 1010 into the register is illustrated below, beginning with the right-most bit. The register is initially clear. The 0 is put onto the data input line, making D=0 for FF_0. When the first clock pulse is applied, FF_0 is reset, thus storing the0.

Next the second bit, which is a 1, is applied to the data input, making D=1 for FF_0 and D=0 for FF_1 because the D input of FF_1 is connected to the Q_0 output. When

the second clock pulse occurs, the 1 on the data input is shifted into FF0, causing FF0 to set; and the 0 that was in FF0 is shifted into FFl.

The third bit, a 0, is now put onto the data-input line, and a clock pulse is applied. The 0 is entered into FF_0, the 1 stored in FF_0 is shifted into FF_l, and the 0 stored in FF_1 is shifted into FF_2.

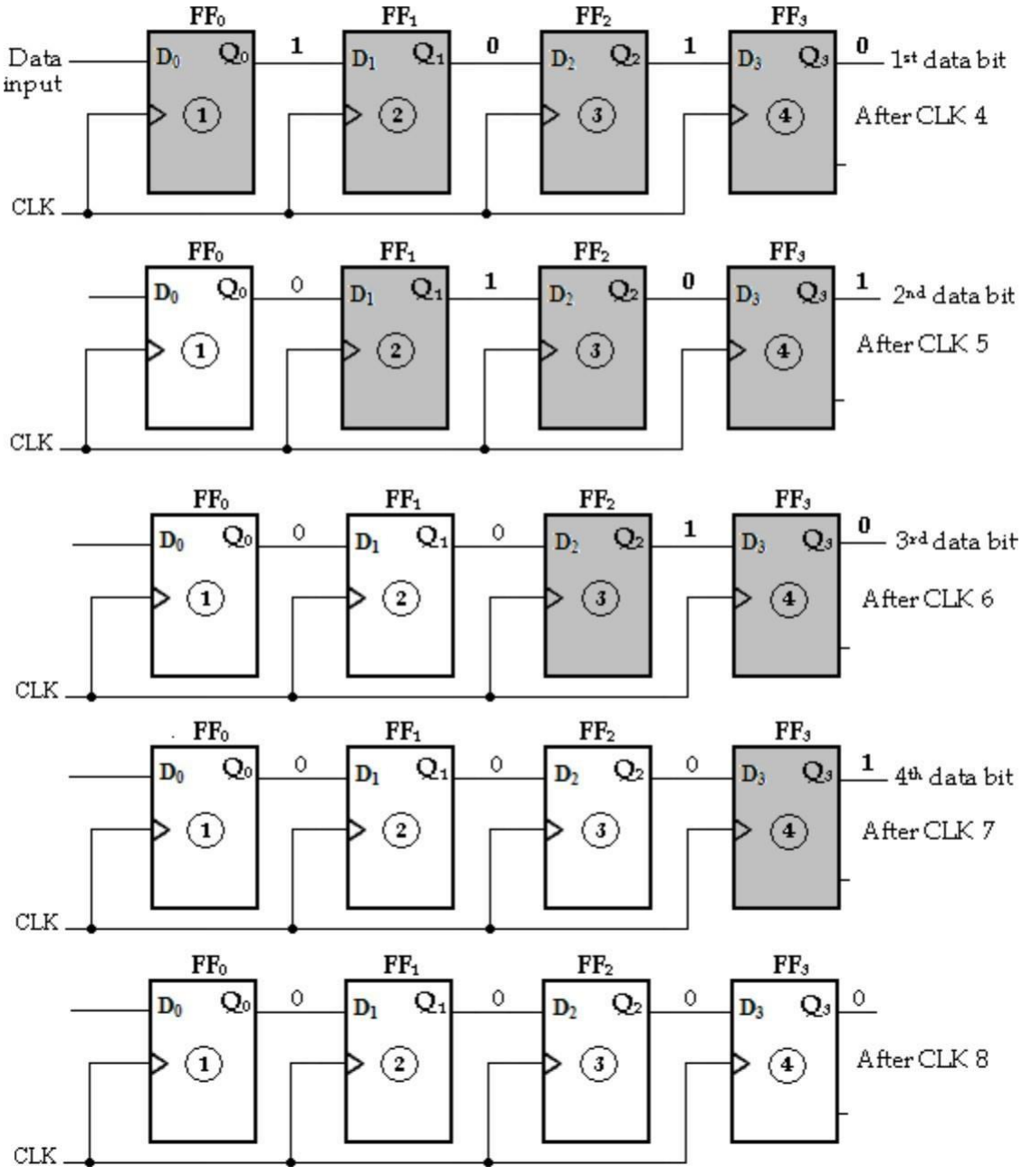
The last bit, a 1, is now applied to the data input, and a clock pulse is applied. This time the 1 is entered into FF0, the 0 stored in FF0 is shifted into FFl, the 1 storedin FF1 is shifted into FF2, and the 0 stored in FF2 is shifted into FF3. This completesthe serial entry of the four bits into the shift register, where they can be stored for any length of time as long as the Flip-Flops have dc power.



Four bits (1010) being entered serially into the register

To get the data out of the register, the bits must be shifted out serially and taken off the Q3 output. After CLK4, the right-most bit, 0, appears on the Q3 output.

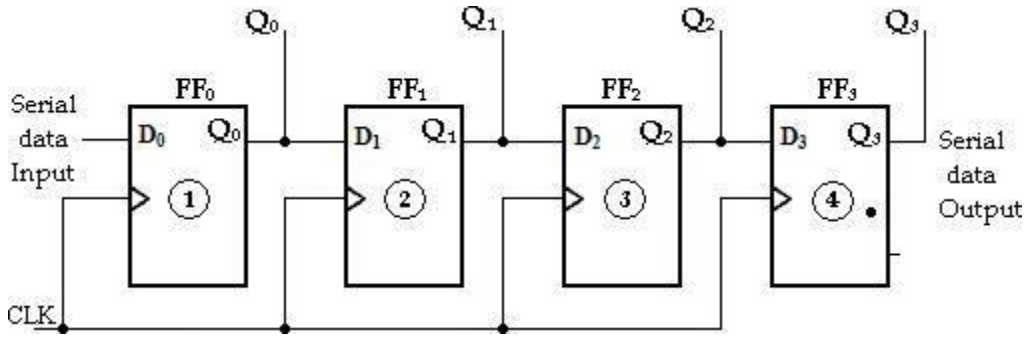
When clock pulse CLK5 is applied, the second bit appears on the Q3 output. Clock pulse CLK6 shifts the third bit to the output, and CLK7 shifts the fourth bit to the output. While the original four bits are being shifted out, more bits can be shifted in. All zeros are shown being shifted out, more bits can be shifted in.



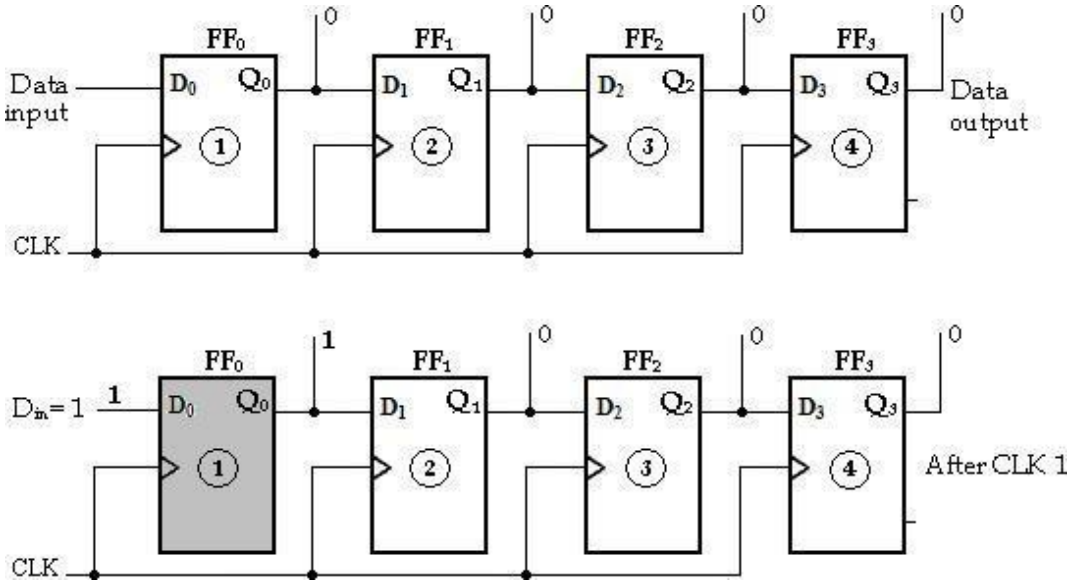
Four bits (1010) being entered serially-shifted out of the register and replaced by all zeros

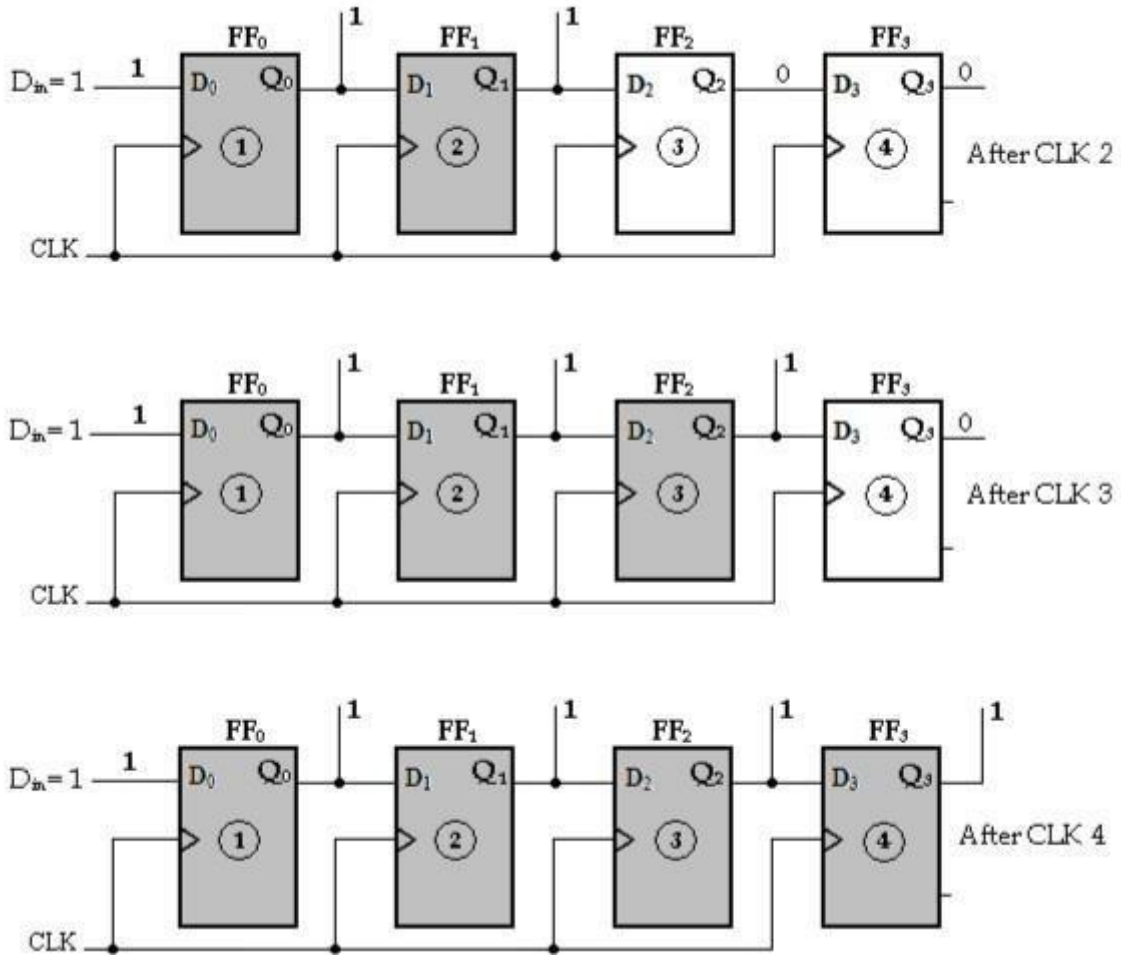
3.15.2 Serial-In Parallel-Out Shift Register:

In this shift register, data bits are entered into the register in the same as serial-in serial-out shift register. But the output is taken in parallel. Once the data arestored, each bit appears on its respective output line and all bits are available simultaneously instead of on a bit-by-bit.



Serial-In parallel-Out Shift Register





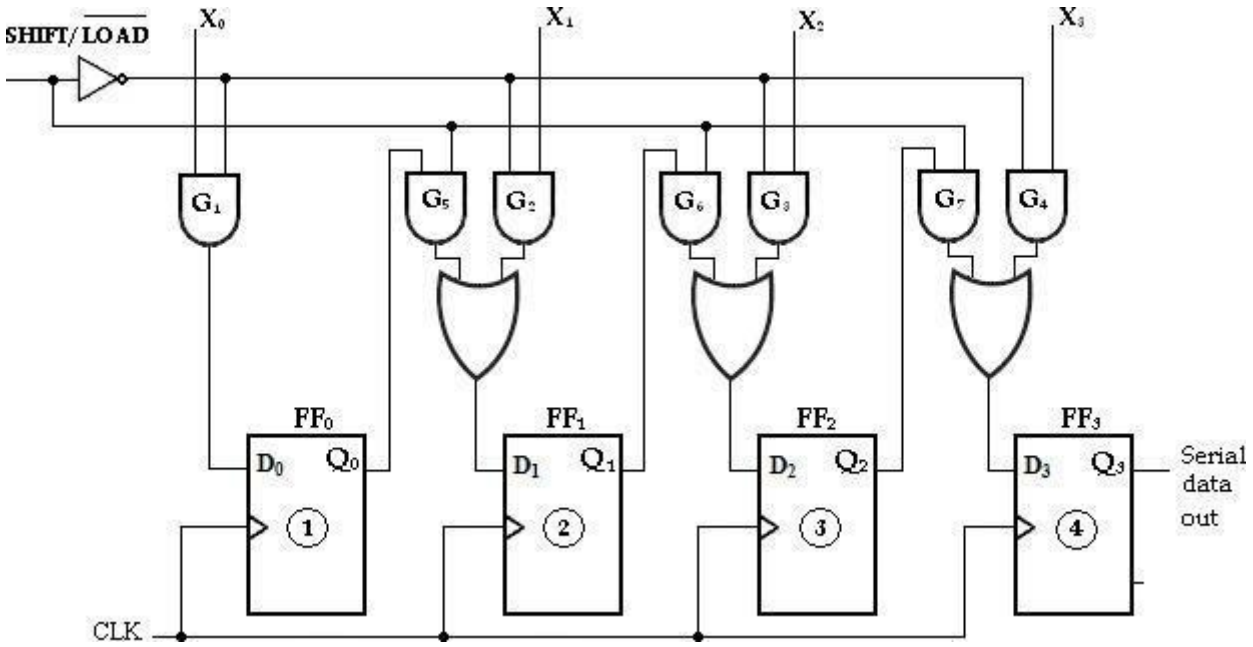
Four bits (1111) being serially entered into the register

3.15.3 Parallel-In Serial-Out Shift Register:

In this type, the bits are entered in parallel i.e., simultaneously into their respective stages on parallel lines.

A 4-bit parallel-in serial-out shift register is illustrated below. There are four data input lines, X_0, X_1, X_2 and X_3 for entering data in parallel into the register. SHIFT/ LOAD input is the control input, which allows four bits of data to **load** in parallel into the register.

When SHIFT/LOAD is LOW, gates G_1, G_2, G_3 and G_4 are enabled, allowing each data bit to be applied to the D input of its respective Flip-Flop. When a clock pulse is applied, the Flip-Flops with D = 1 will **set** and those with D = 0 will **reset,** thereby storing all four bits simultaneously.

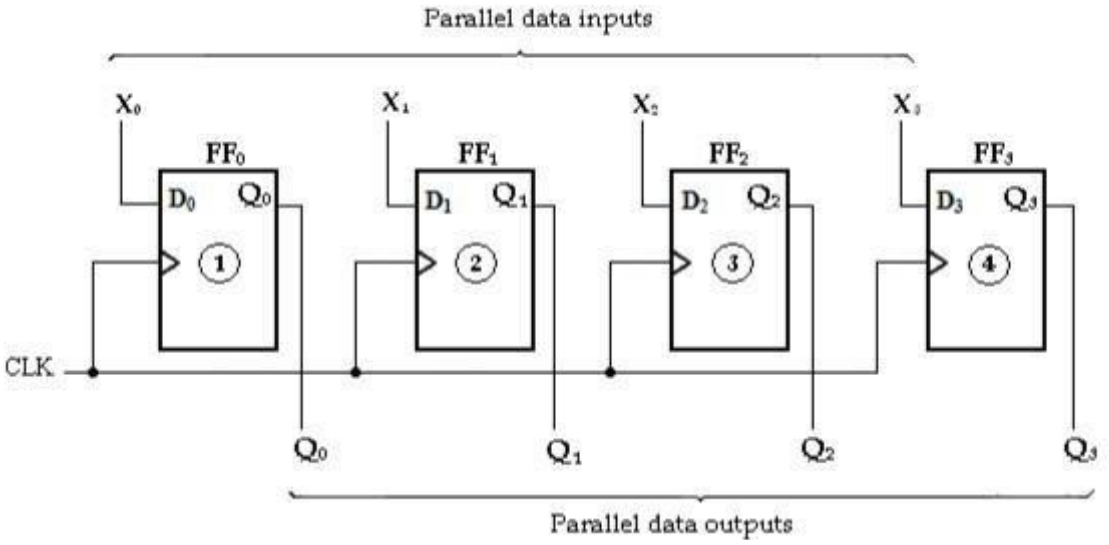


Parallel-In Serial-Out Shift Register

When SHIFT/LOAD is HIGH, gates G_1, G_2, G_3 and G_4 are disabled and gates G_5, G_6 and G_7 are enabled, allowing the data bits to shift right from one stage to the next. The OR gates allow either the normal shifting operation or the parallel data- entry operation, depending on which AND gates are enabled by the level on the SHIFT/LOAD input.

3.15.4 Parallel-In Parallel-Out Shift Register:

In this type, there is simultaneous entry of all data bits and the bits appear on parallel outputs simultaneously.



Parallel-In Parallel-Out Shift Register

3.15.5 UNIVERSAL SHIFT REGISTERS

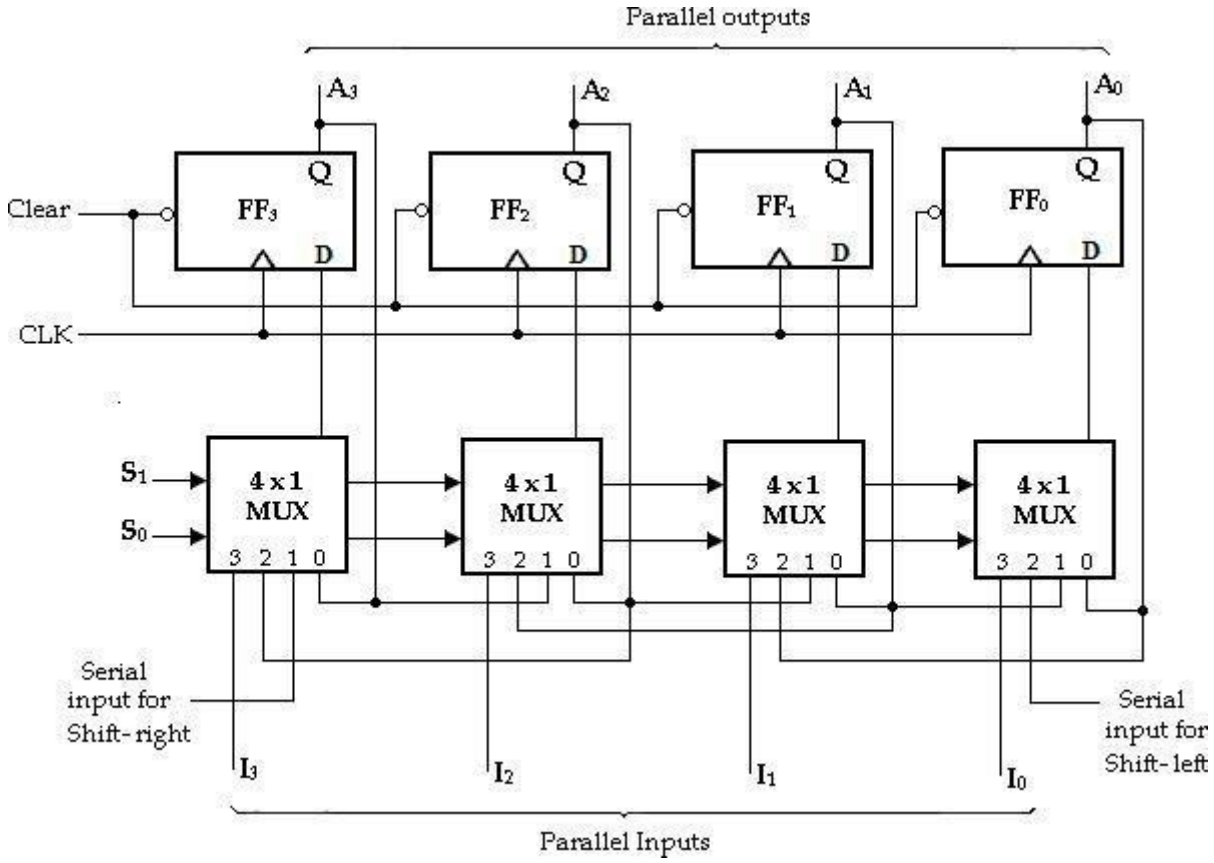
If the register has shift and parallel load capabilities, then it is called a shift register with parallel load or ***universal shift register***. Shift register can be used for converting serial data to parallel data, and vice-versa. If a parallel load capability is added to a shift register, the data entered in parallel can be taken out in serial fashion by shifting the data stored in the register.

The functions of universal shift register are:

1. A clear control to clear the register to 0.
2. A clock input to synchronize the operations.
3. A shift-right control to enable the shift right operation and the serial input and output lines associated with the shift right.
4. A shift-left control to enable the shift left operation and the serial input and output lines associated with the shift left.
5. A parallel-load control to enable a parallel transfer and the n input lines associated with the parallel transfer.
6. 'n' parallel output lines.
7. A control line that leaves the information in the register unchanged even though the clock pulses re continuously applied.

It consists of four D-Flip-Flops and four 4 input multiplexers (MUX). S_0 and S_1 are the two selection inputs connected to all the four multiplexers. These two selection inputs are used to select one of the four inputs of each multiplexer.

The input 0 in each MUX is selected when S_1S_0= 00 and input 1 is selected when S_1S_0= 01. Similarly inputs 2 and 3 are selected when S_1S_0= 10 and S_1S_0= 11 respectively. The inputs S1 and S0 control the mode of the operation of the register.



4-Bit Universal Shift Register

When S_1S_0= 00, the present value of the register is applied to the D-inputs of the Flip-Flops. This is done by connecting the output of each Flip-Flop to the 0 input of the respective multiplexer. The next clock pulse transfers into each Flip-Flop, the binary value is held previously, and hence no change of state occurs.

When S_1S_0= 01, terminal 1 of the multiplexer inputs has a path to the D inputs of the Flip-Flops. This causes a shift-right operation with the lefter serial input transferred into Flip-Flop FF_3.

When S_1S_0= 10, a shift-left operation results with the right serial input going into Flip-Flop FF_1.

Finally when S_1S_0= 11, the binary information on the parallel input lines (I_1, I_2, I_3 and I_4) are transferred into the register simultaneously during the next clock pulse.

The function table of bi-directional shift register with parallel inputs and parallel outputs is shown below.

Mode Control		Operation
S_1	S_0	
0	0	No change
0	1	Shift-right
1	0	Shift-left
1	1	Parallel load

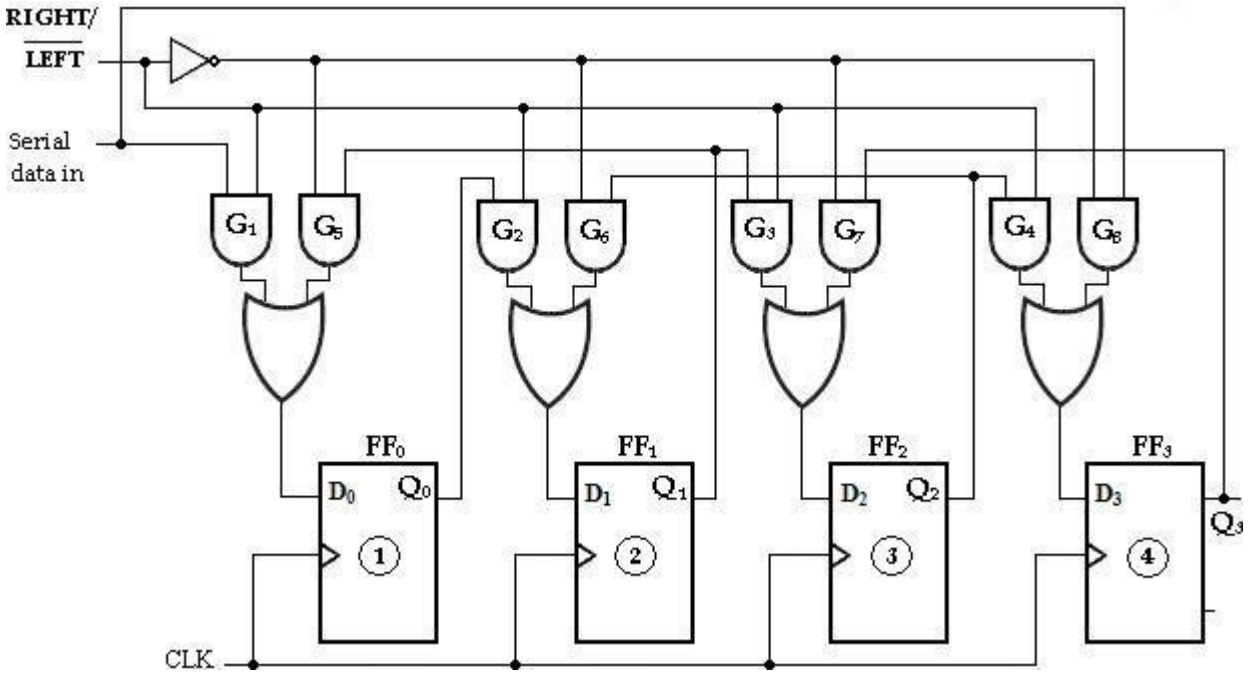
3.15.6 BI-DIRECTION SHIFT REGISTERS:

A bidirectional shift register is one in which the data can be shifted either left or right. It can be implemented by using gating logic that enables the transfer of a data bit from one stage to the next stage to the right or to the left depending on the level of a control line.

A 4-bit bidirectional shift register is shown below. A HIGH on the RIGHT/LEFT control input allows data bits inside the register to be shifted to the right, and a LOW enables data bits inside the register to be shifted to the left.

When the RIGHT/LEFT control input is **HIGH**, gates G_1, G_2, G_3 and G_4 are enabled, and the state of the Q output of each Flip-Flop is passed through to the D input of the following Flip-Flop. When a clock pulse occurs, the data bits are shifted one place to the right.

When the RIGHT/LEFT control input is **LOW**, gates G_5, G_6, G_7 and G_8 are enabled, and the Q output of each Flip-Flop is passed through to the D input of the preceding Flip-Flop. When a clock pulse occurs, the data bits are then shifted one place to the left.



4-bit bi-directional shift register

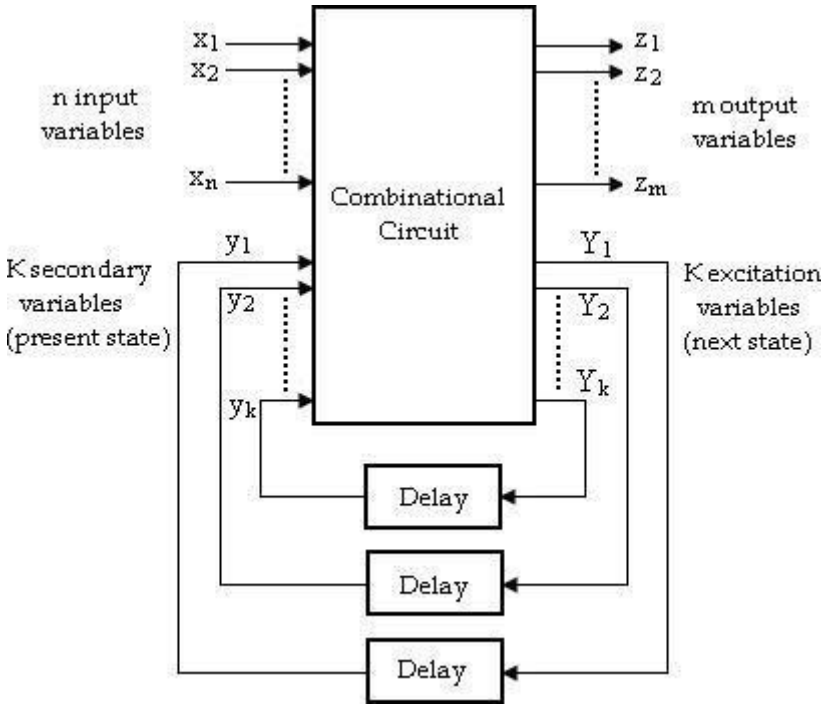
UNIT IV
ASYNCHRONOUS SEQUENTIAL CIRCUITS

4.1 INTRODUCTION

A sequential circuit is specified by a time sequence of inputs, outputs and internal states. In synchronous sequential circuits, the output changes whenever a clock pulse is applied. The memory elements are clocked flip-flops.

Asynchronous sequential circuits do not use clock pulses. The memory elements in asynchronous sequential circuits are either unclocked flip-flops (Latches)or time-delay elements.

S.No	Synchronous sequential circuits	Asynchronous sequential circuits
1	Memory elements are clocked flip-flops	Memory elements are either unclocked flip-flops or time delay elements.
2	The change in input signals can affect memory element upon activation of clock signal.	The change in input signals can affect memory element at any instant of time.
3	The maximum operating speed of clock depends on time delays involved. Therefore synchronous circuits can operate slower than asynchronous.	Because of the absence of clock, it can operate faster than synchronous circuits.
4	Easier to design	More difficult to design



Block diagram of Asynchronous sequential circuits

The block diagram of asynchronous sequential circuit is shown above. It consists of a combinational circuit and delay elements connected to form feedback loops. There are 'n' input variables, 'm' output variables, and 'k' internal states.

The delay elements provide short term memory for the sequential circuit. The present-state and next-state variables in asynchronous sequential circuits are called secondary variables and excitation variables, respectively.

When an input variable changes in value, the 'y' secondary variable does not change instantaneously. It takes a certain amount of time for the signal to propagate from the input terminals through the combinational circuit to the 'Y' excitation variables where the new values are generated for the next state. These values propagate through the delay elements and become the new present state for the secondary variables.

In steady-state condition, excitation and secondary variables are same, but during transition they are different.

To ensure proper operation, it is necessary for asynchronous sequential circuits to attain a stable state before the input is changed to a new value. Because of unequal delays in wires and combinational circuits, it is impossible to have two or

more input variable change at exactly same instant. Therefore, simultaneous changes of two or more input variables are avoided.

Only one input variable is allowed to change at any one time and the time between input changes is kept longer than the time it takes the circuit to reach stable state.

Types:

According to how input variables are to be considered, there are two types

- Fundamental mode circuit
- Pulse mode circuit.

Fundamental mode circuit assumes that:

- The input variables change only when the circuit is stable.
- Only one input variable can change at a given time.
- Inputs are levels (0, 1) and not pulses.

Pulse mode circuit assumes that:

- The input variables are pulses (True, False) instead of levels.
- The width of the pulses is long enough for the circuit to respond to the input.
- The pulse width must not be so long that it is still present after the new state is reached.

4.2 Analysis of Fundamental Mode Circuits

The analysis of asynchronous sequential circuits consists of obtaining a table or a diagram that describes the sequence of internal states and outputs as a function of changes in the input variables.

4.2.1 Analysis procedure

The procedure for obtaining a transition table from the given circuit diagram is as follows.

1. Determine all feedback loops in the circuit.
2. Designate the output of each feedback loop with variable Y1 and its corresponding inputs $y_1, y_2, \ldots y_k$, where k is the number of feedback loops in the circuit.
3. Derive the Boolean functions of all Y's as a function of the external inputs and the y's.
4. Plot each Y function in a map, using y variables for the rows and the external inputs for the columns.
5. Combine all the maps into one table showing the value of $Y= Y_1, Y_2, \ldots Y_k$ inside each square.
6. Circle all stable states where Y=y. The resulting map is the transition table.

4.2.2 Problems

1. An asynchronous sequential circuit is described by the following excitation and output function,

$$\mathbf{Y= x_1x_2+ (x_1+x_2)\, y}$$

$$\mathbf{Z= Y}$$

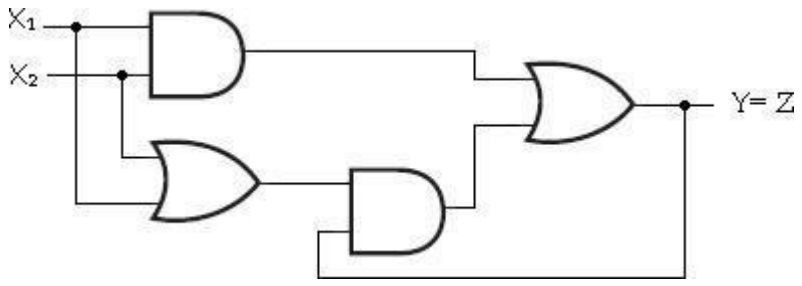
a) Draw the logic diagram of the circuit.

b) Derive the transition table, flow table and output map.

c) Describe the behavior of the circuit.

Soln:

i) The logic diagram is shown as,

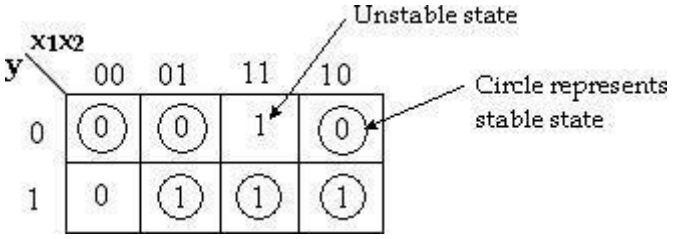


Logic diagram

ii)

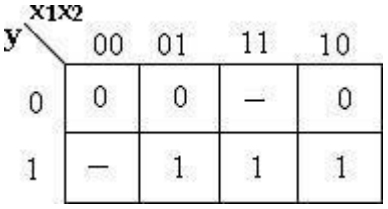
y	x_1	x_2	x_1x_2	$(x_1+x_2)y$	$Y= x_1x_2+ (x_1+x_2)y$	Z= Y
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	1	0	1	1
1	0	0	0	0	0	0
1	0	1	0	1	1	1
1	1	0	0	1	1	1
1	1	1	1	1	1	1

Transition table:



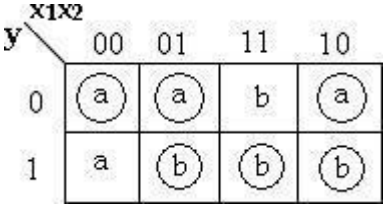
Output map:

Output is mapped for all stable states. For unstable states output is mapped unspecified.



Flow table:

Assign a= 0; b= 1



iii)

The circuit gives carry output of the full adder circuit.

2. Design an asynchronous sequential circuit that has two internal states and one output. The excitation and output function describing the circuit are as follows:

$Y_1 = x_1x_2 + x_1y_2 + x_2y_1$

$Y_2 = x_2 + x_1y_1y_2 + x_1y_1$

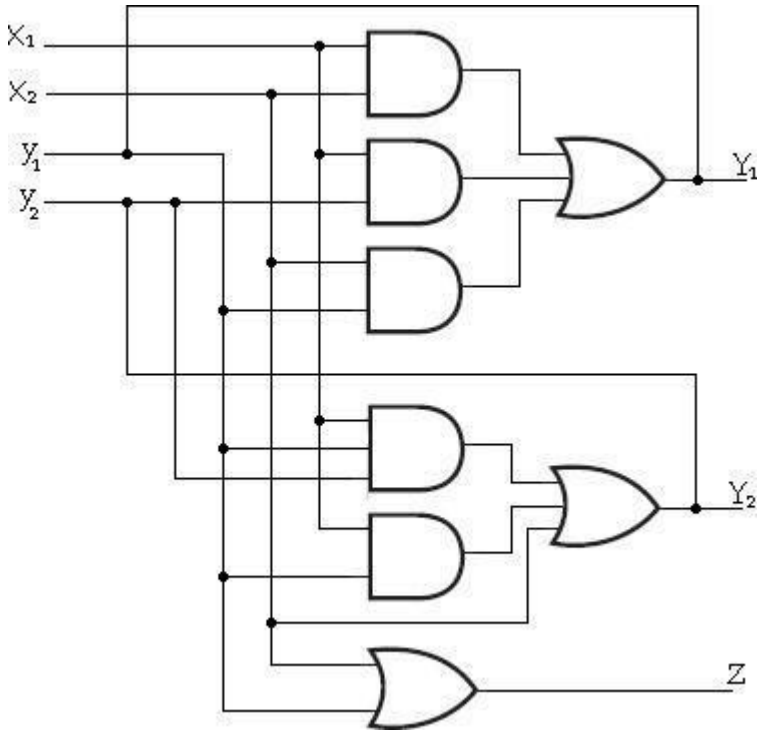
$Z = x_2 + y_1$.

a) Draw the logic diagram of the circuit.

b) Derive the transition table, output map and flow table.

Soln:

i) The logic diagram is shown as,



Logic Diagram

ii)

y_1	y_2	x_1	x_2	x_1x_2	y_2x_1	y_1x_2	$Y_1x_1y_2$	y_2x_1	Y_1	Y_2	y_1
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	1	1
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	0	0	0	1	1	1

0	1	0	0	0	0	0	0	0	0	0	0
0	1	0	1	0	0	0	0	0	0	1	1
0	1	1	0	0	1	0	0	0	1	0	0
0	1	1	1	1	1	0	0	0	1	1	1
1	0	0	0	0	0	0	0	0	0	0	1
1	0	0	1	0	0	1	0	0	1	1	1
1	0	1	0	0	0	0	0	1	0	1	1
1	0	1	1	1	0	1	0	1	1	1	1
1	1	0	0	0	0	0	0	0	0	0	1
1	1	0	1	0	0	1	0	0	1	1	1
1	1	1	0	0	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1

Map for Y_1

y_1y_2 \ x_1x_2	00	01	11	10
00	0	0	1	0
01	0	0	1	1
11	0	1	1	1
10	0	1	1	0

Map for Y_2

y_1y_2 \ x_1x_2	00	01	11	10
00	0	1	1	0
01	0	1	1	0
11	0	1	1	1
10	0	1	1	1

Transition table and Output map

Map for Y_1Y_2

y_1y_2 \ x_1x_2	00	01	11	10
00	(00)	01	11	(00)
01	00	(01)	11	10
11	00	(11)	(11)	(11)
10	00	11	11	01

Transition table

y_1y_2 \ x_1x_2	00	01	11	10
00	0	–	–	0
01	–	1	–	–
11	–	1	1	1
10	–	–	–	–

Output map

Primitive Flow table

y_1y_2 \ x_1x_2	00	01	11	10
a	(a)	b	c	(a)
b	a	(b)	c	d
c	a	(c)	(c)	(c)
d	a	c	c	b

3. An asynchronous sequential circuit is described by the excitation and output functions,

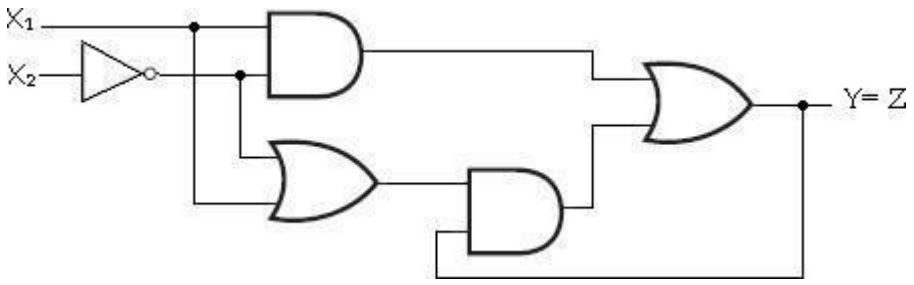
   $Y = x_1x_2' + (x_1+x_2')\,y$

   $Z = Y$

   a) Draw the logic diagram of the circuit.

   b) Derive the transition table, output map and flow table.

Soln:



Logic diagram

ii)

y	x_1	x_2	x_2'	x_1x_2'	$(x_1+x_2')y$	$Y = x_1x_2' + (x_1+x_2')y$	$Z = Y$
0	0	0	1	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	1	1	0	1	1
0	1	1	0	0	0	0	0
1	0	0	1	0	1	1	1
1	0	1	0	0	0	0	0
1	1	0	1	1	1	1	1
1	1	1	0	0	1	1	1

Transition table:

y \ x1x2	00	01	11	10
0	(0)	(0)	(0)	1
1	(1)	0	(1)	(1)

Transition Table

Output map:

Output is mapped for all stable states. For unstable states output is mapped unspecified.

y \ x1x2	00	01	11	10
0	0	0	0	–
1	1	–	1	1

Output map

Flow table:

Assign a= 0; b= 1

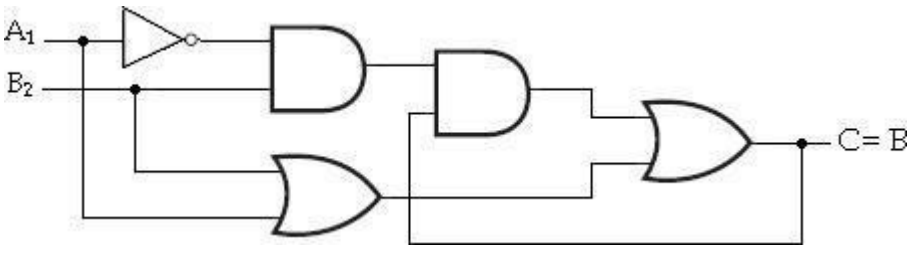
y \ x1x2	00	01	11	10
0	(a)	(a)	(a)	b
1	(b)	a	(b)	(b)

4. An asynchronous sequential circuit is described by the excitation and output functions, **$B= (A_1'B_2)\, b+ (A_1+B_2)$ $C= B$**

a) Draw the logic diagram of the circuit.

b) Derive the transition table, output map and flow table.

Soln:



Logic Diagram

ii)

b	A_1	B_2	A_1'	$(A_1'B_2)b$	A_1+B_2	$B= (A_1'B_2)\, b+ (A_1+B_2)$	C= B
0	0	0	1	0	0	0	0
0	0	1	1	0	1	1	1
0	1	0	0	0	1	1	1
0	1	1	0	0	1	1	1
1	0	0	1	0	0	0	0
1	0	1	1	1	1	1	1
1	1	0	0	0	1	1	1
1	1	1	0	0	1	1	1

Transition table

b \ A_1B_2	00	01	11	10
0	(0)	1	1	1
1	0	(1)	(1)	(1)

Output map

Output is mapped for all stable states.

b \ A_1B_2	00	01	11	10
0	0	–	–	–
1	–	1	1	1

Flow table

Assign a= 0; b= 1

b \ A_1B_2	00	01	11	10
0	(a)	b	b	b
1	a	(b)	(b)	(b)

5. An asynchronous sequential circuit is described by the excitation and output functions,

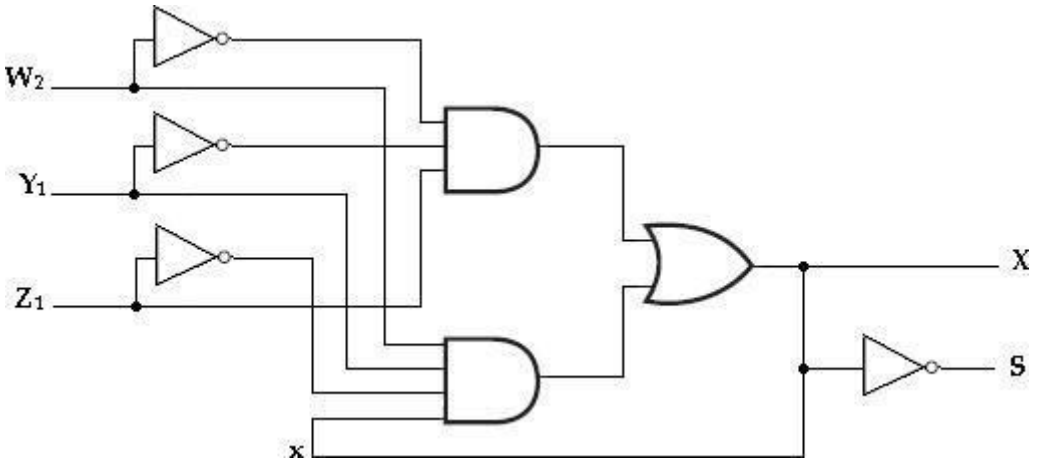
$$X = (Y_1 Z_1' W_2) x + (Y_1' Z_1 W_2')$$

$$S = X'$$

a) Draw the logic diagram of the circuit

b) Derive the translation table and output map

Soln:



x	W_2	W_2'	Y_1	Y_1'	Z_1	Z_1'	$(Y' Z' _2) x$	$Y_1' Z_1$	X	S= X'
0	0	1	0	1	0	1	0	0	0	1
0	0	1	0	1	1	0	0	1	1	0
0	0	1	1	0	0	1	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1
0	1	0	0	1	0	1	0	0	0	1
0	1	0	0	1	1	0	0	0	0	1
0	1	0	1	0	0	1	0	0	0	1
0	1	0	1	0	1	0	0	0	0	1
1	0	1	0	1	0	1	0	0	0	1
1	0	1	0	1	1	0	0	1	1	0
1	0	1	1	0	0	1	0	0	0	1
1	0	1	1	0	1	0	0	0	0	1

1	1	0	0	1	0	1	0	0	0	1
1	1	0	0	1	1	0	0	0	0	1
1	1	0	1	0	0	1	1	0	1	0
1	1	0	1	0	1	0	0	0	0	1

Map for X

xW_2 \ Y_1Z_1	00	01	11	10
00	0	1	0	0
01	0	0	0	0
11	0	0	0	1
10	0	1	0	0

Map for S

xW_2 \ Y_1Z_1	00	01	11	10
00	1	0	1	1
01	1	1	1	1
11	1	1	1	0
10	1	0	1	1

Transition table and Output map:

Map for XS

xW_2 \ Y_1Z_1	00	01	11	10
00	01	10	01	01
01	(01)	(01)	(01)	(01)
11	01	01	01	10
10	01	(10)	01	01

Transition table

xW_2 \ Y_1Z_1	00	01	11	10
00	–	–	–	–
01	1	1	1	1
11	–	–	–	–
10	–	0	–	–

Output map

4.3 Analysis of Pulse Mode Circuits

Pulse mode asynchronous sequential circuits rely on the input pulses rather than levels. They allow only one input variable to change at a time. They can be implemented by employing a SR latch.

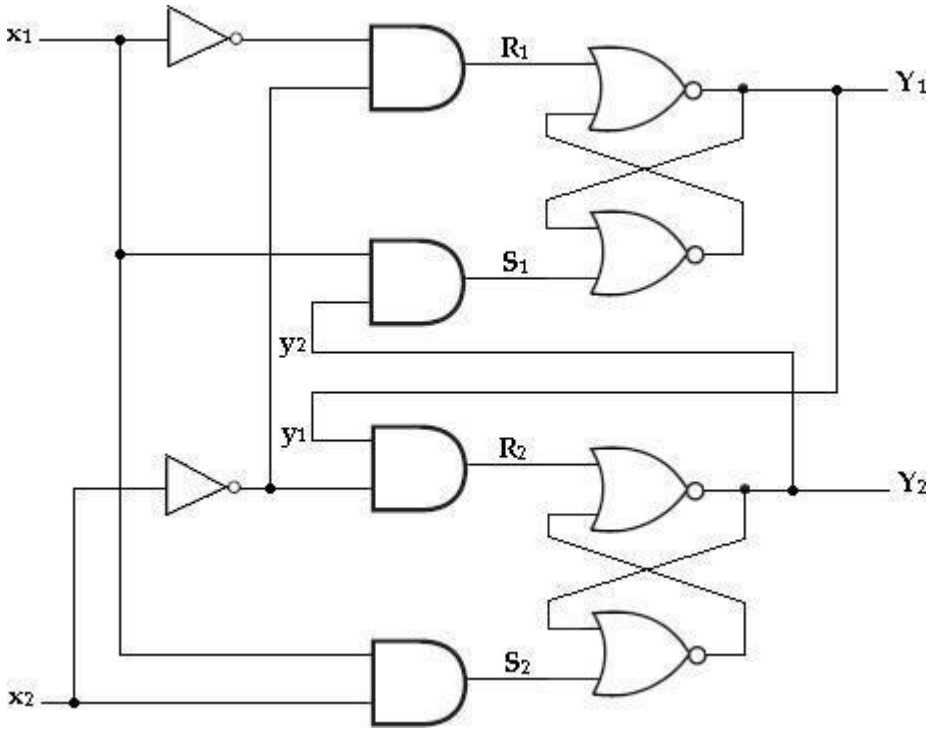
The procedure for analyzing an asynchronous sequential circuit with SR latches can be summarized as follows:

1. Label each latch output with Y_i and its external feedback path (if any) with y_i for

   $i = 1,2,..,, k.$
2. Derive the Boolean functions for the S_i and R_i inputs in each latch.
3. Check whether $\boldsymbol{SR = 0}$ for each NOR latch or whether $\mathbf{S'R' = 0}$ for each NAND latch. If either of these condition is not satisfied, there is a possibility that the circuit may not operate properly.
4. Evaluate $\mathbf{Y = S + R'y}$ for each NOR latch or $\mathbf{Y = S' + Ry}$ for each NAND latch.
5. Construct a map with the y's representing the rows and the x inputs representing the columns.
6. Plot the value of $Y= Y_1Y_2 \ldots\ldots Y_k$ in the map.
7. Circle all stable states such that $\boldsymbol{Y = y}$. The resulting map is the transition table.

The analysis of a circuit with latches will be demonstrated by means of the below example.

1. Derive the transition table for the pulse mode asynchronous sequential circuit shown below.



Example of a circuit with SR latches

Soln:

There are two inputs x_1 and x_2 and two external feedback loops giving rise to the secondary variables y_1 and y_2.

Step 1:

The Boolean functions for the S and R inputs in each latch are:

$$\mathbf{S_1 = x_1 y_2} \qquad \mathbf{S_2 = x_1 x_2}$$

$$\mathbf{R_1 = x_1' x_2'} \qquad \mathbf{R_2 = x_2' y_1}$$

Step 2:

Check whether the conditions SR= 0 is satisfied to ensure proper operation of the circuit.

$$\mathbf{S_1 R_1 = x_1 y_2 \, x_1' x_2' = 0}$$

$$\mathbf{S_2 R_2 = x_1 x_2 \, x_2' y_1 = 0}$$

The result is 0 because $x_1 x_1' = x_2 x_2' = 0$

Step 3:

Evaluate Y_1 and Y_2. The excitation functions are derived from the relation **Y= S+ R'y.** $\mathbf{Y_1 = S_1 + R_1' y_1} = x_1 y_2 + (x_1' x_2')' \, y_1$

$$= x_1 y_2 + (x_1 + x_2) \, y_1 = x_1 y_2 + x_1 y_1 + x_2 y_1$$

$\mathbf{Y_2 = S_2 + R_2' y_2} = x_1 x_2 + (x_2' y_1)' y_2$

$$= x_1 x_2 + (x_2 + y_1') \, y_2 = x_1 x_2 + x_2 y_2 + y_1' y_2$$

y_1	y_2	x_1	x_2	x_1y_2	x_1y_1	x_2y_1	x_1x_2	x_2y_2	$y_1'y_2$	Y_1	Y_2
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	1	0	0	0	1
0	1	0	0	0	0	0	0	0	1	0	1
0	1	0	1	0	0	0	0	1	1	0	1
0	1	1	0	1	0	0	0	0	1	1	1
0	1	1	1	1	0	0	1	1	1	1	1
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0	1	0

1	0	1	0	0	1	0	0	0	0	1	0
1	0	1	1	0	1	1	1	0	0	1	1
1	1	0	0	0	0	0	0	0	0	0	0
1	1	0	1	0	0	1	0	1	0	1	1
1	1	1	0	1	1	0	0	0	0	1	0
1	1	1	1	1	1	1	1	1	0	1	1

Step 4:

Maps for Y_1 and Y_2.

Map for Y_1

y_1y_2 \ x_1x_2	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	1	1
10	0	1	1	1

Map for Y_2

y_1y_2 \ x_1x_2	00	01	11	10
00	0	0	1	0
01	1	1	1	1
11	0	1	1	0
10	0	0	1	0

Step 5:

Transition table

Map for Y_1Y_2

y_1y_2 \ x_1x_2	00	01	11	10
00	(00)	(00)	01	(00)
01	(01)	(01)	11	11
11	00	(11)	(11)	10
10	00	(10)	11	(10)

4.4 RACES:

A race condition is said to exist in an asynchronous sequential circuit when two or more binary state variables change value in response to a change in an input variable.

Races are classified as:

i. Non-critical races

ii. Critical races.

Non-critical races:

If the final stable state that the circuit reaches does not depend on the order in which the state variables change, the race is called a non-critical race.

If a circuit, whose transition table (a) starts with the total stable state y_1y_2x= *000* and then change the input from 0 to 1. The state variables must then change from 00 to 11, which define a race condition.

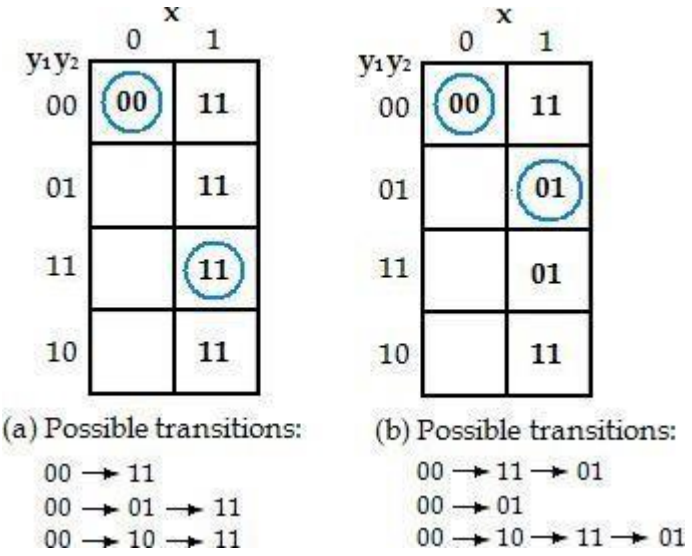
The possible transitions are:

00 → 11

00 → 01 → 11

00 → 10 → 11

In all cases, the final state is the same, which results in a non-critical condition. In (a), the final state is (y_1y_2x= 111), and in (b), it is (y_1y_2x= 011).



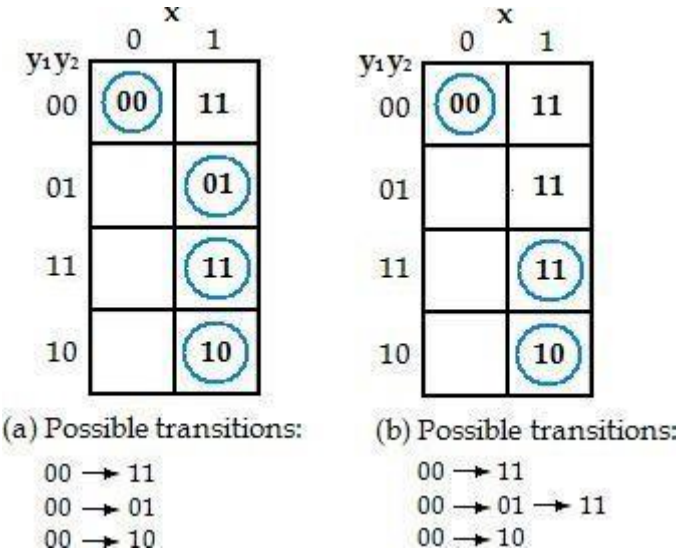
Examples of Non-critical Races

Critical races:

A race becomes critical if the correct next state is not reached during a state transition. If it is possible to end up in two or more different stable states, depending on the order in which the state variables change, then it is a critical race. For proper operation, critical races must be avoided.

The below transition table illustrates critical race condition. The transition table (a) starts in stable state (y_1y_2x= 000), and then change the input from 0 to 1. The state variables must then change from 00 to 11. If they change simultaneously, the final total stable state is 111. In the transition table (a), if, because of unequal propagation delay, Y_2 changes to 1 before Y_1 does, then the circuit goes to the total stable state 011 and remains there. If, however, Y_1 changes first, the internal state becomes 10 and the circuit will remain in the stable total state 101.

Hence, the race is critical because the circuit goes to different stable states, depending on the order in which the state variables change.

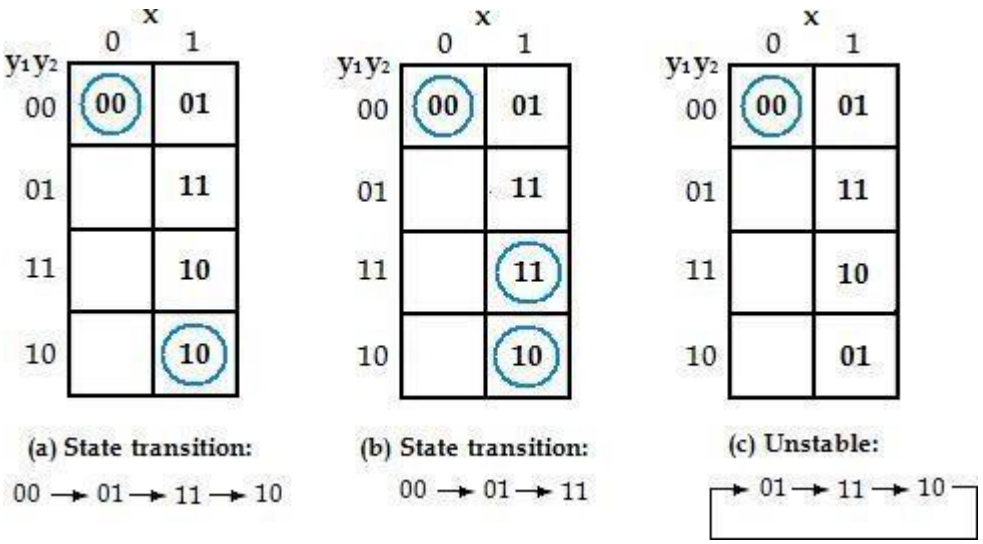


Examples of Critical Races

4.5 CYCLES

Races can be avoided by directing the circuit through intermediate unstable states with a unique state-variable change. When a circuit goes through a unique sequence of unstable states, it is said to have a ***cycle***.

Again, we start with $y_1y_2 = 00$ and change the input from *0* to 1. The transition table (a) gives a *unique* sequence that terminates in a total stable state 101. The tablein (b) shows that even though the state variables change from 00 to 11, the cycleprovides a unique transition from *00* to 01 and then to 11, Care must be taken when using a cycle that terminates with a stable state. If a cycle does not terminate with a stable state, the circuit will keep going from one unstable state to another, making the entire circuit unstable. This is demonstrated in the transition table (c).



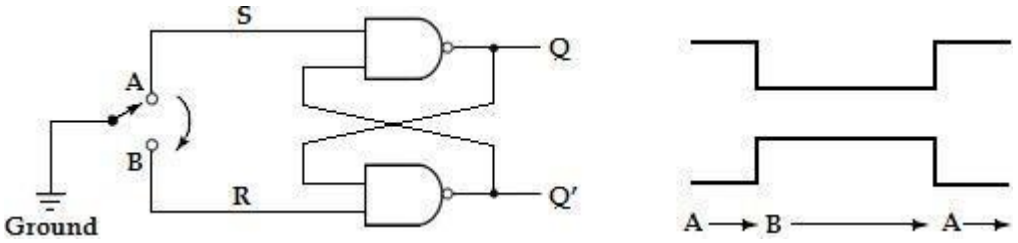
Examples of Cycles

Debounce Circuit:

Input binary information in binary information can be generated manually be means of mechanical switches. One position of the switch provides a voltage equivalent to logic 1, and the other position provides a second voltage equivalent to logic 0. Mechanical switches are also used to start, stop, or reset the digital system. A common characteristic of a mechanical switch is that when the arm is thrown fromone position to the other the switch contact vibrates or bounces several times before coming to a final rest. In a typical switch, the contact bounce may take several milliseconds to die out, causing the signal to oscillate between 1 and 0 because the switch contact is vibrating.

A debounce circuit is a circuit which removes the series of pulses that result from a contact bounce and produces a single smooth transition of the binary signal from *0* to 1 or from 1 to 0. One such circuit consists of a single-pole, double-throw switch connected to an *SR* latch, as shown below. The center contact is connected to

ground that provides a signal equivalent to logic 0. When one of the two contacts, A or *B,* is not connected to ground through the switch, it behaves like a logic-1 signal. When the switch is thrown from position *A* to position *B* and back, the outputs of the latch produce a single pulse as shown, negative for *Q* and positive for *Q′*. The switch is usually a push button whose contact rests in position A. When the pushbutton is depressed, it goes to position B and when released, it returns to position A.



Debounce Circuit

The operation of the debounce circuit is as follows: When the switch resets in position A, we have the condition S = 0, *R* = 1 and *Q* = 1, *Q′* = 0. When the switch is moved to position *B,* the ground connection causes R to go to *0,* while *S* becomes a 1 because contact A is open. This condition in turn causes output Q to go to 0 and Q' to go to 1. After the switch makes an initial contact with *B,* it bounces several times. The output of the latch will be unaffected by the contact bounce because Q' remains 1 (and Q remains *0)* whether *R* is equal to *0* (*contact with ground*) or equal to 1 (*no contact with ground*). When the switch returns to position A, *S* becomes *0* and *Q* returns to 1. The output again will exhibit a smooth transition, even if there is a contact bounce in position A.

4.6 DESIGN OF FUNDAMENTAL MODE SEQUENTIAL CIRCUITS

The design of an asynchronous sequential circuit starts from the statement of the problem and concludes in a logic diagram. There are a number of design steps that must be carried out in order to minimize the circuit complexity and to produce a stable circuit without critical races.

The design steps are as follows:

1. State the design specifications.
2. Obtain a primitive flow table from the given design specifications.
3. Reduce the flow table by merging rows in the primitive flow table.
4. Assign binary state variables to each row of the reduced flow table to obtain the transition table. The procedure of state assignment eliminates any possible critical races.
5. Assign output values to the dashes associated with the unstable states to obtain the output maps.
6. Simplify the Boolean functions of the excitation and output variables and draw the logic diagram.

1. Design a gated latch circuit with inputs, G (gate) and D (data), and one output, Q. Binary information present at the D input is transferred to the Q output when G is equal to 1. The Q output will follow the D input as long as G= 1. When G goesto0, the information that was present at the D input at the time of transitionoccurred is retained at the Q output. The gated latch is a memory element that accepts the value of D when G= 1 and retains this value after G goes to 0, a changein D does not change the value of the output Q.

Soln:

Step 1:

From the design specifications, we know that Q= 0 if DG= 01

and Q= 1 if DG= 11

because D must be equal to Q when G= 1.

When G goes to 0, the output depends on the last value of D. Thus, if the transition is from 01 to 00 to 10, then Q must remain 0 because D is 0 at the time ofthe transition from 1 to 0 in G.

If the transition of DG is from 11 to 10 to 00, then Q must remain 1. This information results in six different total states, as shown in the table.

State	Inputs		Output	Comments
	D	G	Q	
a	0	1	0	D= Q because G= 1
b	1	1	1	D= Q because G= 1
c	0	0	0	After state a or d
d	1	0	0	After state c
e	1	0	1	After state b or f
f	0	0	1	After state e

Step 2: A primitive flow is a flow table with only one stable total state in each row. It has one row for each state and one column for each input combination.

Step 3:

	DG 00	01	11	10
a	c, –	(a), 0	b, –	–, –
b	–, –	a, –	(b), 1	e, –
c	(c), 0	a, –	–, –	d, –
d	c, –	–, –	b, –	(d), 0
e	f, –	–, –	b, –	(e), 1
f	(f), 1	a, –	–, –	e, –

Primitive flow table

The primitive flow table has only stable state in each row. The table can be reduced to a smaller number of rows if two or more stable states are placed in the same row of the flow table. The grouping of stable states from separate rows into one common row is called ***merging***.

	DG			
	00	01	11	10
a	c, –	ⓐ, 0	b, –	–, –
c	ⓒ, 0	a, –	–, –	d, –
d	c, –	–, –	b, –	ⓓ, 0

	DG			
	00	01	11	10
b	–, –	a, –	ⓑ, 1	e, –
e	f, –	–, –	b, –	ⓔ, 1
f	ⓕ, 1	a, –	–, –	e, –

States that are candidates for merging

Thus, the three rows a, c, and d can be merged into one row. The second row of the reduced table results from the merging of rows b, e, and f of the primitive flow table.

	DG			
	00	01	11	10
a, c, d	ⓒ, 0	ⓐ, 0	b, –	ⓓ, 0
b, e, f	ⓕ, 1	a, –	ⓑ, 1	ⓔ, 1

Reduced table- 1

The states c & d are replaced by state a, and states e & f are replaced by state b

	DG			
	00	01	11	10
a	ⓐ, 0	ⓐ, 0	b, –	ⓐ, 0
b	ⓑ, 1	a, –	ⓑ, 1	ⓑ, 1

Reduced table- 2

Step 4:

Assign distinct binary value to each state. This assignment converts the flow table into a transition table. A binary state assignment must be made to ensure that the circuit will be free of critical races.

Assign 0 to state a, and 1 to state b in the reduced state table.

y \ DG	00	01	11	10
0	0	0	1	0
1	1	0	1	1

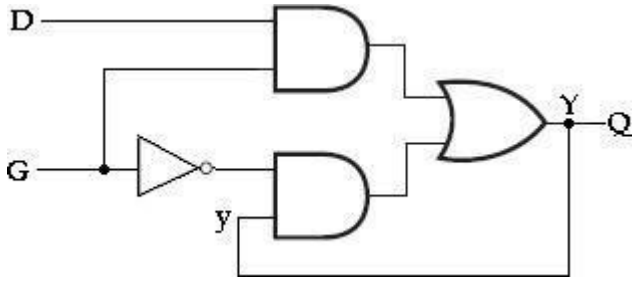
$Y = DG + \overline{G}y$

y \ DG	00	01	11	10
0	0	0	1	0
1	1	0	1	1

$Q = Y$

Transition table and output map

Step 5:



Gated-Latch Logic diagram

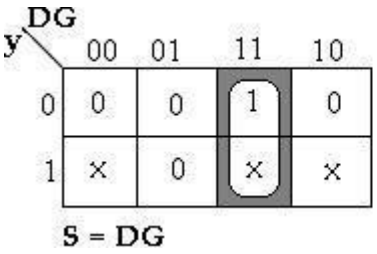
The diagram can be implemented also by means of an SR latch. Obtain the Boolean function for S and R inputs.

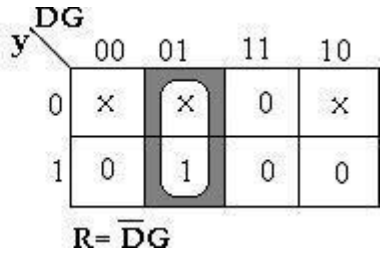
y	Y	S	R
0	0	0	x
0	1	1	0
1	0	0	1
1	1	x	0

SR Latch excitation table

From the information given in the transition table and from the latch excitation table conditions, we can obtain the maps for the S and R inputs of the latch.

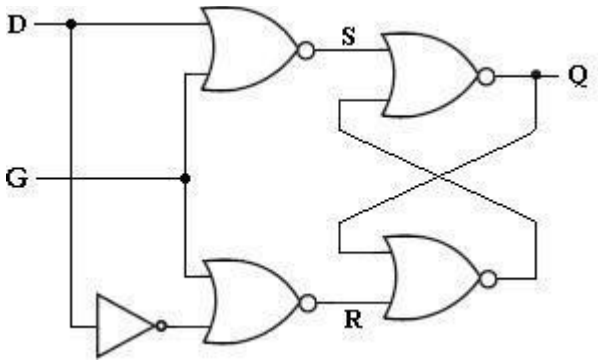
Maps for S and R



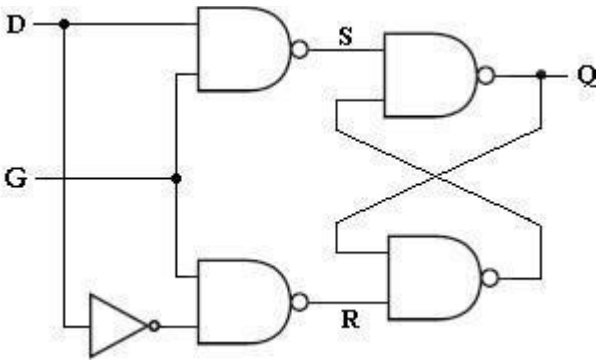


The logic diagram consists of an SR latch using NOR latch and the gates required to implement the S and R Boolean functions. With a NAND latch, we must use the complemented values for S and R.

S' = (DG)' and **R' = (D'G)'**



Logic diagram with NOR latch



Logic diagram with NAND latch

2. Design a negative-edge triggered T flip-flop. The circuit has two inputs, T (toggle) and G (clock), and one output, Q. the output state is complemented if T= 1 and the clock changes from 1 to 0 (negative-edge triggering). Otherwise, underany other input condition, the output Q remains unchanged.

Step 1:

Starting with the input condition TC= 11 and assign it to a. The circuit goes to state b and output Q complements from 0 to 1 when C changes from 1 to 0 while T remains a 1.

Another change in the output occurs when the circuit changes from state c to state d. In this case, T=1, C changes from 1 to 0, and the output Q complements from 1 to 0. The other four states in the table do not change the output, because T is equal to 0. If Q is initially 0, it stays at 0, and if initially at 1, it stays at 1 even though the clock input changes.

State	Inputs		Output	Comments
	T	G	Q	
a	1	1	0	Initial output is 0
b	1	0	1	After state a
c	1	1	1	Initial output is 1
d	1	0	0	After state c
e	0	0	0	After state d or f
f	0	1	0	After state e or a
g	0	0	1	After state b or h
h	0	1	1	After state g or c

Specifications of total states

Step 2**: Merging of the flow table**

The information for the primitive flow table can be obtained directly from the condition listed in the above table. We first fill in one square in each row belonging to stable state in that row as listed in the table.

Then we enter dashes in those squares whose input differs by two variables from the input corresponding to the stable state.

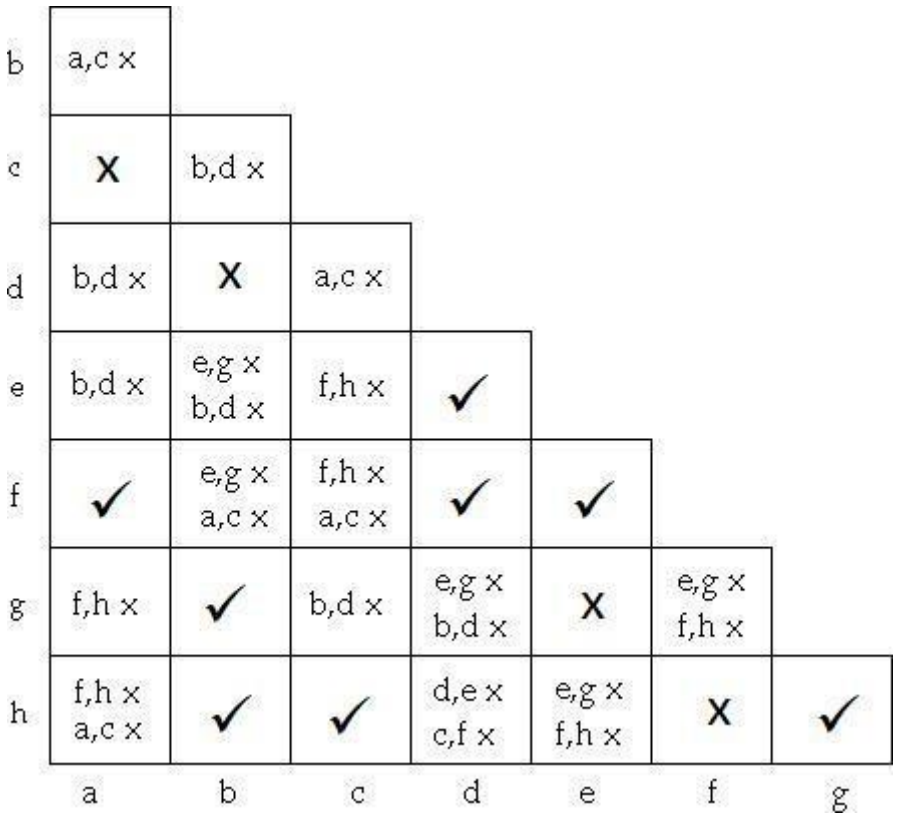
The unstable conditions are then determined by utilizing the information listed under the comments in the above table.

Step 3: Compatible pairs

	TC 00	01	11	10
a	–, –	f, –	(a), 0	b, –
b	g, –	–, –	c, –	(b), 1
c	–, –	h, –	(c), 1	d, –
d	e, –	–, –	a, –	(d), 0
e	(e), 0	f, –	–, –	d, –
f	e, –	(f), 0	a, –	–, –
g	(g), 1	h, –	–, –	b, –
h	g, –	(h), 1	c, –	–, –

Primitive flow table

The rows in the primitive flow table are merged by first obtaining all compatible pairs of states. This is done by means of the implication table.



Implication table

The implication table is used to find the compatible states. The only difference is that when comparing rows, we are at liberty to adjust the dashes to fit any desired condition. The two states are compatible if in every column of the corresponding rows in the primitive flow table, there are identical or compatible pairs and if there is no conflict in the output values.

A check mark ( ) designates a square whose pair of states is compatible. Those states that are not compatible are marked with a cross (x). The remaining squares are recorded with the implied pairs that need further investigation.

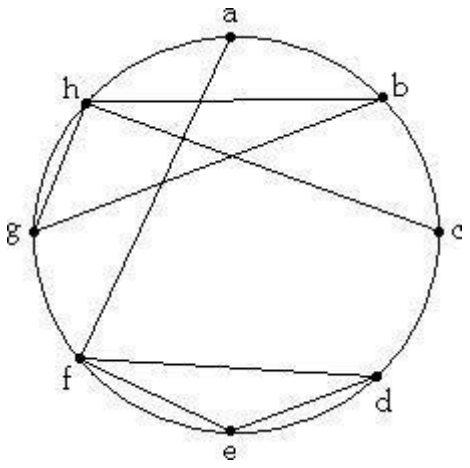
The squares that contain the check marks define the compatible pairs: (a, f) (b, g) (b, h) (c, h) (d, e) (d, f) (e, f) (g, h)

Step 4: Maximal compatibles

Having found all the compatible pairs, the next step is to find larger set of states that are compatible. The ***maximal compatible*** is a group of compatibles that contain all the possible combinations of compatible states. The maximal compatible can be obtained from a merger diagram.

The **merger diagram** is a graph in which each state is represented by a dot placed along the circumference of a circle. Lines are drawn between any two corresponding dots that form a compatible pair. All possible compatibles can be obtained from the merger diagram by observing the geometrical patterns in which states are connected to each other.

- A line represents a compatible pair
- A triangle constitutes a compatible with three states
- An n-state compatible is represented in the merger diagram by an n-sided polygon with all its diagonals connected.



Merger Diagram

The merger diagram is obtained from the list of compatible pairs derived from the implication table. There are eight straight lines connecting the dots, one foreach compatible pair. The lines form a geometrical pattern consisting of two triangles connecting (b, g, h) & (d, e, f) and two lines (a, f) & (c, h). The maximal compatibles are:

(a, f) (b, g, h) (c, h) (d, e, f)

TC

	00	01	11	10
a, f	e, –	(f), 0	(a), 0	b, –
b, g, h	(g), 1	(h), 1	c, –	(b), 1
c, h	g, –	(h), 1	(c), 1	d, –
d, e, f	(e), 0	(f), 0	a, –	(d), 0

Reduced Flow table

The reduced flow table is drawn. The compatible states are merged into one row that retains the original letter symbols of the states. The four compatible set of states are used to merge the flow table into four rows.

	TC 00	01	11	10
a	d, –	(a), 0	(a), 0	b, –
b	(b), 1	(b), 1	c, –	(b), 1
c	b, –	(c), 1	(c), 1	d, –
d	(d), 0	(d), 0	a, –	(d), 0

Final Reduced Flow table

Here we assign a common letter symbol to all the stable states in each merged row. Thus, the symbol f is replaced by a; g & h are replaced by b, and similarly for the other two rows.

Step 5: State Assignment and Transition table

Find the race-free binary assignment for the four stable states in the reduced flow table. Assign a= 00, b= 01, c= 11 and d= 10.

Substituting the binary assignment into the reduced flow table, the transition table is obtained. The output map is obtained from the reduced flow table.

Transition Table and Output Map

y_1y_2 \ TC	00	01	11	10
00	10	(00)	(00)	01
01	(01)	(01)	11	(01)
11	01	(11)	(11)	10
10	(10)	(10)	00	(10)

Transition table

y_1y_2 \ TC	00	01	11	10
00	0	0	0	X
01	1	1	1	1
11	1	1	1	X
10	0	0	0	0

Output map Q= y_2

Logic Diagram:

y_1y_2 \ TC	00	01	11	10
00	1	0	0	0
01	0	0	1	0
11	0	X	X	X
10	X	X	0	X

$S_1 = y_2TC + y'_2T'C'$

y_1y_2 \ TC	00	01	11	10
00	0	X	X	X
01	X	X	0	X
11	1	0	0	0
10	0	0	1	0

$R_1 = y_2T'C' + y_2'TC$

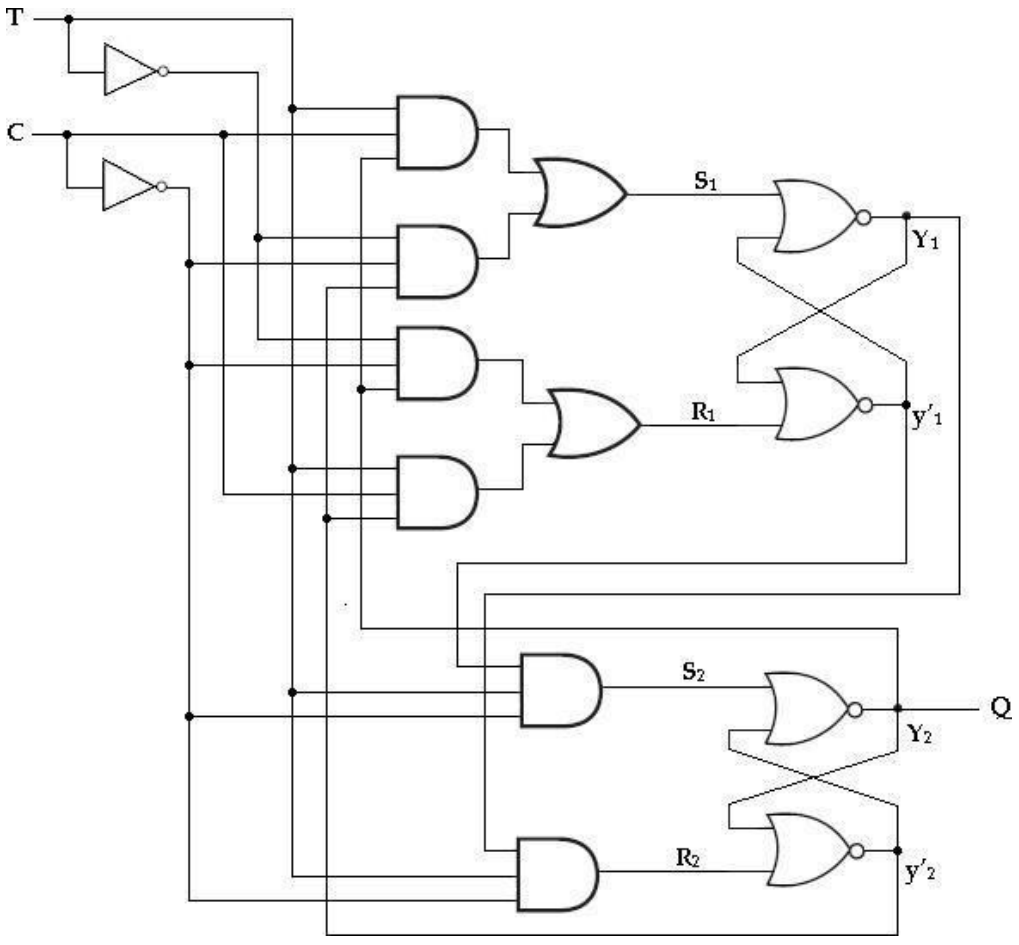
y_1y_2 \ TC	00	01	11	10
00	0	0	0	1
01	X	X	X	X
11	X	X	X	0
10	0	0	0	0

$S_2 = y'_1TC'$

y_1y_2 \ TC	00	01	11	10
00	X	X	X	0
01	0	0	0	0
11	0	0	0	1
10	X	X	X	X

$R_2 = y_1TC'$

Maps for Latch Inputs



3. Develop a state diagram and primitive flow table for a logic system that has two inputs, X and Y, and a single output X, which is to behave in the following manner. Initially, both inputs and output are equal to 0. Whenever X= 1 and Y= 0, the Z becomes 1 and whenever X= 0 and Y= 1, the Z becomes 0. When inputs are zero, i.e. X= Y= 0 or inputs are one, i.e. X= Y= 1, the output Z does not change; it remains in the previous state. The logic system has edge triggered inputs withouthaving a clock. The logic system changes state on the rising edges of the two inputs. Static input values are not to have any effect in changing the Z
output.

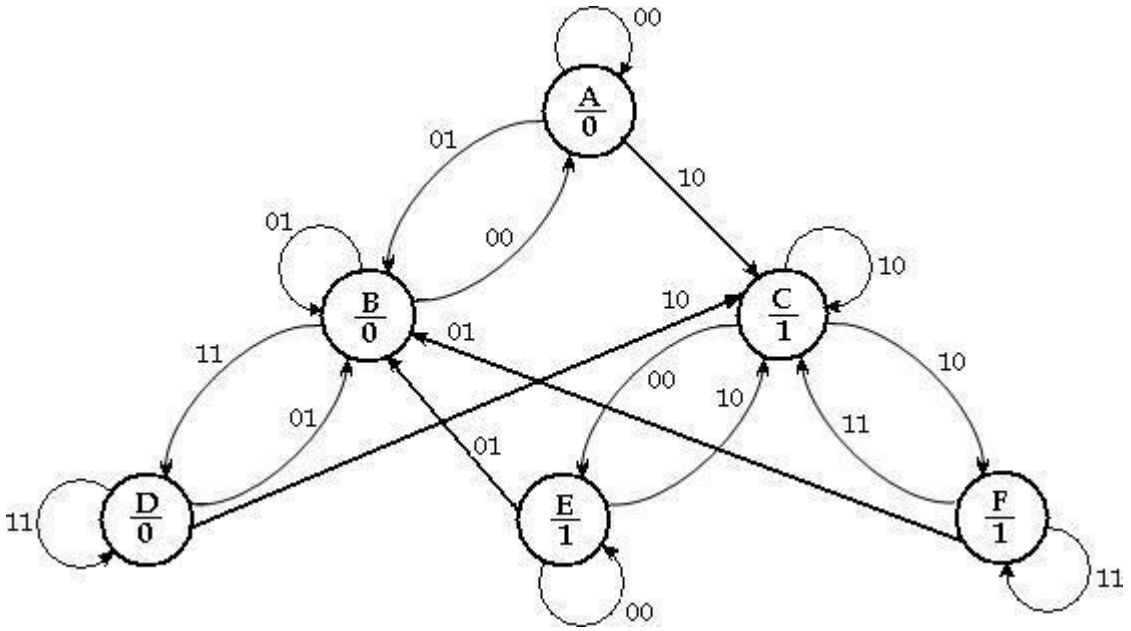
Soln:

The conditions given are,

- Initially both inputs X and Y are 0.
- When X= 1, Y= 0; Z= 1
- When X= 0, Y= 1; Z= 0
- When X= Y= 0 or X= Y= 1, then Z does not change, it remains in the previous state.

Step 1:

The above state transitions are represented in the state diagram as,



State diagram

Step 2:

A primitive flow table is constructed from the state diagram. The primitive flow table has one row for each state and one column for each input combination. Only one stable state exists for each row in the table. The stable state can be easily identified from the state diagram. For example, state A is stable with output 0 when inputs are 00, state C is stable with output 1 when inputs are 10 and so on.

We know that both inputs are not allowed to change simultaneously, so we can enter dash marks in each row that differs in two or more variables from the input variables associated with the stable state. For example, the first row in the flowtable shows a stable state with an input of 00. Since only one input can change at anygiven time, it can change to 01 or 10, but not to 11. Therefore we can enter two dashes in the 11 column of row A.

The remaining places in the primitive flow table can be filled by observing state diagram. For example, state B is the next state for present state A when input combination is 01; similarly state C is the next state for present state A when input combination is 10.

	XY 00	01	11	10
A	(A), 0	B, –	–, –	C, –
B	A, –	(B), 0	D, –	–, –
C	E, –	–, –	F, –	(C), 1
D	–, –	B, –	(D), 0	C, –
E	(E), 1	B, –	–, –	C, –
F	–, –	B, –	(F), 1	C, –

Primitive flow table

Step 3:

The rows in the primitive flow table are merged by first obtaining all compatible pairs of states. This is done by means of the implication table.

	A	B	C	D	E
B	✓				
C	A,E ×	A,E × D,F ×			
D	✓	✓	D,F ×		
E	A,E ×	A,E ×	✓	✓	
F	✓	D,F ×	✓	D,F ×	✓

The squares that contain the check marks (✓) define the compatible pairs:

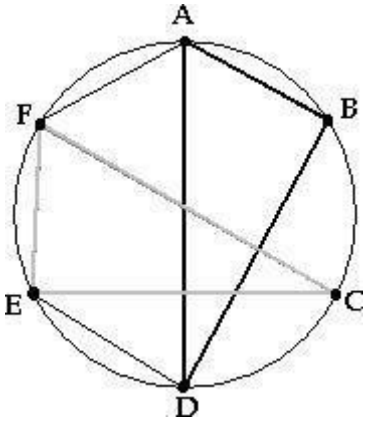
(A, B) (A, D) (A, F) (B, D) (C, E) (C, F) (D, E) (E, F)

Step 4:

The merger diagram is obtained from the list of compatible pairs derived from the implication table. There are eight straight lines connecting the dots, one for each compatible pair. The lines form a geometrical pattern consisting of two triangles connecting (A, B, D) & (C, E, F) and two lines (A, F) & (D, E). The maximal compatibles are:

(A, B, D) **(C, E, F)** **(A, F)** **(D, E)**

Closed covering condition:



Merger diagram

The condition that must be satisfied for merging rows is that the set of chosen compatibles must *cover* all the states and must be *closed.* The set will cover all the states if it includes all the states of the original state table. The closure condition is

satisfied if there are no implied states *or* if the implied states are included within the set. A closed set of compatibles that covers all the states is called a *closed covering.*

If we remove (A, F*)* and (D, E)*,* we are left with a set of two compatibles:

(A, B, D) (C, E, F)

All six states from the primitive flow table are included in this set. Thus, the set satisfies the covering condition.

The reduced flow table is drawn as below.

	XY 00	01	11	10
A,B,D	(A),0	(B),0	(D),0	C,–
C,E,F	(E),1	B,–	(F),1	(C),1

Reduced flow table

Here we assign a common letter symbol to all the stable states in each merged row. Thus, the symbol B & D is replaced by A; E & F are replaced by C.

	XY 00	01	11	10
A	(A),0	(A),0	(A),0	C,–
C	(C),1	A,–	(C),1	(C),1

Step 5:

Find the race-free binary assignment for the four stable states in the reduced flow table. Assign A= 0 and C= 1

Substituting the binary assignment into the reduced flow table, the transition table is obtained. The output map is obtained from the reduced flow table.

q \ XY	00	01	11	10
0	0	0	0	1
1	1	0	1	1

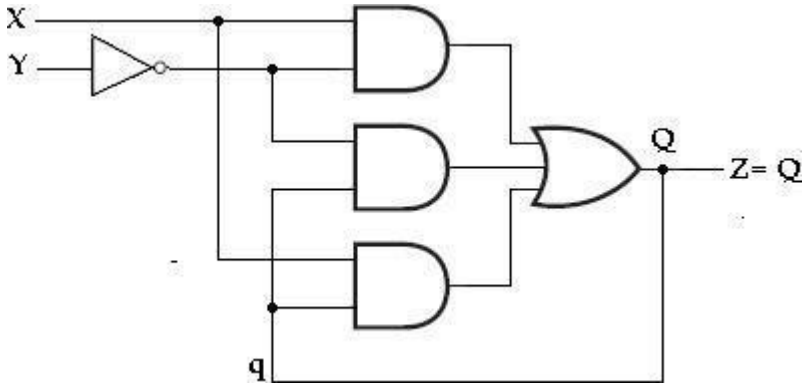
$Q = qY' + XY' + qX$

q \ XY	00	01	11	10
0	0	0	0	1
1	1	0	1	1

$Z = Q$

Transition table and output map

Step 6:



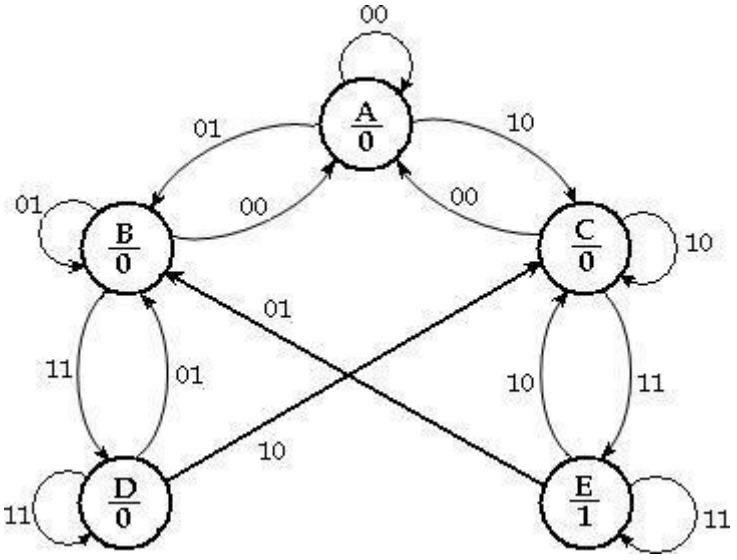
Gated-Latch Logic diagram

4. Design a circuit with inputs X and Y to give an output Z= 1 when XY= 11 but only if X becomes 1 before Y, by drawing total state diagram, primitive flow table and output map in which transient state is included.

Soln:

Step 1:

The state diagram can be drawn as,



State table

Step 2:

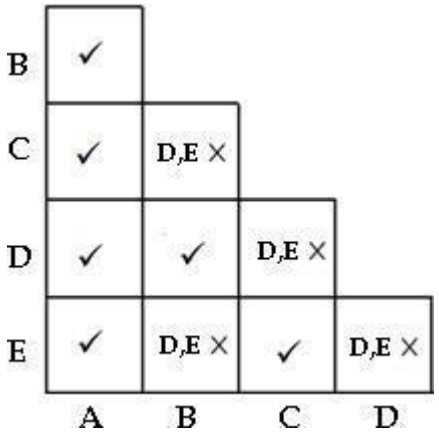
A primitive flow table is constructed from the state table as,

	XY 00	01	11	10
A	(A), 0	B, –	–, –	C, –
B	A, –	(B), 0	D, –	–, –
C	A, –	–, –	E, –	(C), 0
D	–, –	B, –	(D), 0	C, –
E	–, –	B, –	(E), 1	C, –

Primitive flow table

Step 3:

The rows in the primitive flow table are merged by first obtaining all compatible pairs of states. This is done by means of the implication table.



Implication table

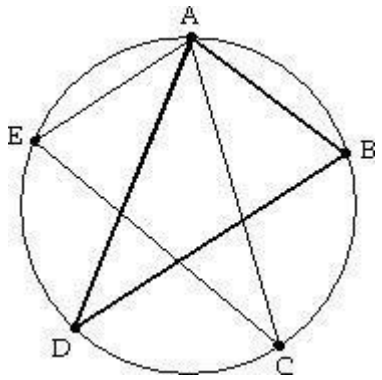
The squares that contain the check marks (✓) define the compatible pairs:

(A, B) (A, C) (A, D) (A, E) (B, D) (C, E)

Step 4:

The merger diagram is obtained from the list of compatible pairs derived from the implication table. There are six straight lines connecting the dots, one for each compatible pair. The lines form a geometrical pattern consisting of one triangle connecting (A, B, D) & a line (C, E). The maximal compatibles are:

(A, B, D) **(C, E)**



Merger diagram

The reduced flow table is drawn as below.

	XY 00	01	11	10
A, B, D	(A), 0	(B), 0	(D), 0	C, –
C, E	A, –	B, –	(E), 1	(C), 0

Reduced flow table

Here we assign a common letter symbol to all the stable states in each merged row. Thus, the symbol B & D is replaced by A; E is replaced by C.

	XY 00	01	11	10
A	(A), 0	(A), 0	(A), 0	C, –
C	A, –	A, –	(C), 1	(C), 0

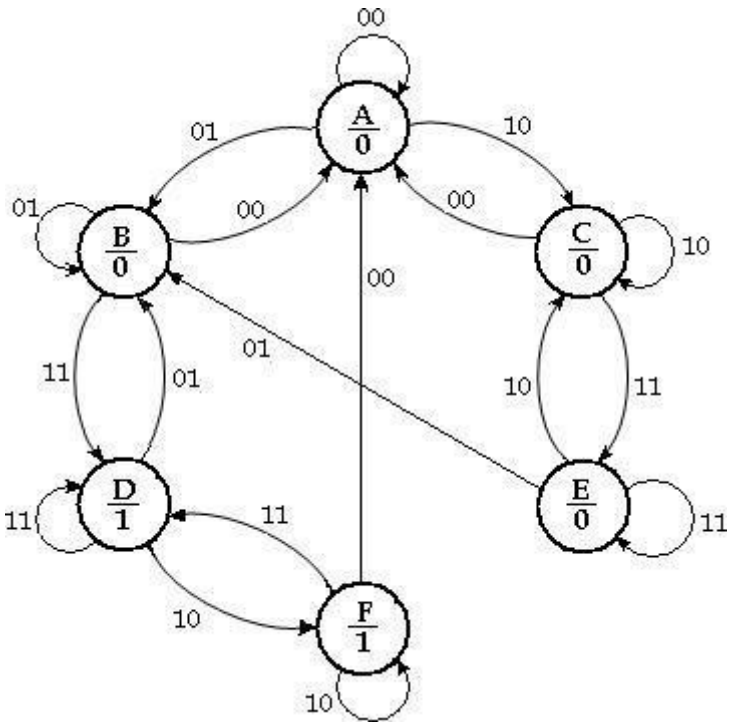
Transition table

5. Design a circuit with primary inputs A and B to give an output Z equal to 1 when A becomes 1 if B is already 1. Once Z= 1 it will remain so until A goes to 0. Draw the total state diagram, primitive flow table for designing this circuit.

Soln:

Step 1:

The state diagram can be drawn as,



State diagram

Step 2:

A primitive flow table is constructed from the state table as,

	AB 00	01	11	10
A	(A), 0	B, –	–, –	C, –
B	A, –	(B), 0	D, –	–, –
C	A, –	–, –	E, –	(C), 0
D	–, –	B, –	(D), 1	F, –
E	–, –	B, –	(E), 0	C, –
F	A, –	–, –	D, –	(F), 1

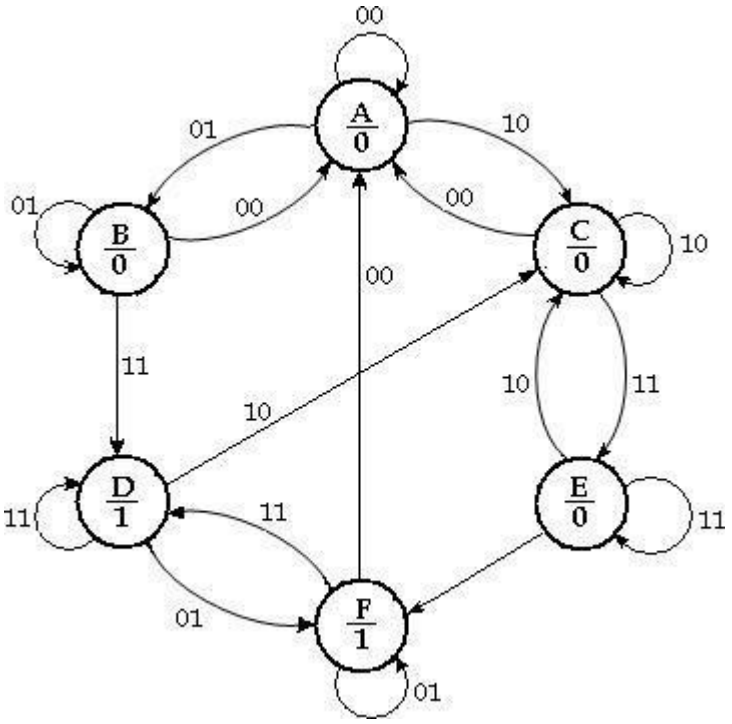
Primitive flow table

6. Design an asynchronous sequential circuit that has two inputs X_2 and X_1 and one output Z. When $X_1 = 0$, the output Z is 0. The first change in X_2 that occurs while X_1 is 1 will cause output Z to be 1. The output Z will remain 1 until X_1 returns to 0.

Soln:

Step 1:

The state diagram can be drawn as,



State diagram

Step 2:

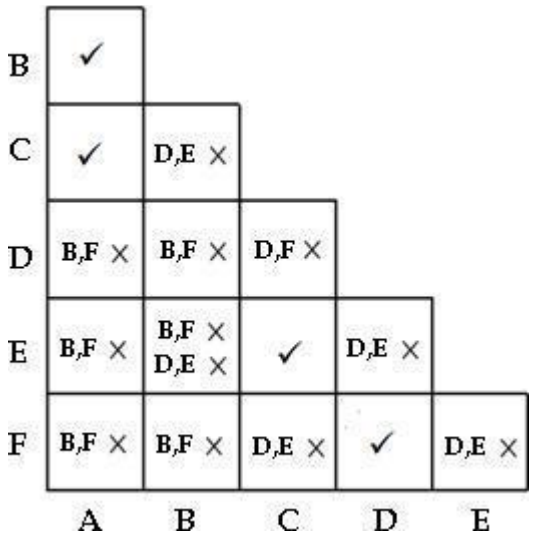
A primitive flow table is constructed from the state table as,

	X_2X_1 = 00	01	11	10
A	(A), 0	B, –	–, –	C, –
B	A, –	(B), 0	D, –	–, –
C	A, –	–, –	E, –	(C), 0
D	–, –	F, –	(D), 1	C, –
E	–, –	F, –	(E), 0	C, –
F	A, –	(F), 1	D, –	–, –

Primitive flow table

Step 3:

The rows in the primitive flow table are merged by obtaining all compatible pairs of states. This is done by means of the implication table.



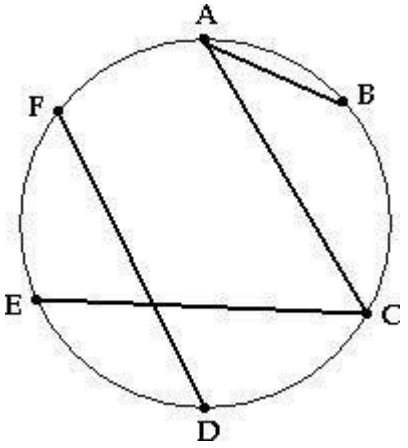
Implication table

The squares that contain the check marks (✓) define the compatible pairs:

(A, B) (A, C) (C, E) (D, F)

Step 4:

The merger diagram is obtained from the list of compatible pairs derived from the implication table. There are four straight lines connecting the dots, one for each compatible pair. It consists of four lines (A, B), (A, C), (C, E) and (D, F).



Merger diagram

The maximal compatibles are:

(A, B) (C, E) (D, F)

This set of maximal compatible covers all the original states resulting in the reduced flow table.

The reduced flow table is drawn as below.

	X_2X_1 00	01	11	10
A, B	(A), 0	(B), 0	D, –	C, –
C, E	A, –	F, –	(E), 0	(C), 0
D, F	A, –	(F), 1	(D), 1	C, –

Flow table

Here we assign a common letter symbol to all the stable states in each merged row. Thus, the symbol B is replaced by A; E is replaced by C and F is replaced by D.

Step 5:

	X_2X_1 00	01	11	10
A	(A), 0	(A), 0	D, –	C, –
C	A, –	D, –	(C), 0	(C), 0
D	A, –	(D), 1	(D), 1	C, –

Reduced Flow table

Find the race-free binary assignment for the four stable states in the reduced flow table. Assign A= S_0, C= S_1 and D= S_2.

	X_2X_1 00	01	11	10
S_0	(S_0), 0	(S_0), 0	S_2, –	S_1, –
S_1	S_0, –	S_2, –	(S_1), 0	(S_1), 0
S_2	S_0, –	(S_2), 1	(S_2), 1	S_1, –

Now, if we assign S_0= 00, S_1 = 01 and S_2 = 10, then we need one more state S_3= 11 to prevent critical race during transition of $S_0 \longrightarrow S_1$ or $S_2 \longrightarrow S_1$. By introducing S_3 the transitions $S_1 \longrightarrow S_2$ and $S_2 \longrightarrow S_1$ are routed through S_4.

Thus after state assignment the flow table can be given as,

Present State	Next state for Inputs X_2X_1, Output			
F_2F_1	00	01	11	10
$S_0 \rightarrow 00$	(S0), 0	(S0), 0	S2, –	S1, –
$S_1 \rightarrow 01$	S0, –	S3, –	(S1), 0	(S1), 0
$S_2 \rightarrow 10$	S0, –	(S2), 1	(S2), 1	S3, –
$S_3 \rightarrow 11$	–, –	S2, –	–, –	S1, –

Flow table with state assignment

Substituting the binary assignment into the reduced flow table, the transition table is obtained. The output map is obtained from the reduced flow table.

Present State		Next state for Inputs X_2X_1, Output			
F_2	F_1	00	01	11	10
0	0	(00), 0	(00), 0	10, –	01, –
0	1	00, –	11, –	(01), 0	(01), 0
1	0	00, –	(10), 1	(10), 1	11, –
1	1	–, –	10, –	–, –	01, –

K- Map simplification:

For F_2^+

F_2F_1 \ X_2X_1	00	01	11	10
00	0	0	1	0
01	0	1	0	0
11	X	1	X	0
10	0	1	1	1

$F_2^+ = \overline{F}_1X_2X_1 + F_1\overline{X}_2X_1 + F_2X_1 + F_2\overline{F}_1X_2$

For F_1^+

F_2F_1 \ X_2X_1	00	01	11	10
00	0	0	0	1
01	0	1	1	1
11	X	0	X	1
10	0	0	0	1

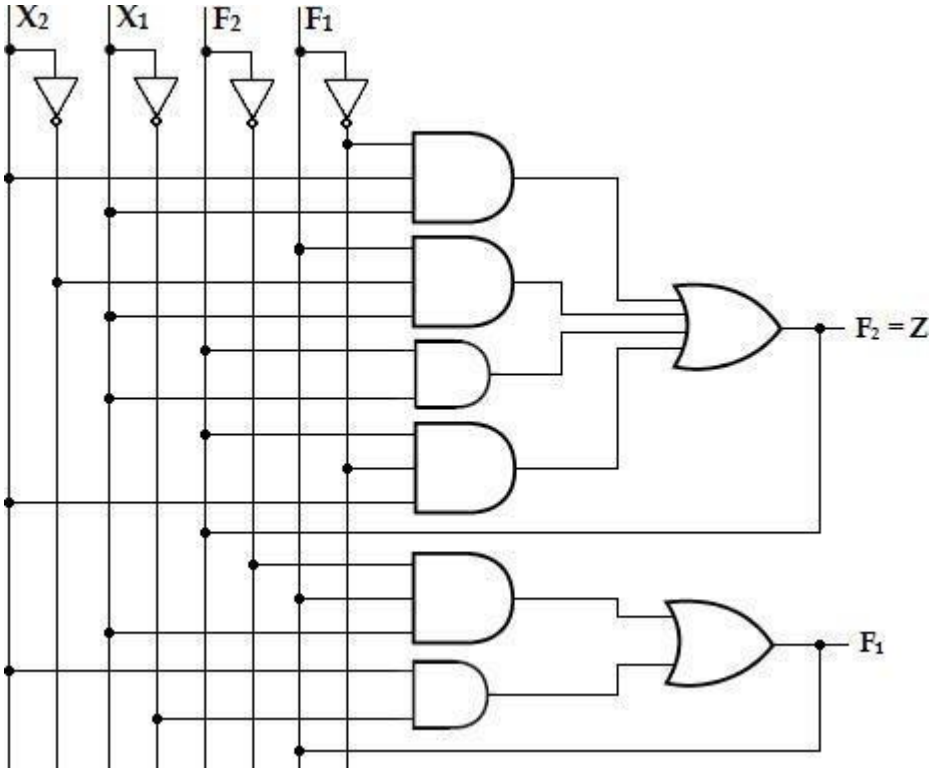
$F_1^+ = \overline{F}_2F_1X_1 + X_2\overline{X}_1$

For Z

F_2F_1 \ X_2X_1	00	01	11	10
00	0	0	X	X
01	X	X	0	0
11	X	X	X	X
10	X	1	1	X

$Z = F_2$

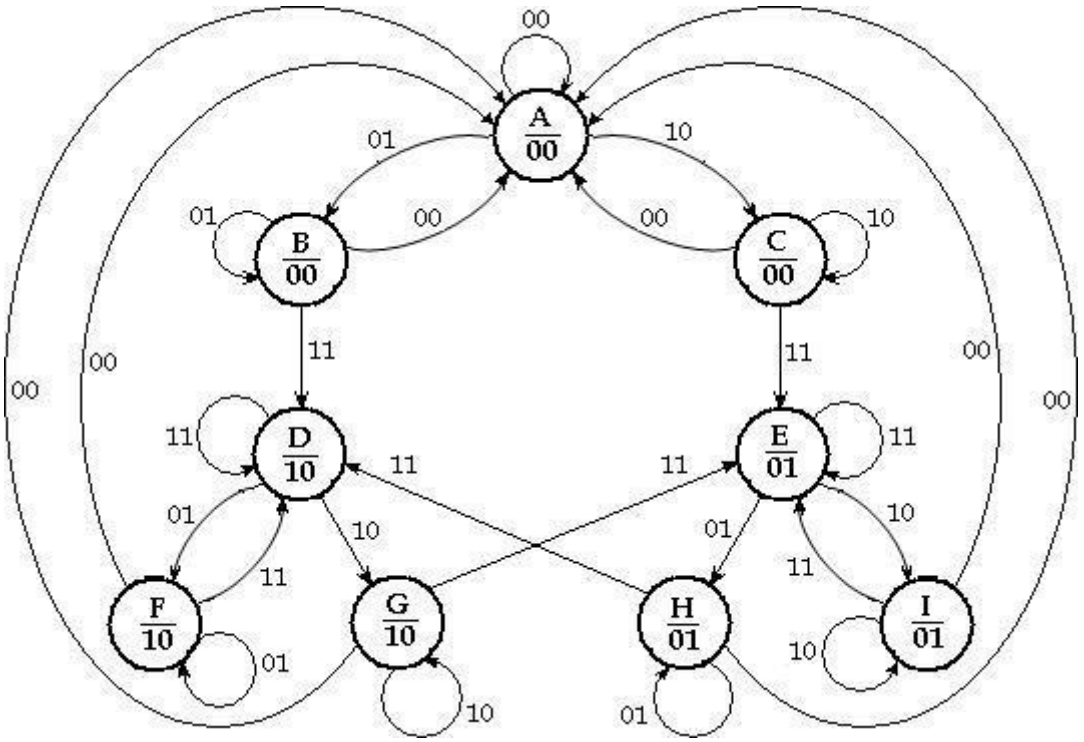
Logic Diagram:



7. Obtain a primitive flow table for a circuit with two inputs x_1 and x_2 and two outputs z_1 and z_2 that satisfies the following four conditions.
   i. When $x_1x_2 = 00$, output $z_1z_2 = 00$.
   ii. When $x_1 = 1$ and x_2 changes from 0 to 1, the output $z_1z_2 = 01$.
   iii. When $x_2 = 1$ and x_1 changes from 0 to 1, the output $z_1z_2 = 10$.
   iv. Otherwise the output does not change.

Soln:

The state diagram can be drawn as,



State diagram

Step 2: A primitive flow table is constructed from the state table as,

	X_1X_2 = 00	01	11	10
A	(A), 00	B, –	–, –	C, –
B	A, –	(B), 00	D, –	–, –
C	A, –	–, –	E, –	(C), 00
D	–, –	F, –	(D), 10	G, –
E	–, –	H, –	(E), 01	I, –
F	A, –	(F), 10	D, –	–, –
G	A, –	–, –	E, –	(G), 10
H	A, –	(H), 01	D, –	–, –
I	A, –	–, –	E, –	(I), 01

Primitive flow table

4.7 HAZARDS

Hazards are unwanted switching transients that may appear at the output of a circuit because different paths exhibit different propagation delays.

Hazards occur in combinational circuits, where they may cause a temporary false-output value. When this condition occurs in asynchronous sequential circuits, it may result in a transition to a wrong stable state.

Hazards in Combinational Circuits:

A hazard is a condition where a single variable change produces a momentary output change when no output change should occur.

Types of Hazards:

- Static hazard
- Dynamic hazard

4.7.1 Static Hazard

In digital systems, there are only two possible outputs, a '0' or a '1'. The hazard may produce a wrong '0' or a wrong '1'. Based on these observations, there are three types,

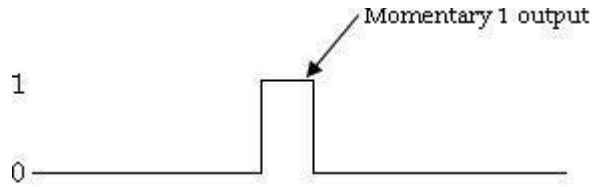
- Static- 0 hazard,
- Static- 1 hazard,

Static- 0 hazard:

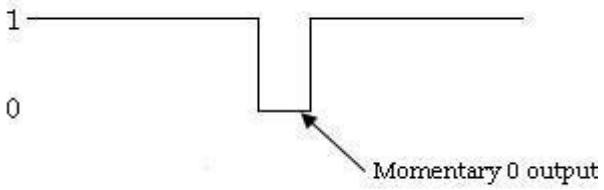
When the output of the circuit is to remain at 0, and a momentary 1 output is possible during the transmission between the two inputs, then the hazard is called a static 0-hazard.

Static- 1 hazard:

When the output of the circuit is to remain at 1, and a momentary 0 output is possible during the transmission between the two inputs, then the hazard is called a static 1-hazard.



(a) Static 0-hazard

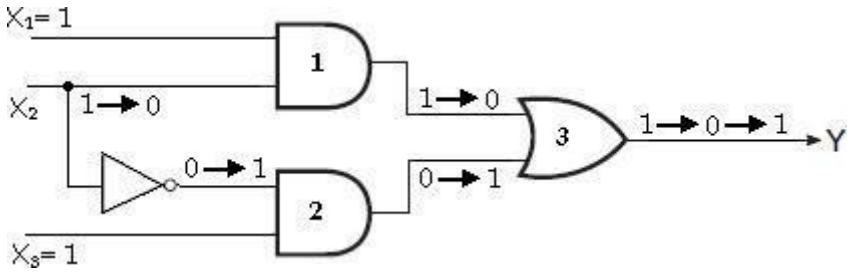


(b) Static 1-hazard

The below circuit demonstrates the occurrence of a static 1-hazard. Assume that all three inputs are initially equal to 1 i.e., $X_1X_2X_3$= 111. This causes the output of the gate 1 to be 1, that of gate 2 to be 0, and the output of the circuit to be equal to 1. Now consider a change of X_2 from 1 to 0 i.e., $X_1X_2X_3$= 101. The output of gate 1 changes to 0 and that of gate 2 changes to 1, leaving the output at 1. The output may momentarily go to 0 if the propagation delay through the inverter is taken into consideration.

The delay in the inverter may cause the output of gate 1 to change to 0 before the output of gate 2 changes to 1. In that case, both inputs of gate 3 are momentarily equal to 0, causing the output to go to 0 for the short interval of time that the input signal from X_2 is delayed while it is propagating through the inverter circuit.

Thus, a static 1-hazard exists during the transition between the input states $X_1X_2X_3$= 111 and $X_1X_2X_3$= 101.

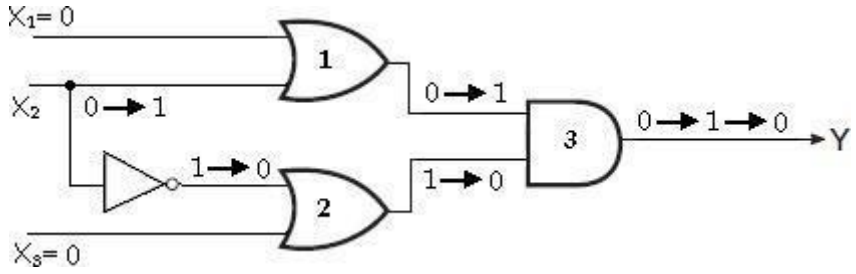


Circuit with static-1 hazard

Now consider the below network, and assume that the inverter has an appreciably greater propagation delay time than the other gates. In this case there is a static 0-hazard in the transition between the input states $X_1X_2X_3$= 000 and $X_1X_2X_3$= 010 since it is possible for a logic-1 signal to appear at both input terminals of the AND gate for a short duration.

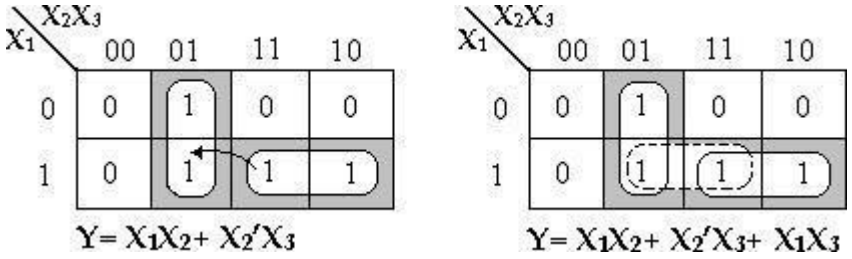
The delay in the inverter may cause the output of gate 1 to change to 1 before the output of gate 2 changes to 0. In that case, both inputs of gate 3 are momentarily equal to 0, causing the output to go to 1 for the short interval of time that the input signal from X_2 is delayed while it is propagating through the inverter circuit.

Thus, a static 0-hazard exists during the transition between the input states $X_1X_2X_3$= 000 and $X_1X_2X_3$= 010.



Circuit with static-0 hazard

A hazard can be detected by inspection of the map of the particular circuit. To illustrate, consider the map in the circuit with static 0-hazard, which is a plot of the function implemented. The change in X_2 from *1* to 0 moves the circuit from minterm 111 to minterm 101. The hazard exists because the change in input results in a different product term covering the two minterrns.



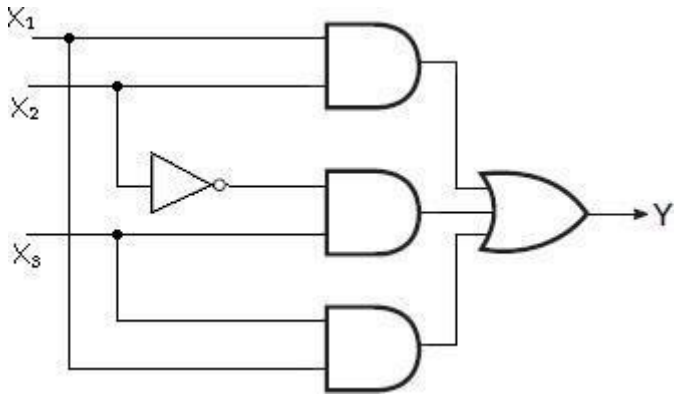
Maps demonstrating a Hazard and its Removal

The minterm 111 is covered by the product term implemented in gate 1 and minterm 101 is covered by the product term implemented in gate 2. Whenever the circuit must move from one product term to another, there is a possibility of a momentary interval when neither term is equal to 1, giving rise to an undesirable 0 output.

The remedy for eliminating a hazard is to enclose the two minterms in question with another product term that overlaps both groupings. This situation is

shown in the *map* above, where the two terms that causes the hazard are combined into one product term. The hazard- free circuit obtained by this combinational is shown below.

Hazard-free Circuit

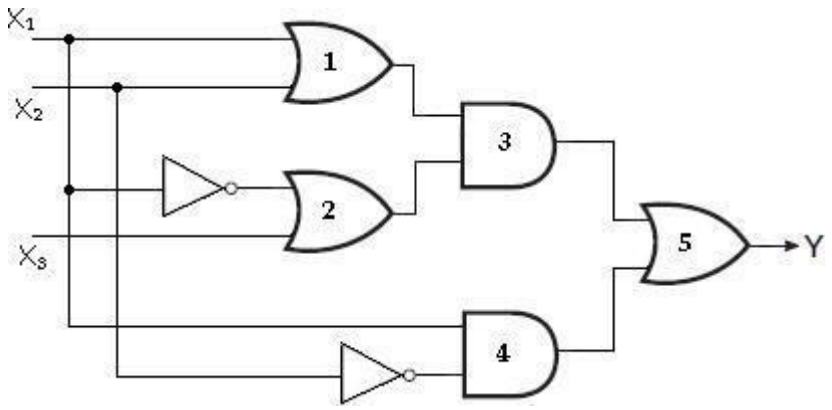


The extra gate in the circuit generates the product term X_1X_4. The hazards in combinational circuits can be removed by covering any two minterms that may produce a hazard with a product term common to both. The removal of hazards requires the addition of redundant gates to the circuit.

4.7.2 Dynamic Hazard

A dynamic hazard is defined as a transient change occurring three or more times at an output terminal of a logic network when the output is supposed to change only once during a transition between two input states differing in the valueof one variable.

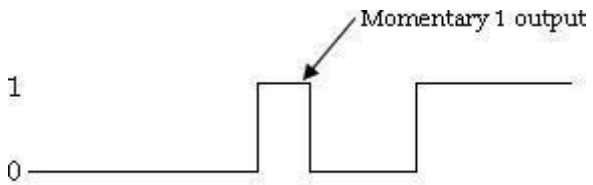
Now consider the input states $X_1X_2X_3$= 000 and $X_1X_2X_3$= 100. For the first input state, the steady state output is 0; while for the second input state, the steady state output is 1. To facilitate the discussion of the transient behavior of this network, assume there are no propagation delays through gates G_3 and G_5 and that the propagation delays of the other three gates are such that G_1 can switch faster than G_2 and G_2 can switch faster than G_4.



Circuit with Dynamic hazard

When X_1 changes from 0 to 1, the change propagates through gate G_1 before gate G_2 with the net effect that the inputs to gate G_3 are simultaneously 1 and the network output changes from 0 to 1. Then, when X_1 change propagates through gate G_2, the lower input to gate G_3 becomes 0 and the network output changes back to 0.

Finally, when the X_1= 1 signal propagates through gate G_4, the lower input to gate G_5 becomes 1 and the network output again changes to 1. It is therefore seen that during the change of X_1 variable from 0 to 1 the output undergoes the sequence, 0 → 1 → 0 → 1, which results in three changes when it should have undergoneonly a single change.



Dynamic hazard

4.7.3 Essential Hazard

An essential hazard is caused by unequal delays along two or more paths that originate from the same input. An excessive delay through an inverter circuit in comparison to the delay associated with the feedback path may cause such a hazard.

Essential hazards elimination:

Essential hazards can be eliminated by adjusting the amount of delays in the affected path. To avoid essential hazards, each feedback loop must be handled with individual care to ensure that the delay in the feedback path is long enough compared with delays of other signals that originate from the input terminals.

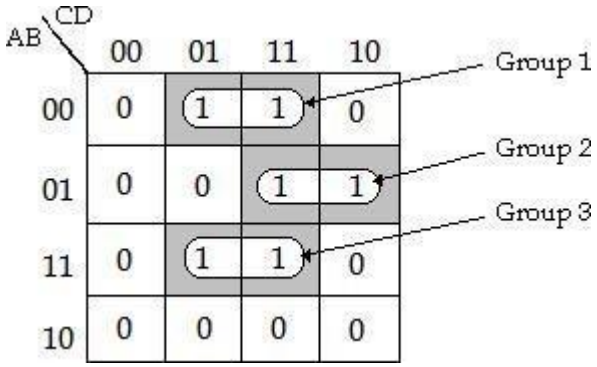
4.8 Design Of Hazard Free Circuits

1. Design a hazard-free circuit to implement the following function.

F (A, B, C, D) = ∑m (1, 3, 6, 7, 13, 15)

Soln:

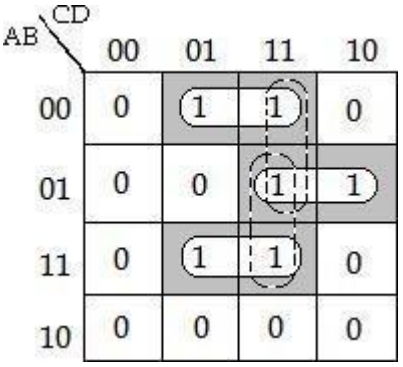
a) K-map Implementation and grouping



F=A'B'D+ A'BC+ ABD

b) Hazard- free realization

The first additional product term A'CD, overlapping two groups (group 1 & 2) and the second additional product term, BCD, overlapping the two groups (group 2 & 3).



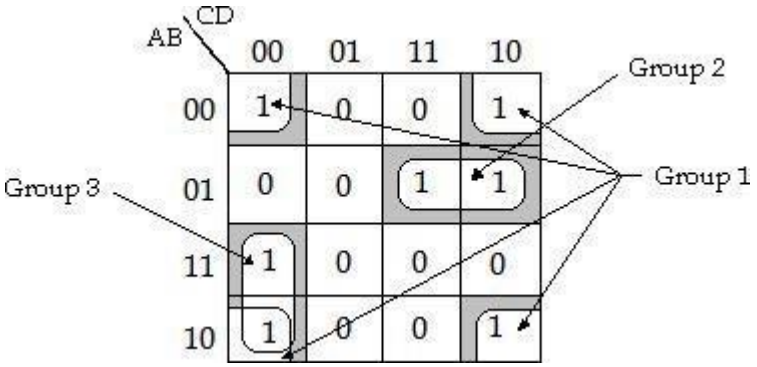
F=A'B'D+ A'BC+ ABD+ A'CD+ BCD

2. Design a hazard-free circuit to implement the following function.

F (A, B, C, D) = ∑m (0, 2, 6, 7, 8, 10, 12).

Soln:

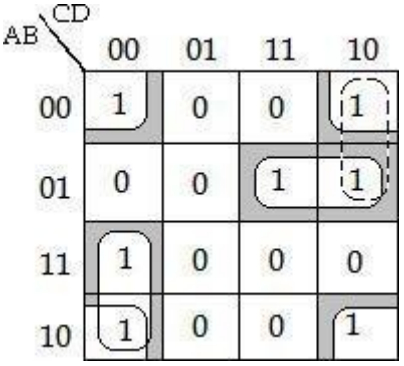
a) K-map Implementation and grouping



F= B'D'+ A'BC+ AC'D'

b) Hazard- free realization

The additional product term, A'CD' overlapping two groups (group 1 & 2) for hazard free realization. Group 1 and 3 are already overlapped hence they do not require additional minterm for grouping.

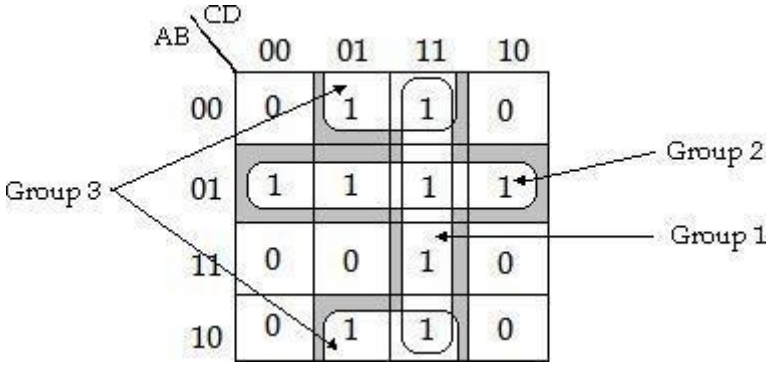


F= B'D'+ A'BC+ AC'D'+ A'CD'

3. Design a hazard-free circuit to implement the following function.

F (A, B, C, D) = ∑m (1, 3, 4, 5, 6, 7, 9, 11, 15).

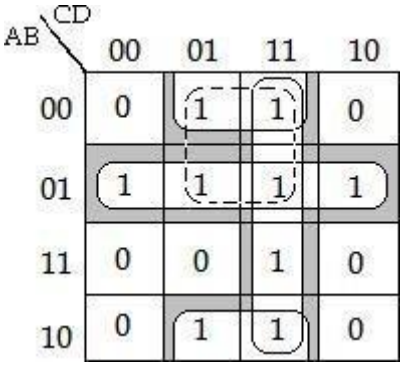
a) K-map Implementation and grouping



F= CD+ A'B+ B'D

b) Hazard- free realization

The additional product term, A'D overlapping two groups (group 2 & 3) for hazard free realization. Group 1 and 2 are already overlapped hence they do not require additional minterm for grouping.



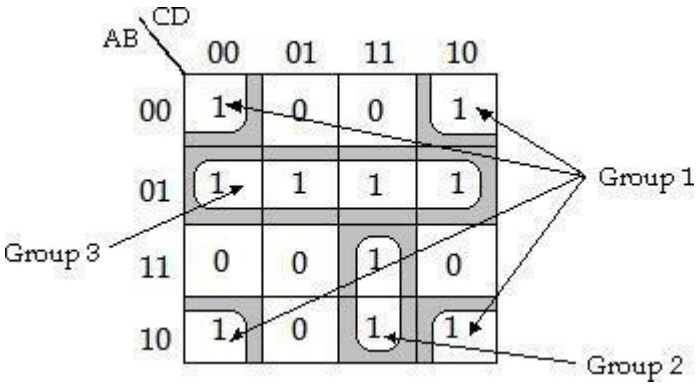
F= CD+ A'B+ B'D+ A'D

4. Design a hazard-free circuit to implement the following function.

F (A, B, C, D) = ∑m (0, 2, 4, 5, 6, 7, 8, 10, 11, 15).

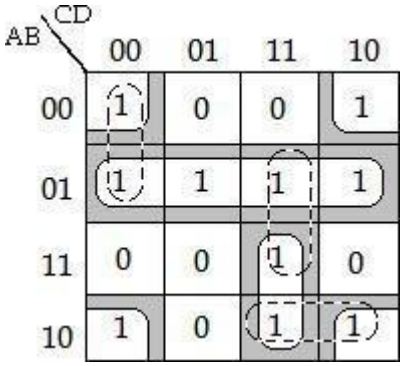
Soln:

a) K-map Implementation and grouping



F= B'D'+ A'B+ ACD

b) Hazard- free realization

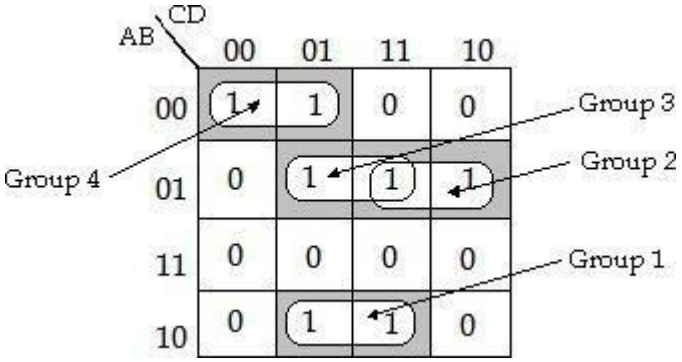


F= B'D'+ A'B+ ACD+ A'C'D'+ BCD+ AB'C

5. Design a hazard-free circuit to implement the following function.

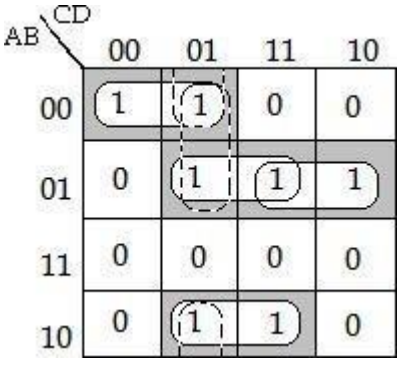
   F (A, B, C, D) = ∑m (0, 1, 5, 6, 7, 9, 11).

a) <u>K-map Implementation and grouping</u>



F= AB'D+ A'BC+ A'BD+ A'B'C'

b) <u>Hazard- free realization:</u>



F= AB'D+ A'BC+ A'BD+ A'B'C'+ A'C'D+ B'C'D

UNIT V LOGIC FAMILIES AND PROGRAMMABLE LOGIC DEVICES

5.1 INTRODUCTION

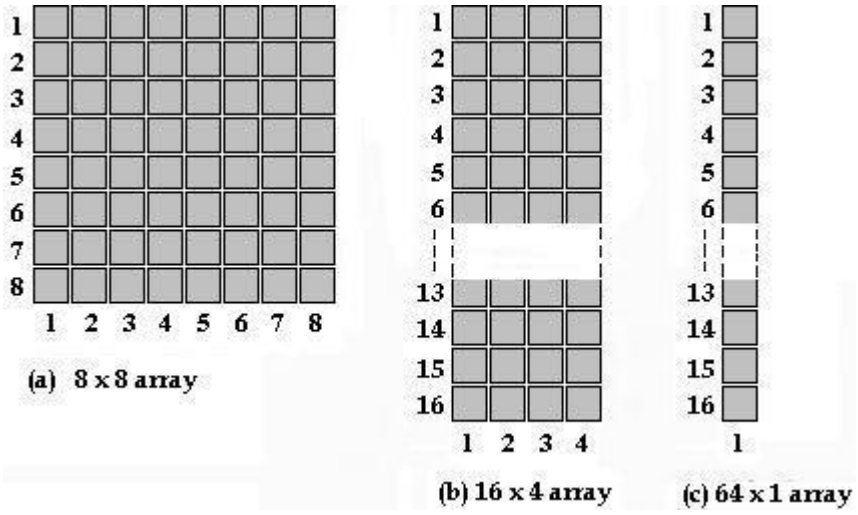
A memory unit is a collection of storage cells with associated circuits needed to transfer information in and out of the device. The binary information is transferredfor storage and from which information is available when needed for processing. When data processing takes place, information from the memory is transferred to selected registers in the processing unit. Intermediate and final results obtained in the processingunit are transferred back to be stored in memory.

5.2 Units of Binary Data: Bits, Bytes, Nibbles and Words

As a rule, memories store data in units that have from one to eight bits. The smallest unit of binary data is the **bit**. In many applications, data are handled in an 8-bit unit called a **byte** or in multiples of 8-bit units. The byte can be split into two 4-bit units that are called **nibbles**. A complete unit of information is called a **word** and generally consists of one or more bytes. Some memories store data in 9-bit groups; a 9-bit group consists of a byte plus a parity bit.

5.3 Basic Semiconductor Memory Array

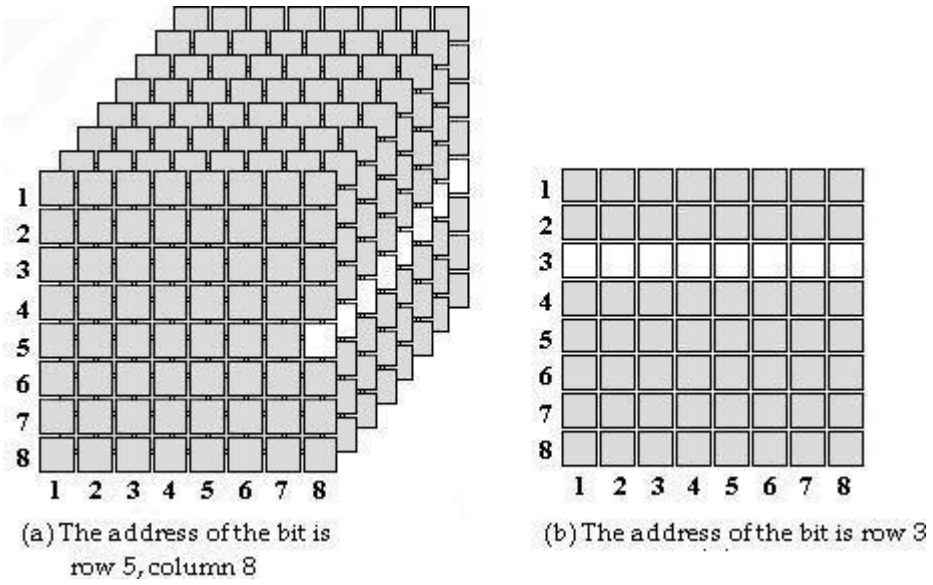
Each storage element in a memory can retain either a 1 or a 0 and is called a **cell**. Memories are made up of arrays of cells, as illustrated in Figure below using 64 cellsas an example. Each block in the memory array represents one storage cell, andits location can be identified by specifying a row and a column.



A 64-cell memory array organized in three different ways

5.4 Memory Address and Capacity

The *location* of a unit of data in a memory array is called its **address**. For example, in Figure (a), the address of a bit in the 3-dimensional array is specified by the row and column. In Figure (b), the address of a byte is specified only by the row in the 2-dimensional array. So, as you can see, the address depends on how the memory is organized into units of data. Personal computers have random-access memories organized in bytes. This means that the smallest group of bits that can be addressed is eight.



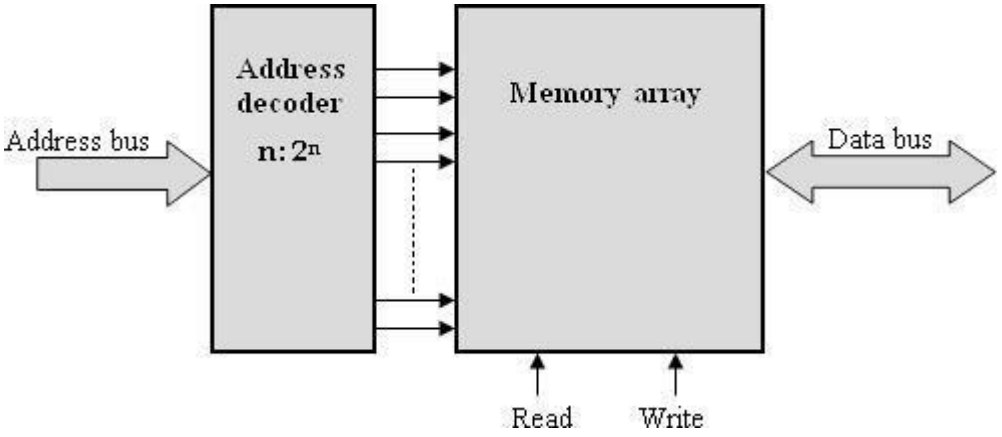
Examples of memory address

The **capacity** of a memory is the total number of data units that can be stored. For example, in the bit-organized memory array in Figure (a), the capacity is 64 bits. In the byte-organized memory array in Figure (b), the capacity is 8 bytes, which is also 64 bits. Computer memories typically have 256 MB (megabyte) or more of internal memory.

5.5 Basic Memory Operations

Since a memory stores binary data, data must be put into the memory and data must be copied from the memory when needed. The write operation puts data into a specified address in the memory, and the read operation copies data out of a specified address in the memory. The addressing operation, which is part of both thewrite and the read operations, selects the specified memory address.

Data units go into the memory during a write operation and come out of the memory during a read operation on a set of lines called the *data bus*. As indicated in Figure, the data bus is bidirectional, which means that data can go in either directional (into the memory or out of the memory).



Block diagram of memory operation

For a write or a read operation, an address is selected by placing a binary code representing the desired address on a set of lines called the address bus. The address code is decoded internally and the appropriate address is selected. The number oflines in the address bus depends on the capacity of the memory. For example, a 15-bit address code can select 32,768 locations (2_{15}) in the memory; a 16-bit address code can select 65,536 locations (2_{16}) in the memory and so on.

In personal computers a 32-bit address bus can select 4,294,967,296 locations (2_{32}), expressed as 4GB.

5.5.1 Write Operation

To store a byte of data in the memory, a code held in the address register is placed on the address bus. Once the address code is on the bus, the address decoder decodes the address and selects the specified location in the memory. The memory then gets a write command, and the data byte held in the data register is placed on the data bus and stored in the selected memory address, thus completing the write operation. When a new data byte is written into a memory address, the current data byte stored at that address is overwritten (replaced with a new data byte).

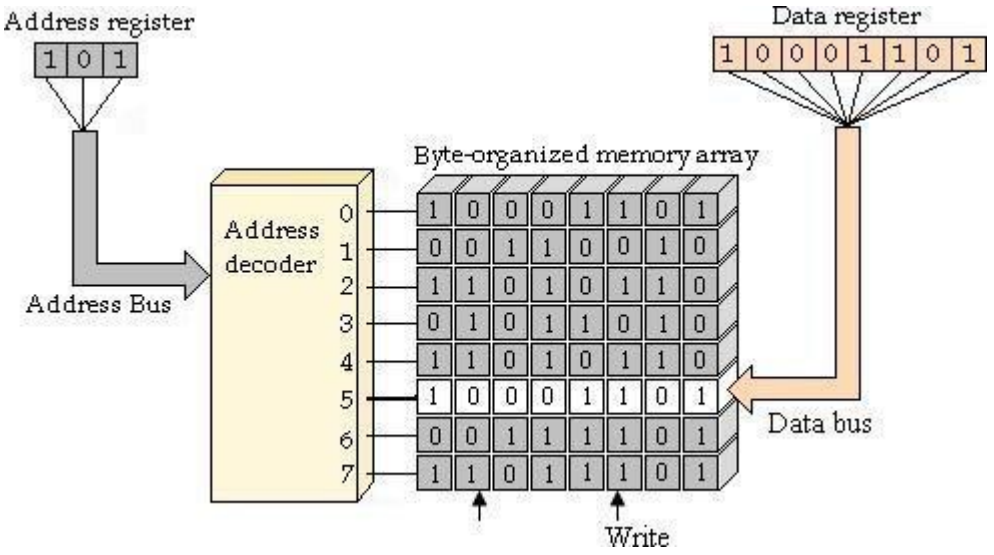


Illustration of the Write operation

5.5.2 Read Operation

A code held in the address register is placed on the address bus. Once the address code is on the bus, the address decoder decodes the address and selects the specified location in the memory. The memory then gets a read command, and a "copy" of the data byte that is stored in the selected memory address is placed on the data bus and loaded into the data register, thus completing the read operation. Whena data byte is read from a memory address, it also remains stored at that address. This is called *nondestructive* read.

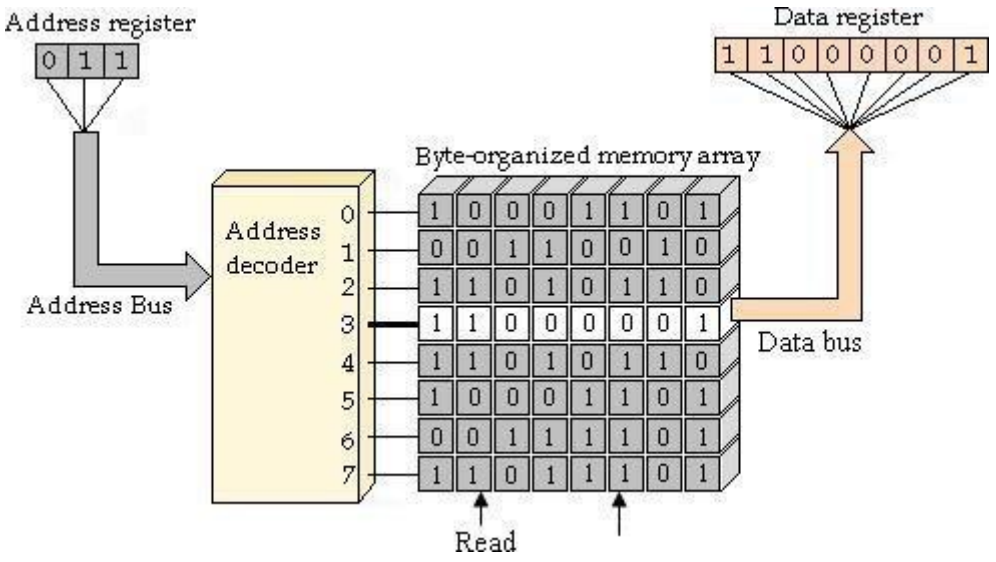


Illustration of the Read operation

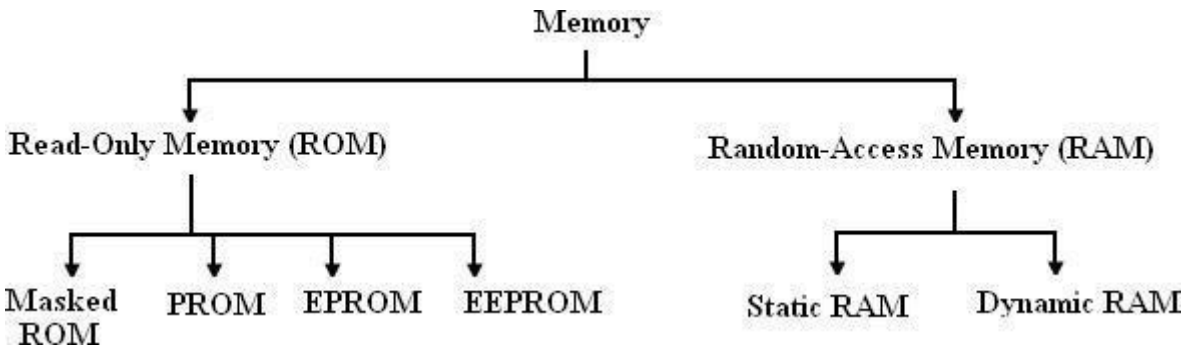
5.6 Classification of Memories

There are two types of memories that are used in digital systems:

- Random-Access Memory (RAM),
- Read-Only Memory (ROM).

RAM (random-access memory) is a type of memory in which all addresses are accessible in an equal amount of time and can be selected in any order for a read or write operation. All RAMs have both read and write capability. Because RAMs losestoreddata when the power is turned off, they are ***volatile*** memories.

ROM (read-only memory) is a type of memory in which data are stored permanently or semi permanently. Data can be read from a ROM, but there is no write operation as in the RAM. The ROM, like the RAM, is a random-access memorybut the term RAM traditionally means a random-access read/write memory. Because ROMs retain stored data even if power is turned off, they are ***nonvolatile*** memories.



Classification of memories

5.6.1 RANDOM-ACCESS MEMORIES (RAMS)

RAMs are read/write memories in which data can be written into or read from any selected address in any sequence. When a data unit is written into a given address in the RAM, the data unit previously stored at that address is replaced by the new data unit. When a data unit is read from a given address in the RAM, the data unit remains stored and is not erased by the read operation. This nondestructive read operation can be viewed as copying the content of an address while leaving the content intact.

A RAM is typically used for short-term data storage because it cannot retain stored data when power is turned off.

The two categories of RAM are the ***static* RAM** (SRAM) and the ***dynamic* RAM** (DRAM). Static RAMs generally use flip-flops as storage elements and can therefore store data indefinitely *as long as dc power is applied*. Dynamic RAMs use capacitors as storage elements and cannot retain data very long without the capacitors being recharged by a process called **refreshing**. Both SRAMs and DRAMs will lose stored data when dc power is removed and, therefore, are classified as ***volatile memories***.

Data can be read much faster from SRAMs than from DRAMs. However, DRAMs can store much more data than SRAMs for a given physical size and cost because the DRAM cell is much simpler, and more cells can be crammed into a givenchip area than in the SRAM.

5.6.1.1 Static RAM (SRAM)

Storage Cell:

All static RAMs are characterized by flip-flop memory cells. As long as dc power is applied to a static memory cell, it can retain a 1 or 0 state indefinitely. If power is removed, the stored data bit is lost.

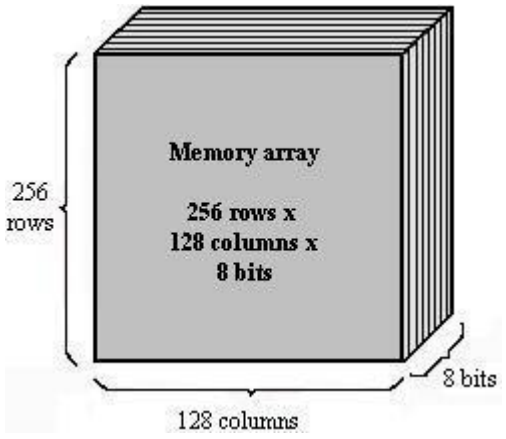
The cell is selected by an active level on the Select line and a data bit (l or 0) is written into the cell by placing it on the Data in line. A data bit is read by taking it off the Data out line.

Basic SRAM Organization:

Basic Static Memory Cell Array

The memory cells in a SRAM are organized in rows and columns. All the cells in a row share the same Row Select line. Each set of Data in and Data out lines go to each cell in a given column and are connected to a single data line that serves as both an input and output (Data I/O) through the data input and data output buffers.

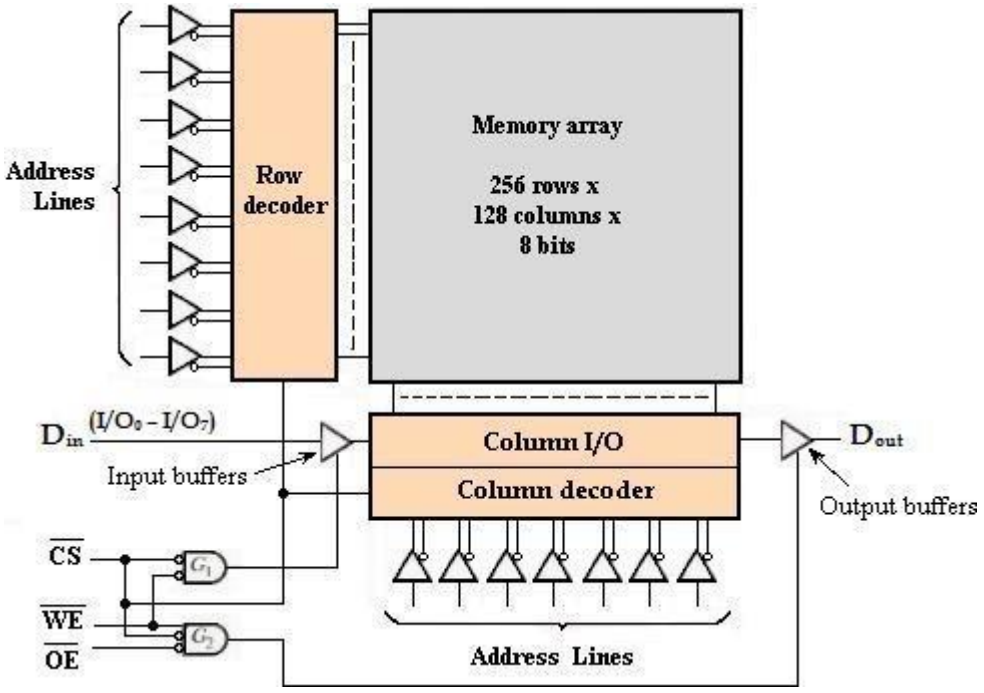
SRAM chips can be organized in single bits, nibbles (4 bits), bytes (8 bits), or multiple bytes (16, 24, 32 bits, etc.). The memory cell array is arranged in 256 rows and 128 columns, each with 8 bits as shown below. There are actually 2_{15} = 32,768 addresses and each address contains 8 bits. The capacity of this example memory is 32,768 bytes (typically expressed as 32 Kbytes).



Memory array configuration

Operation:

The SRAM works as follows. First, the chip select, CS, must be LOW for the memory to operate. Eight of the fifteen address lines are decoded by the row decoder to select one of the 256 rows. Seven of the fifteen address lines are decoded by the column decoder to select one of the 128 8-bit columns.



Memory block diagram

Read:

In the READ mode, the write enable input, WE‘ is HIGH and the output enable, OE_ is LOW. The input tri state buffers are disabled by gate **G1**, and the column output tristate buffers are enabled by gate **G2**. Therefore, the eight data bits from the selected address are routed through the column I/O to the data lines (I/O_1 through I/O_7), which are acting as data output lines.

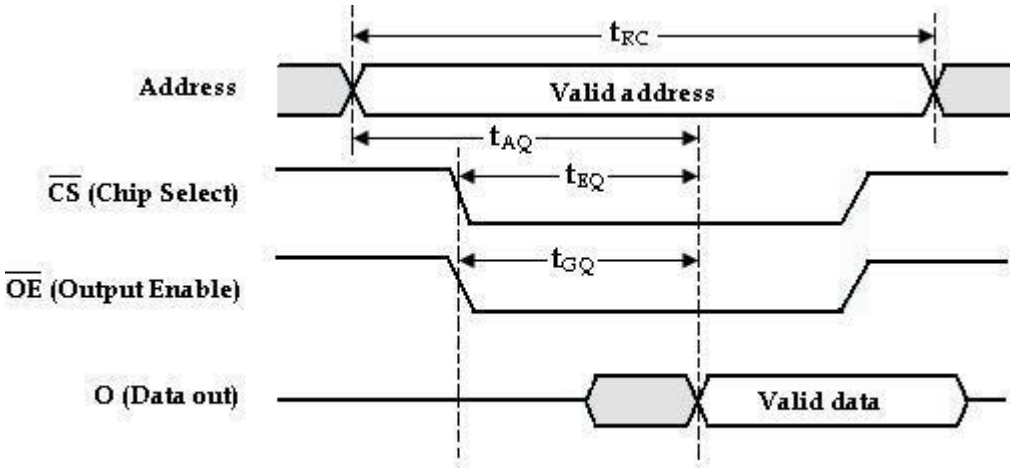
Write:

In the WRITE mode, WE‘ is LOW and OE‘ is HIGH. The input buffers are enabled by gate G_1, and the output buffers are disabled by gate G_2. Therefore the eight input data bits on the data lines are routed through the input data control and the column I/O to the selected address and stored.

Read and Write Cycles:

For the read cycle shown in part (a), a valid address code is applied to the address lines for a specified time interval called the ***read cycle time***, t_{WC}. Next, the chip select (CS) and the output enable (DE) inputs go LOW. One time interval after the DE input goes LOW; a valid data byte from the selected address appears on the data lines. This time interval is called the ***output enable access time***, t_{GQ}. Two other access times for the read cycle are the ***address access time***, t_{AQ}, measured from the beginning of a valid address to the appearance of valid data on the data lines and the **chip enable access time**, t_{EQ}, measured from the HIGH-to-LOW transition of CS to the appearance of valid data on the data lines.

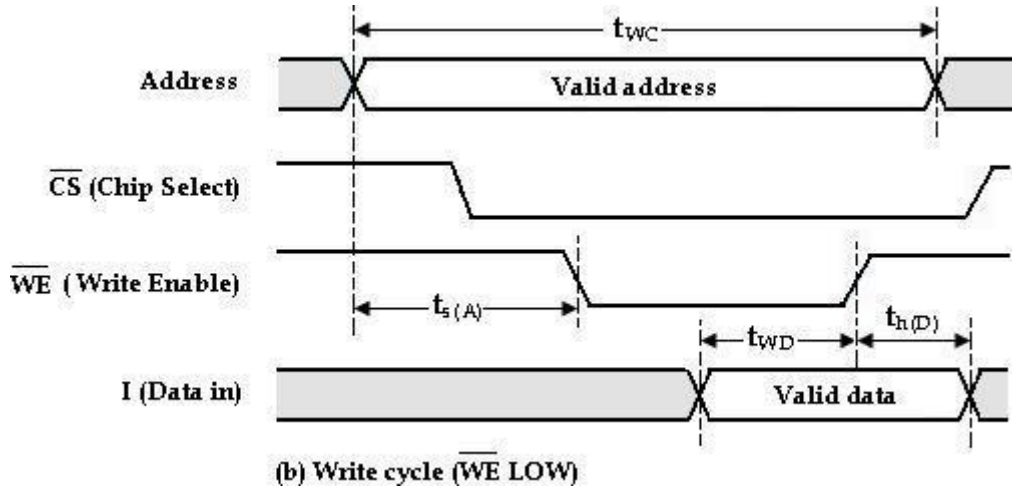
During each read cycle, one unit of data, a byte in this case is read from the memory.



(a) Read Cycle ($\overline{WE}$ HIGH)

For the write cycle shown in Figure (b), a valid address code is applied to the address lines for a specified time interval called the ***write cycle time***, t_{WE} . Next, the chip select (CS) and the write enable (WE) in puts go LOW. The required time interval from the beginning of a valid address until the WE input goes LOW is called the ***address setup time***, $t_{s(A)}$. The time that the WE input must be LOW is the write pulse width. The time that the input WE must remain LOW after valid data are applied to the data inputs is designated t_{WD}; the time that the valid input data must remain on the data lines after the WE input goes HIGH is the data hold time, $t_{h(D)}$.

During each write cycle, one unit of data is written into the memory.



(b) Write cycle ($\overline{WE}$ LOW)

5.6.2 READ- ONLY MEMORIES (ROMS)

A ROM contains permanently or semi-permanently stored data, which can be read from the memory but either cannot be changed at all or cannot be changed without specialization equipment. A ROM stores data that are used repeatedly in system applications, such as tables, conversions, or programmed instructions for system initialization and operation. ROMs retain stored data when the power is OFF and are therefore nonvolatile memories.

The ROMs are classified as follows:

i. Masked ROM (ROM)

ii. Programmed ROM (PROM)

iii. Erasable PROM (EPROM)

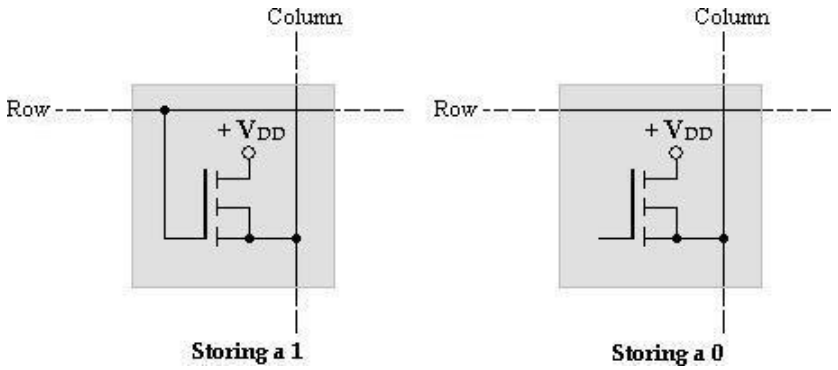
iv. Electrically Erasable PROM (EEPROM)

5.6.2.1 Masked ROM

The mask ROM is usually referred to simply as a ROM. It is permanently programmed during the manufacturing process to provide widely used standard functions, such as popular conversions, or to provide user-specified functions. Once the memory is programmed, it cannot be changed.

Most IC ROMs utilize the presence or absence of a transistor connection at a row/column junction to represent a 1 or a 0. The presence of a connection from a row line to the gate of a transistor represents a 1 at that location because when the row line is taken HIGH; all transistors with a gate connection to that row line turn on

and connect the HIGH (1) to the associated column lines.



ROM Cells

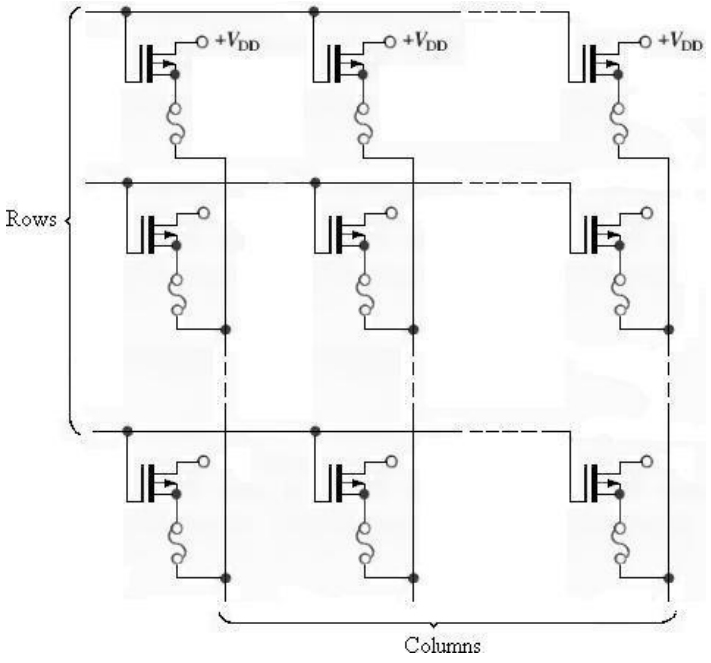
At row/column junctions where there are no gate connections, the column lines remain LOW (0) when the row is addressed.

5.6.2.2 PROM (Programmable Read-Only Memory)

The PROM (Programmable Read-only memory), comes from the manufacturer unprogrammed and are custom programmed in the field to meet the user‘s needs.

A PROM uses some type of fusing process to store bits, in which a memory link is burned open or left intact to represent a 0 or a 1. The fusing process is irreversible; once a PROM is programmed, it cannot be changed.

The fusible links are manufactured into the PROM between the source of each cell's transistor and its column line. In the programming process, a sufficient current is injected through the fusible link to bum it open to create a stored O. The link is left intact for a stored 1. All drains are commonly connected to V_{DD}.



PROM array with fusible links

Three basic fuse technologies used in PROMs are metal links, silicon links, and pn junctions. A brief description of each of these follows.

1. **Metal links** are made of a material such as ***nichrome***. Each bit in the memory array is represented by a separate link. During programming, the link is either "blown" open or left intact. This is done basically by first addressing a given cell and then forcing a sufficient amount of current through the link to cause it to open. When the fuse is intact, the memory cell is configured as a logic 1 and whenfuse is blown (open circuit) the memory cell is logic 0.

2. **Silicon links** are formed by narrow, notched strips of ***polycrystalline silicon***. Programming of these fuses requires melting of the links by passing a sufficient amount of current through them. This amount of current causes a high temperature at the fuse location that oxidizes the silicon and forms insulation
around the now-open link.

3. **Shorted junction**, or avalanche-induced migration, technology consists basically of two pn junctions arranged back-to-back. During programming, one of thediode junctions is avalanched, and the resulting voltage and heat causealuminum ions to migrate and short the junction. The remaining junction is then used as a forward- biased diode to represent a data bit.

5.6.2.3 EPROM (Erasable Programmable ROM)

An EPROM is an erasable PROM. Unlike an ordinary PROM, an EPROM can be reprogrammed if an existing program in the memory array is erased first.

An EPROM uses an NMOSFET array with an isolated-gate structure. The isolated transistor gate has no electrical connections and can store an electrical charge for indefinite periods of time. The data bits in this type of array are represented by the presence or absence of a stored gate charge. Erasure of a data bitis a process that removes the gate charge.

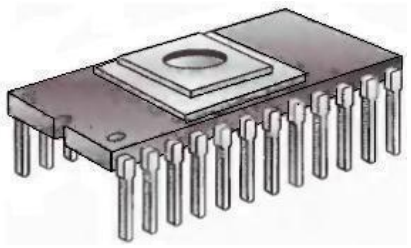
Two basic types of erasable PROMs are the ultraviolet erasable PROM (UV EPROM) and the electrically erasable PROM (EEPROM).

- **UV EPROM:**

You can recognize the UV EPROM device by the transparent quartz lid on the package, as shown in Figure below. The isolated gate in the FET of an ultraviolet EPROM is "floating" within an oxide insulating material. The programming process causes electrons to be removed from the floating gate. Erasure is done by exposure of the memory array chip to high-intensity ultraviolet radiation through the quartz window on top of the package.

The positive charge stored on the gate is neutralized after several minutes to an hour of exposure time. In EPROM‘s, it is not possible to erase selective information, when erased the entire information is lost. The chip can be reprogrammed.

It is ideally suited for product development, college laboratories, etc.



Ultraviolet Erasable PROM

During programming, address and datas are applied to address and data pins of the EPROM. The program pulse is applied to the program input of the EPROM. The program pulse duration is around 50msec and its amplitude depends on EPROM IC. It is typically 11.5V to 25V.

In EPROM, it is possible to program any location at any time- either individually, sequentially or at random.

5.6.2.4 EEPROM (Electrically Erasable PROM)

The EEPROM (Electrically Erasable PROM), also uses MOS circuitry. Data is stored as charge or no charge on an insulating layer, which is made very thin (< 200Å). Therefore a voltage as low as 20- 25V can be used to move charges across thethin barrier in either direction for programming or erasing ROM.

An electrically erasable PROM can be both erased and programmed with electrical pulses. Since it can be both electrically written into and electrically erased, the EEPROM can be rapidly programmed and erased in-circuit for reprogramming.

It allows selective erasing at the register level rather than erasing all the information, since the information can be changed by using electrical signals.

It has chip erase mode by which the entire chip can be erased in 10 msec. Hence EEPROM's are most expensive.

Advantages of RAM:

1. Fast operating speed **(< 150 nsec**),
2. Low power dissipation **(< 1mW**),
3. Economy,
4. Compatibility,
5. Non-destructive read-out.

Advantages of ROM:

1. Ease and speed of design,
2. Faster than MSI devices (PLD and FPGA)
3. The program that generates the ROM contents can easily be structured to handle unusual or undefined cases,
4. A ROM's function is easily modified just by changing the stored pattern, usually without changing any external connections,
5. More economical.

Disadvantages of ROM:

1. For functions more than 20 inputs, a ROM based circuit is impractical because of the limit on ROM sizes that are available.
2. For simple to moderately complex functions, ROM based circuit may be costly: consume more power; run slower.

Comparison between RAM and ROM:

S.No	RAM	ROM
1	RAMs have both read and write capability.	ROMs have only read operation.
2	RAMs are volatile memories.	ROMs are non-volatile memories.
3	They lose stored data when the power is turned OFF.	They retain stored data even if power is turned off.
4	RAMs are available in both bipolar and MOS technologies.	RAMs are available in both bipolar and MOS technologies.
5	**Types**: SRAM, DRAM, EEPROM	**Types**: PROM, EPROM.

Comparison between SRAM and DRAM:

S.No	Static RAM	Dynamic RAM
1	It contains less memory cells per unit area.	It contains more memory cells per unit area.
2	Its access time is less, hence faster memories.	Its access time is greater than static RAM
3	It consists of number of flip-flops. Each flip-flop stores one bit.	It stores the data as a charge on the capacitor. It consists of MOSFET and capacitor for each cell.
4	Refreshing circuitry is not required.	Refreshing circuitry is required to maintain the charge on the capacitors every time after every few milliseconds. Extra hardware is required to control refreshing.
5	Cost is more	Cost is less.

Comparison of Types of Memories:

Memory type	Non- Volatile	High Density	One- Transistor cell	In-system writability
SRAM	No	No	No	Yes
DRAM	No	Yes	Yes	Yes
ROM	Yes	Yes	Yes	No
EPROM	Yes	Yes	Yes	No
EEPROM	Yes	No	No	Yes

5.8 PROGRAMMABLE LOGIC DEVICES:

5.8.1 INTRODUCTION:

A combinational PLD is an integrated circuit with programmable gates divided into an AND array and an OR array to provide an AND-OR sum of product implementation. The PLD's can be reprogrammed in few seconds and hence gives more flexibility to experiment with designs. Reprogramming feature of PLDs also makes it possible to accept changes/modifications in the previously design circuits.

The advantages of using programmable logic devices are:

1. Reduced space requirements.
2. Reduced power requirements.
3. Design security.
4. Compact circuitry.
5. Short design cycle.
6. Low development cost.
7. Higher switching speed.
8. Low production cost for large-quantity production.

According to architecture, complexity and flexibility in programming in PLD‘s are classified as—

- PROMs : Programmable Read Only memories,
- PLAs : Programmable Logic Arrays,
- PAL : Programmable Logic Array,
- FPGA : Field Programmable Gate Arrays,
- CPLDs : Complex Programmable Logic Devices.

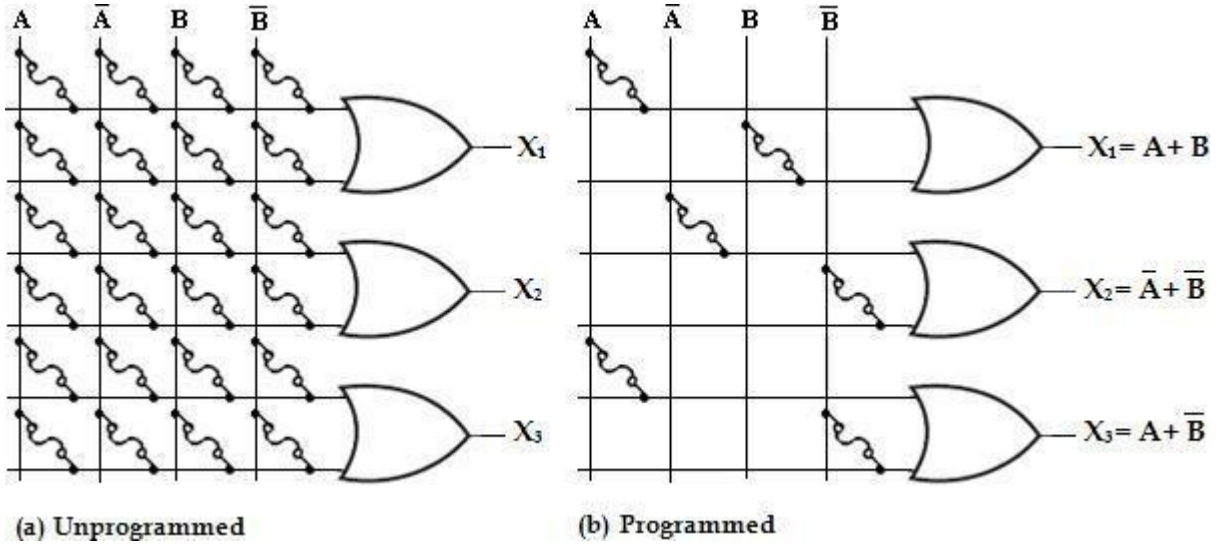
Programmable Arrays:

All PLDs consists of programmable arrays. A programmable array isessentially a grid of conductors that form rows and columns with a fusible link at each cross point. Arrays can be either fixed or programmable.

The OR Array:

It consists of an array of OR gates connected to a programmable matrix with fusible links at each cross point of a row and column, as shown in the figure below. The array can be programmed by blowing fuses to eliminate selected variables from the output functions. For each input to an OR gate, only one fuse is left intact in order to connect the desired variable to the gate input. Once the fuse is blown, it cannot be reconnected.

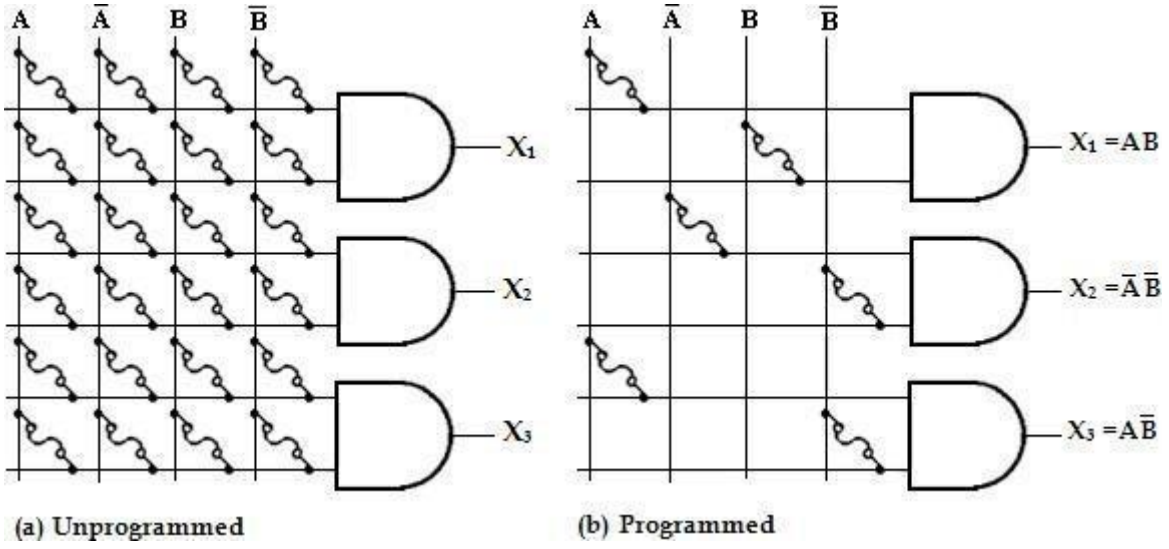
Another method of programming a PLD is the antifuse, which is the opposite of the fuse. Instead of a fusible link being broken or opened to program a variable, a normally open contact is shorted by ―melting‖ the antifuse material to form a connection.



An example of a basic programmable OR array

The AND Array:

This type of array consists of AND gates connected to a programmable matrix with fusible links at each cross points, as shown in the figure below. Like the OR array, the AND array can be programmed by blowing fuses to eliminate selected variables from the output functions. For each input to an AND gate, only one fuse is left intact in order to connect the desired variable to the gate input. Also, like the ORarray, the AND array with fusible links or with antifuses is one-time programmable.



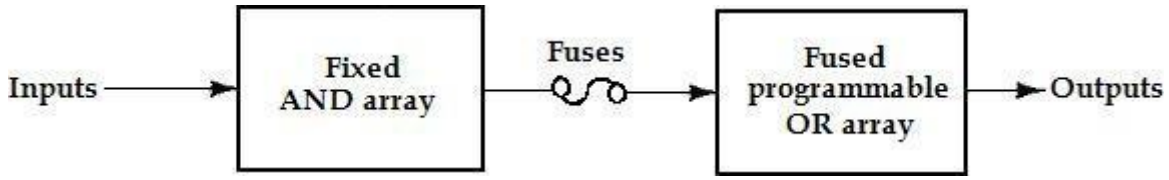
An example of a basic programmable AND array

5.8.2 Classification of PLDs

There are three major types of combinational PLDs and they differ in the placement of the programmable connections in the AND-OR array. The configuration of the three PLDs is shown below.

1. **Programmable Read-Only Memory (PROM):**

A PROM consists of a set of fixed (non-programmable) AND array constructed as a decoder and a programmable OR array. The programmable OR gates implement the Boolean functions in sum of minterms.



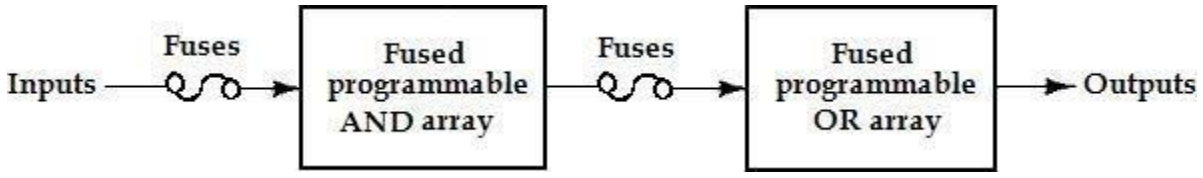
(a) **Programmable read- only memory (PROM)**

2. **Programmable Logic Array (PLA):**

A PLA consists of a programmable AND array and a programmable OR array.

The product terms in the AND array may be shared by any OR gate to provide the required sum of product implementation.

The PLA is developed to overcome some of the limitations of the PROM. The PLA is also called an FPLA (Field Programmable Logic Array) because the user in the field, not the manufacturer, programs it.

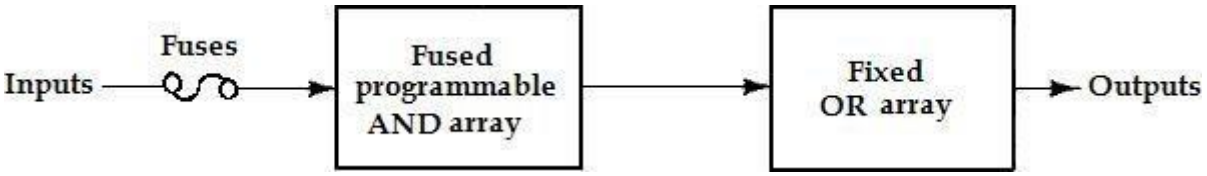


Programmable Logic Array (PLA)

3. Programmable Array Logic (PAL):

The basic PAL consists of a programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate.

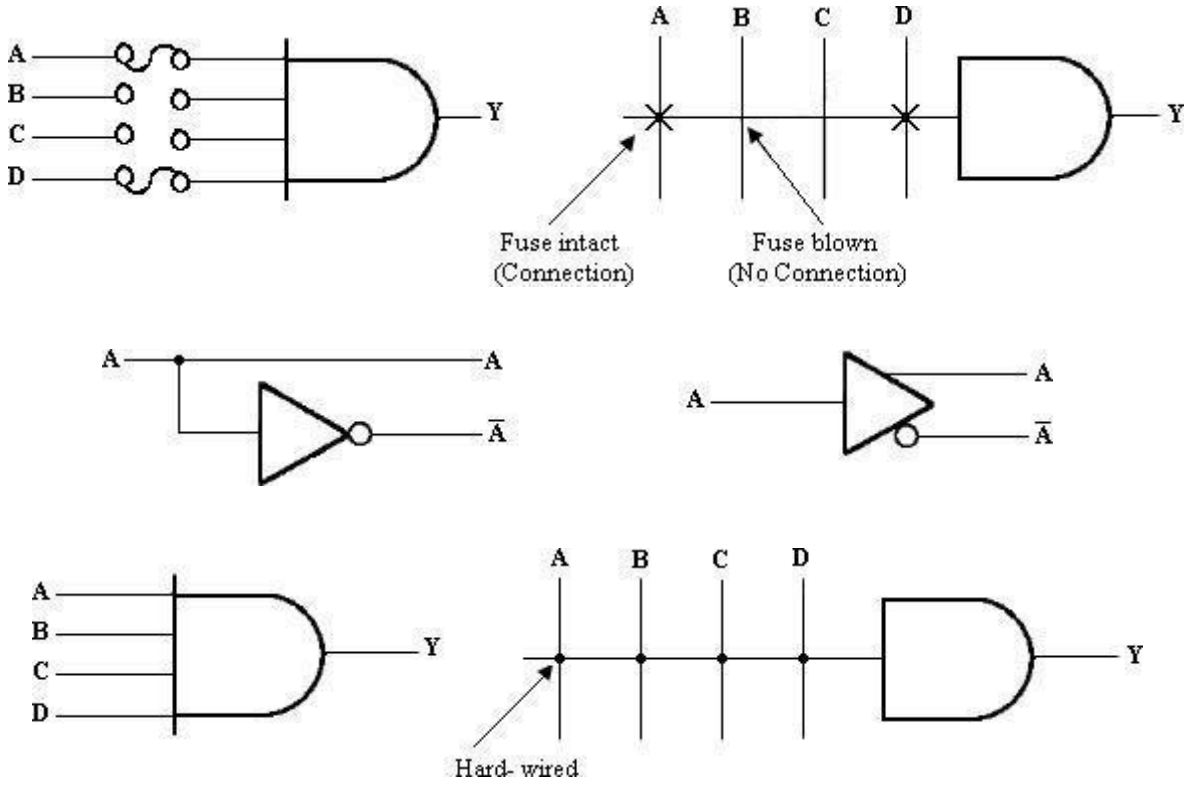
It is developed to overcome certain disadvantages of the PLA, such as longer delays due to the additional fusible links that result from using two programmable arrays and more circuit complexity.



Programmable Array Logic (PAL)

Array logic Symbols:

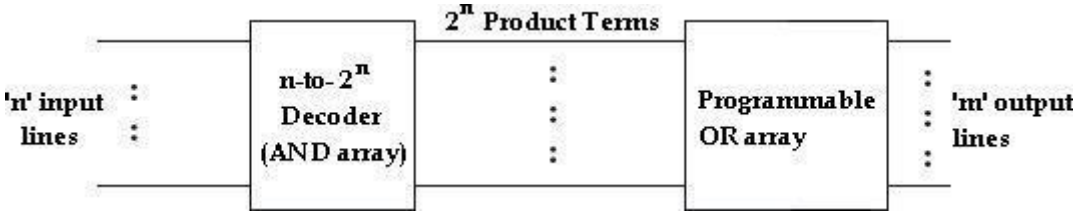
PLDs have hundreds of gates interconnected through hundreds of electronic fuses. It is sometimes convenient to draw the internal logic of such device in a compact form referred to as **array logic**.



5.8.3 PROGRAMMABLE ROM:

PROMs are used for code conversions, generating bit patterns for characters and as look-up tables for arithmetic functions.

As a PLD, PROM consists of a fixed AND-array and a programmable OR array. The AND array is an n-to-2^n decoder and the OR array is simply a collection of programmable OR gates. The OR array is also called the memory array. The decoder serves as a minterm generator. The n-variable minterms appear on the 2^n lines at the decoder output. The 2^n outputs are connected to each of the _m' gates in the OR array via programmable fusible links.



2^n x m PROM

5.8.4 Implementation of Combinational Logic Circuit using PROM

1. Using PROM realize the following expression

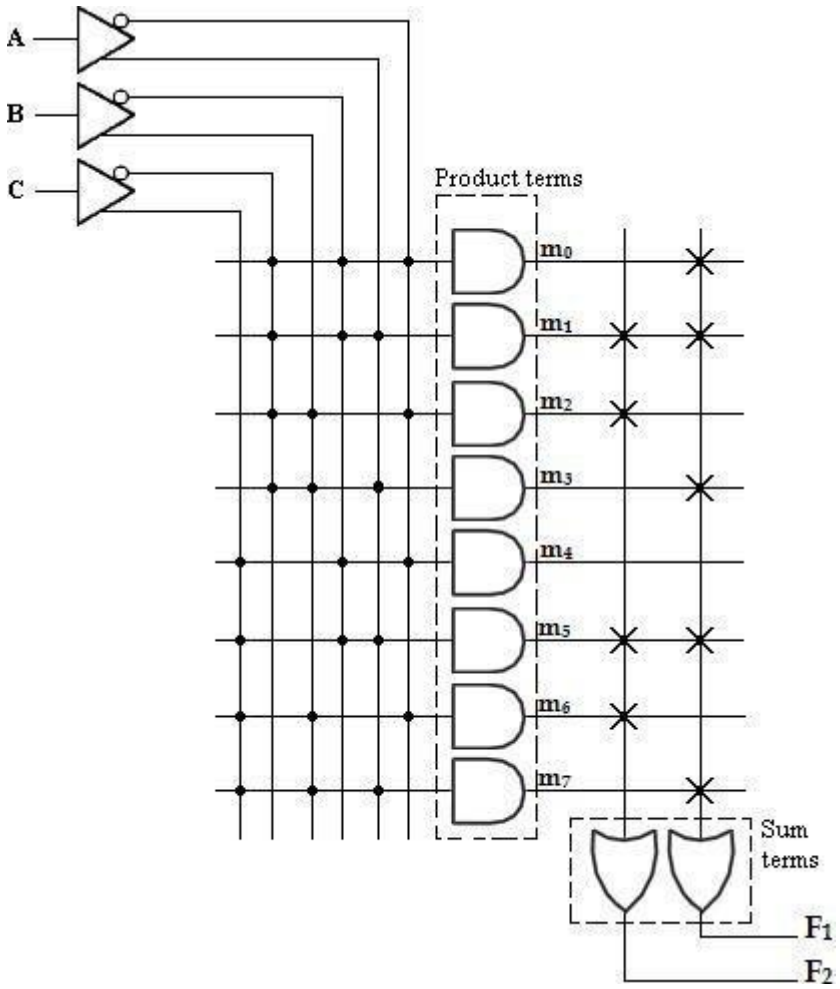
F_1 (A, B, C) = ∑m (0, 1, 3, 5, 7)

F_2 (A, B, C) = ∑m (1, 2, 5, 6)

Step1: Truth table for the given function

A	B	C	F_1	F_2
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	1	0

Step 2: PROM diagram

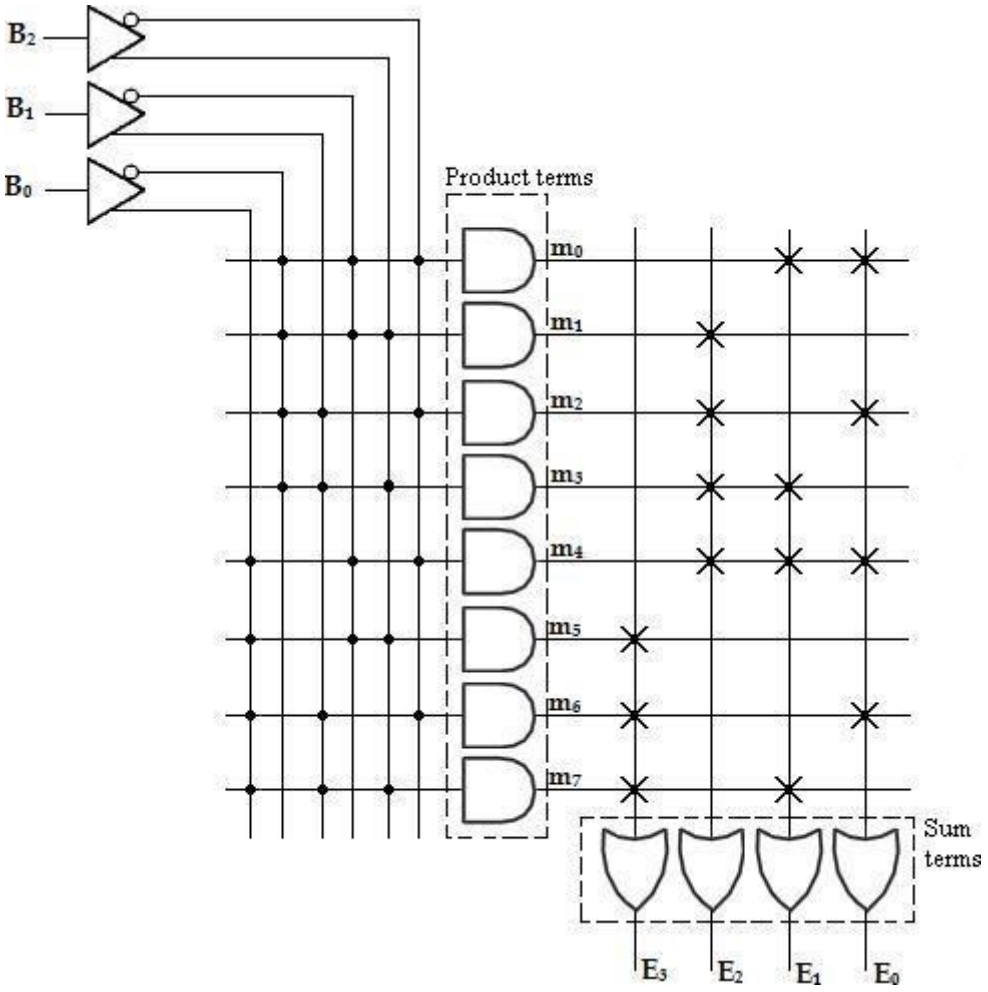


2. Design a combinational circuit using PROM. The circuit accepts 3-bit binary and generates its equivalent Excess-3 code.

Step1: Truth table for the given function

B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	1	1
0	0	1	0	1	0	0
0	1	0	0	1	0	1
0	1	1	0	1	1	0
1	0	0	0	1	1	1
1	0	1	1	0	0	0
1	1	0	1	0	0	1
1	1	1	1	0	1	0

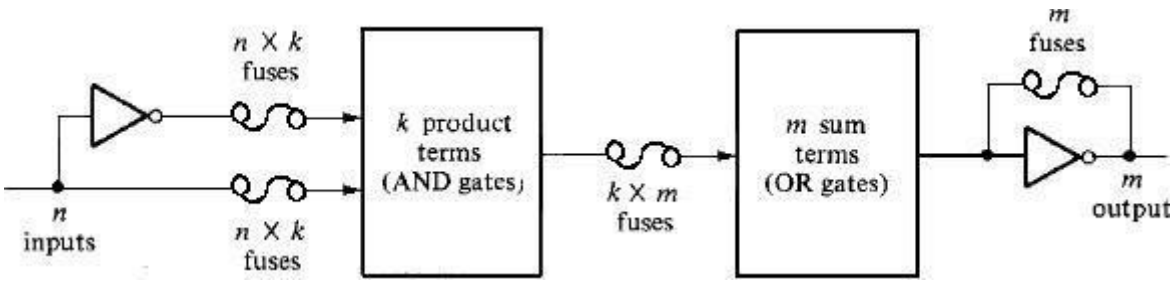
Step 2: PROM diagram



5.8.5 PROGRAMMABLE LOGIC ARRAY: (PLA)

The PLA is similar to the PROM in concept except that the PLA does not provide full coding of the variables and does not generate all the minterms.

The decoder is replaced by an array of AND gates that can be programmed to generate any product term of the input variables. The product term are then connected to OR gates to provide the sum of products for the required Boolean functions. The AND gates and OR gates inside the PLA are initially fabricated with fuses among them. The specific boolean functions are implemented in sum of products form by blowing the appropriate fuses and leaving the desired connections.



PLA block diagram

The block diagram of the PLA is shown above. It consists of _n' inputs, _m' outputs, _k' product terms and _m' sum terms. The product terms constitute a group of _k' AND gates and the sum terms constitute a group of _m' OR gates. Fuses are inserted between all _n' inputs and their complement values to each of the AND gates. Fuses are also provided between the outputs of the AND gate and the inputs of the OR gates.

Another set of fuses in the output inverters allow the output function to be generated either in the AND-OR form or in the AND-OR-INVERT form. With the inverter fuse in place, the inverter is bypassed, giving an AND-OR implementation. With the fuse blown, the inverter becomes part of the circuit and the function is implemented in the AND-OR- INVERT form.

5.8.6 Implementation of Combinational Logic Circuit using PLA

1. **Implement the combinational circuit with a PLA having 3 inputs, 4 product terms and 2 outputs for the functions.**

   F_1 (A, B, C) = ∑m (0, 1, 2, 4)

   F_2 (A, B, C) = ∑m (0, 5, 6, 7)

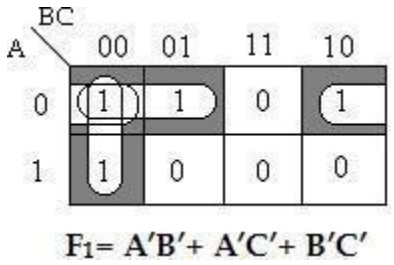
Solution:

Step 1: Truth table for the given functions

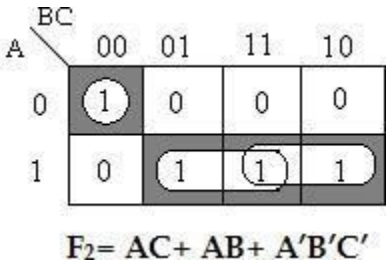
A	B	C	F_1	F_2
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0

1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Step 2: K-map Simplification

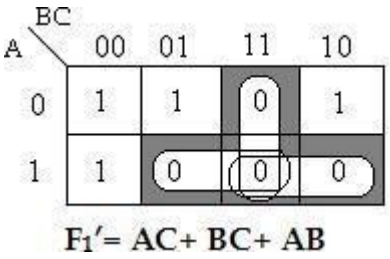


$F_1 = A'B' + A'C' + B'C'$

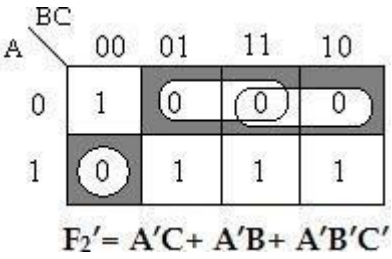


$F_2 = AC + AB + A'B'C'$

With this simplification, total number of product term is 6. But we require only 4 product terms. Therefore find out F_1' and F_2'.



$F_1' = AC + BC + AB$



$F_2' = A'C + A'B + A'B'C'$

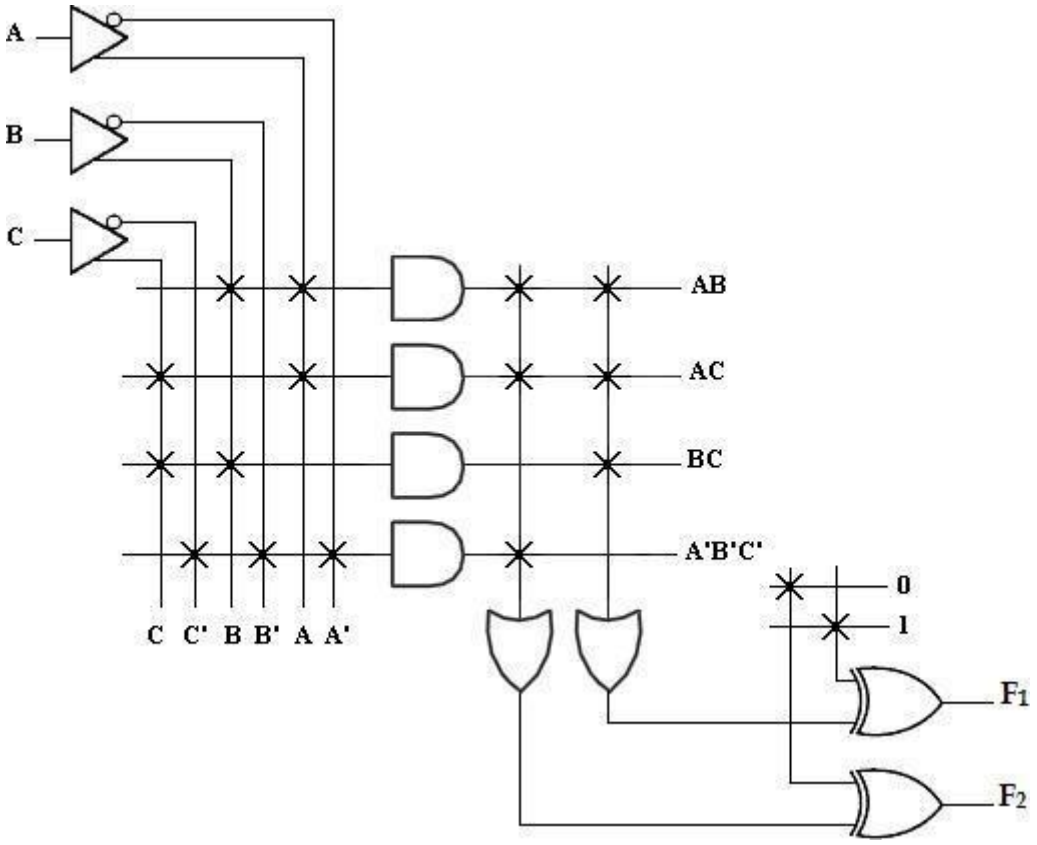
Now select, F_1' and F_2, the product terms are AC, AB, BC and A'B'C'

Step 3: PLA Program table:

	Product term	**Inputs**			**Outputs**	
		A	**B**	**C**	**F_1 (C)**	**F_2 (T)**
AB	1	1	1	-	1	1
AC	2	1	-	1	1	1
BC	3	-	1	1	1	-
A'B'C'	4	0	0	0	-	1

In the PLA program table, first column lists the product terms numerically as 1, 2, 3, and 5. The second column (Inputs) specifies the required paths between the AND gates and the inputs. For each product term, the inputs are marked with 1, 0, or - (dash). If a variable in the product form appears in its normal form, the corresponding input variable is marked with a 1. If it appears complemented, the corresponding input variable is marked with a 0. If the variable is absent in the product term, it is marked with a dash (-). The third column (output) specifies the path between the AND gates and the OR gates. The output variables are marked with 1‘s for all those product terms that formulate the required function.

Step 4: PLA Diagram



The PLA diagram uses the array logic symbols for complex symbols. Each input and its complement is connected to the inputs of each AND gate as indicatedby the intersections between the vertical and horizontal lines. The output of the AND gate are connected to the inputs of each OR gate. The output of the OR gate goes toan EX-OR gate where the other input can be programmed to receive a signal equal to either logic 1 or 0.

The output is inverted when the EX-OR input is connected to 1 ie., **(x ⊕1= x')**. The output does not change when the EX-OR input is connected to 0 ie., **(x ⊕0= x)**.

2. **Implement the combinational circuit with a PLA having 3 inputs, 4 product terms and 2 outputs for the functions.**

   F_1 (A, B, C) = ∑m (3, 5, 6, 7)

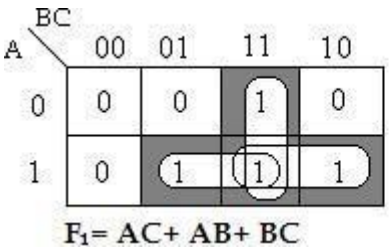
   F_2 (A, B, C) = ∑m (0, 2, 4, 7)

   Solution:

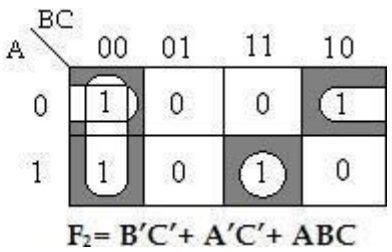
Step 1: Truth table for the given functions

A	B	C	F_1	F_2
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Step 2: K-map Simplification

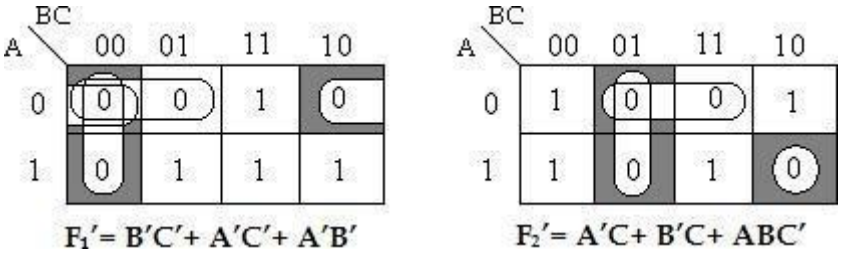


F_1= AC+ AB+ BC



F_2= B'C'+ A'C'+ ABC

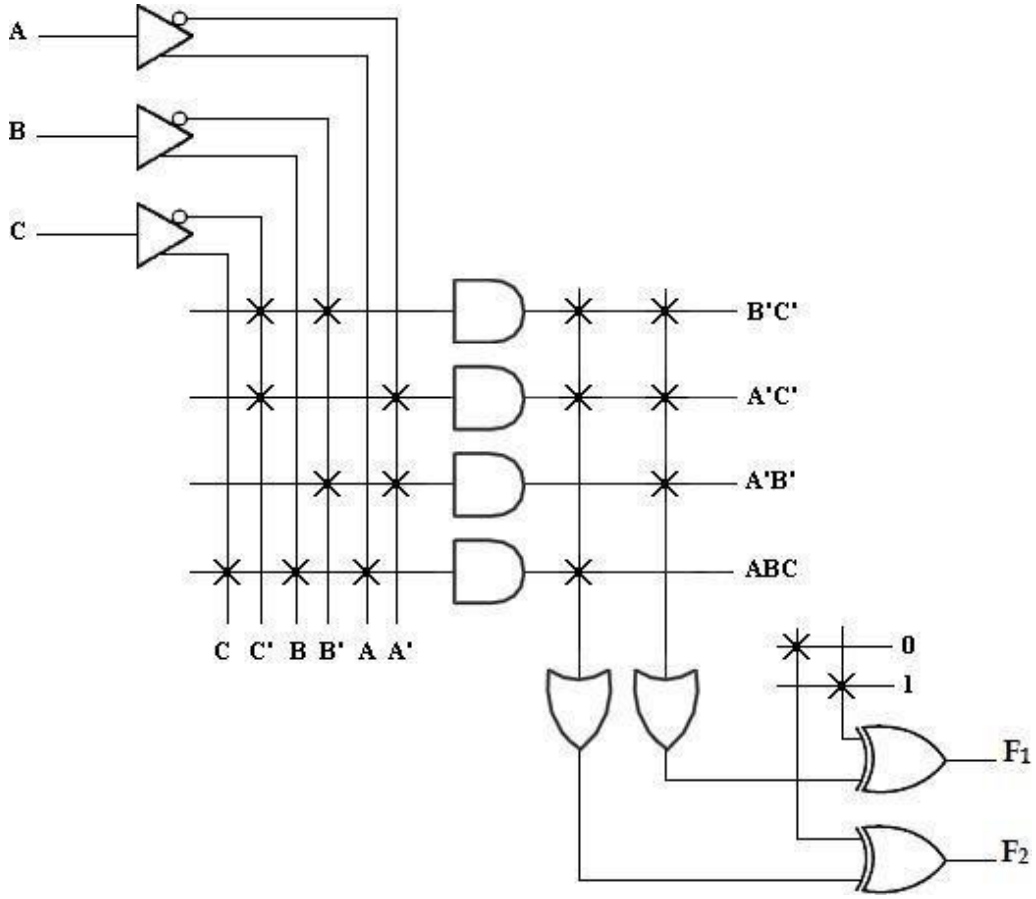
With this simplification, total number of product term is 6. But we require only 4 product terms. Therefore find out F_1' and F_2'.



Now select, F_1' and F_2, the product terms are **B'C', A'C', A'B' and ABC**. **Step 3:** PLA Program table

	Product term	Inputs			Outputs	
		A	B	C	F_1 (C)	F_2 (T)
B'C'	1	-	0	0	1	1
A'C'	2	0	-	0	1	1
A'B'	3	0	0	-	1	-
ABC	4	1	1	1	-	1

Step 4: PLA Diagram



3. Implement the following functions using PLA.

   F_1 (A, B, C) = ∑m (1, 2, 4, 6)

   F_2 (A, B, C) = ∑m (0, 1, 6, 7)

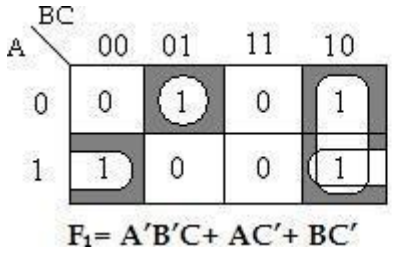
   F_3 (A, B, C) = ∑m (2, 6)

Solution:

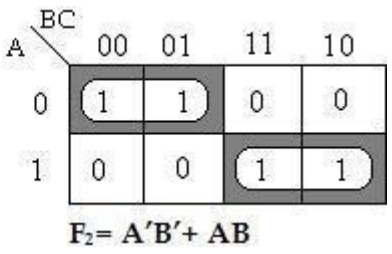
Step 1: Truth table for the given functions

A	B	C	F_1	F_2	F_3
0	0	0	0	1	0
0	0	1	1	1	0
0	1	0	1	0	1
0	1	1	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	1	0	1	1	1
1	1	1	0	1	0

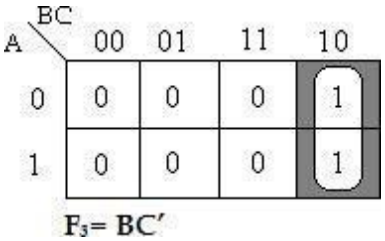
Step 2: K-map Simplification



F_1= A′B′C+ AC′+ BC′



F_2= A′B′+ AB

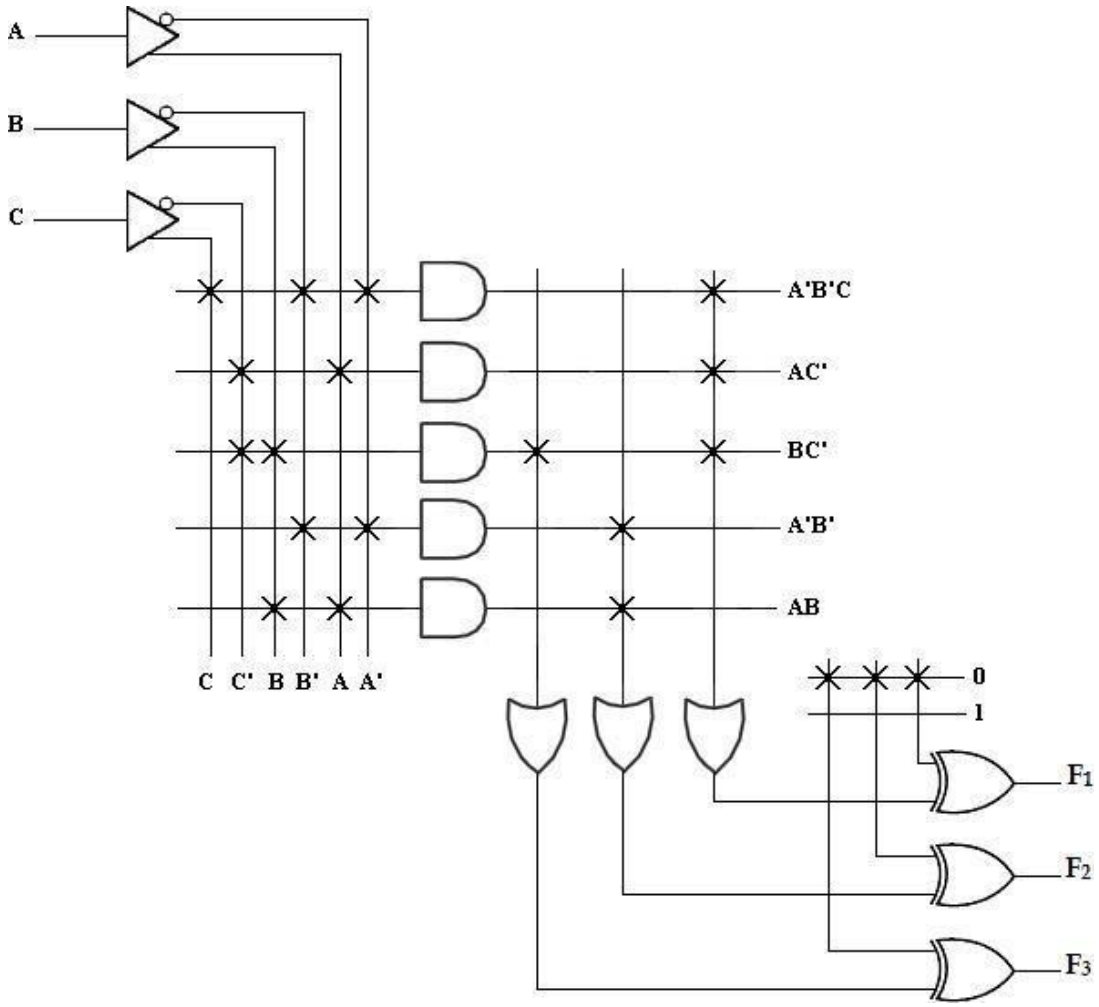


F_3= BC′

Step 3: PLA Program table

	Product term	Inputs			Outputs		
		A	B	C	F_1 (T)	F_2 (T)	F_3 (T)
A‘B‘C	1	0	0	1	1	-	-
AC‘	2	1	-	0	1	-	-
BC‘	3	-	1	0	1	-	1
A‘B‘	4	0	0	-	-	1	-
AB	5	1	1	-	-	1	-

Step 4: PLA Diagram



4. **A combinational circuit is designed by the function**

   F_1 (A, B, C) = ∑m (3, 5, 7)

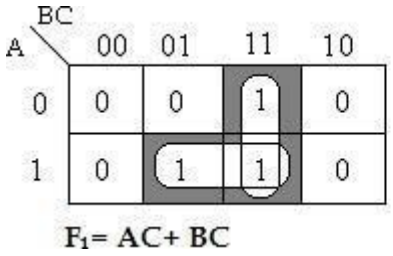
   F_2 (A, B, C) = ∑m (4, 5, 7)

Solution:

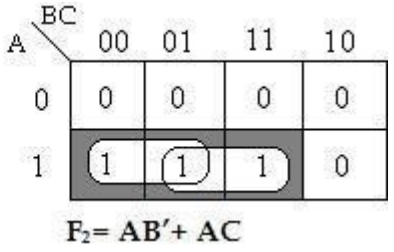
Step 1: Truth table for the given functions

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	1
1	0	1	1	1
1	1	0	0	0
1	1	1	1	1

Step 2: K-map Simplification



$F_1 = AC + BC$

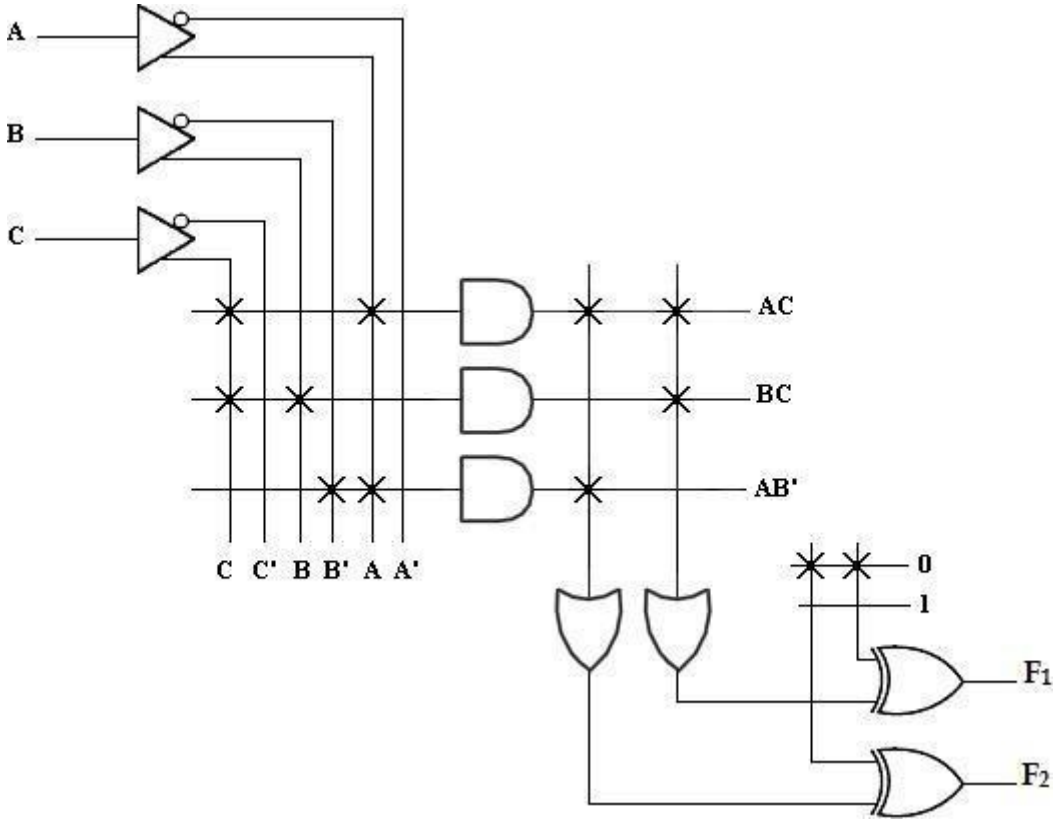


$F_2 = AB' + AC$

Step 3: PLA Program table

	Product term	Inputs			Outputs	
		A	B	C	F_1 (C)	F_2 (T)
AC	1	1	-	1	1	1
BC	2	-	1	1	1	-
AB‘	3	1	0	-	-	1

Step 4: PLA Diagram



5. **A combinational circuit is defined by the functions, F_1 (A, B, C) = ∑m (1, 3, 5)**

   F_2 (A, B, C) = ∑m (5, 6, 7)

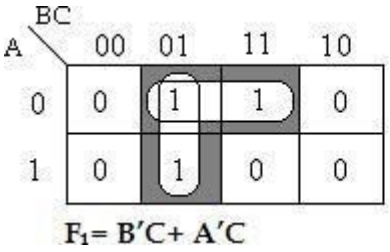
   Implement the circuit with a PLA having 3 inputs, 3 product terms and 2 outputs.

Solution:

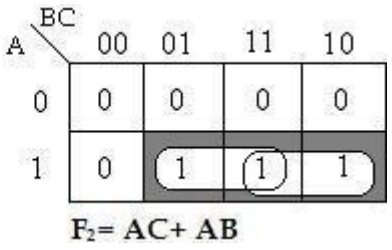
Step 1: Truth table for the given functions

A	B	C	F_1	F_2
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	0
1	0	0	0	0
1	0	1	1	1
1	1	0	0	1
1	1	1	0	1

Step 2: K-map Simplification

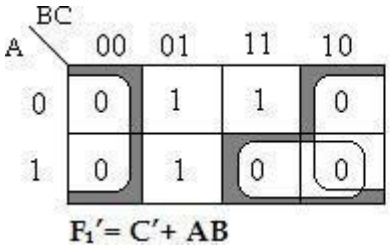


$F_1 = B'C + A'C$

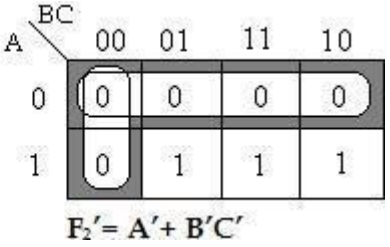


$F_2 = AC + AB$

With this simplification, total number of product term is 5. But we require only 3 product terms. Therefore find out F_1' and F_2'.



$F_1' = C' + AB$



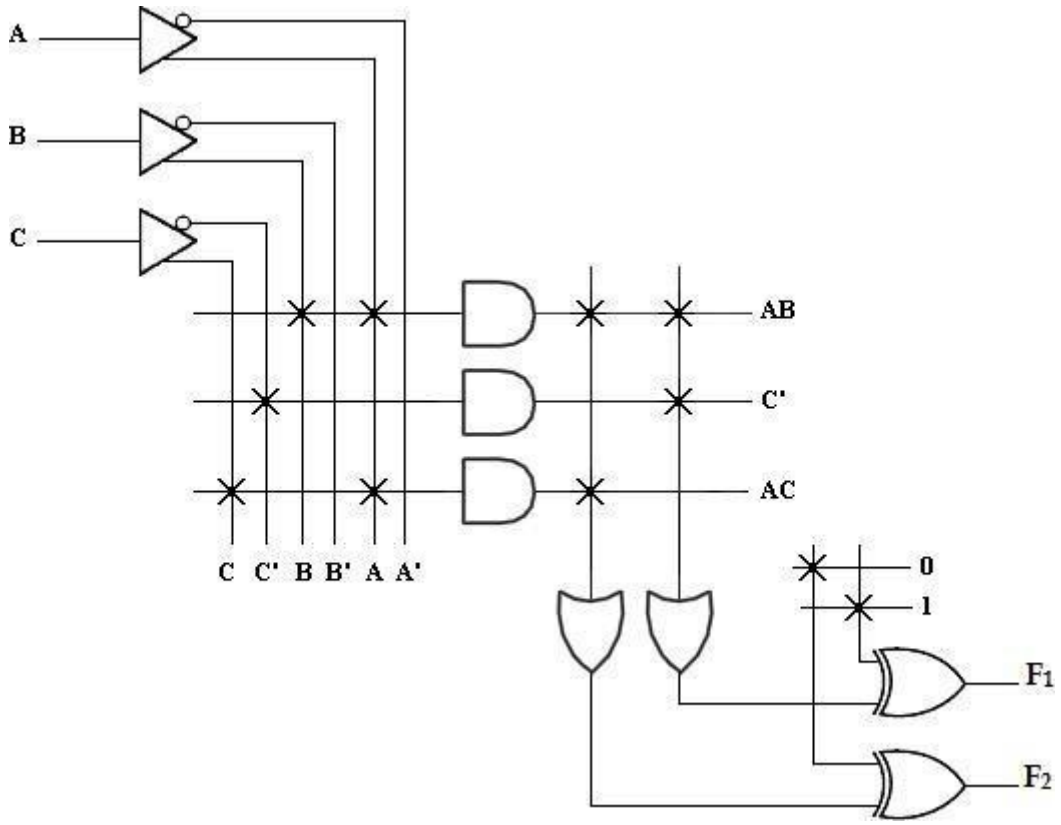
$F_2' = A' + B'C'$

Now select, F_1' and F_2, the product terms are **AC, AB and C'**.

Step 3: PLA Program table

	Product term	Inputs			Outputs	
		A	**B**	**C**	**F_1 (C)**	**F_2 (T)**
AB	1	1	1	-	1	1
C'	2	-	-	0	1	-
AC	3	1	-	1	-	1

Step 4: PLA Diagram



6. **A combinational circuit is defined by the**

functions, F_1 (A, B, C) = ∑m (0, 1, 3, 4)

F_2 (A, B, C) = ∑m (1, 2, 3, 4, 5)

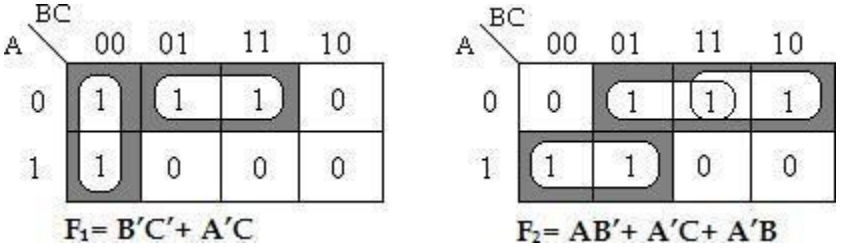
Implement the circuit with a PLA having 3 inputs, 4 product terms and 2 outputs.

Solution:

Step 1: Truth table for the given functions

A	B	C	F_1	F_2
0	0	0	1	0
0	0	1	1	1
0	1	0	0	1
0	1	1	1	1
1	0	0	1	1
1	0	1	0	1
1	1	0	0	0
1	1	1	0	0

Step 2: K-map Simplification

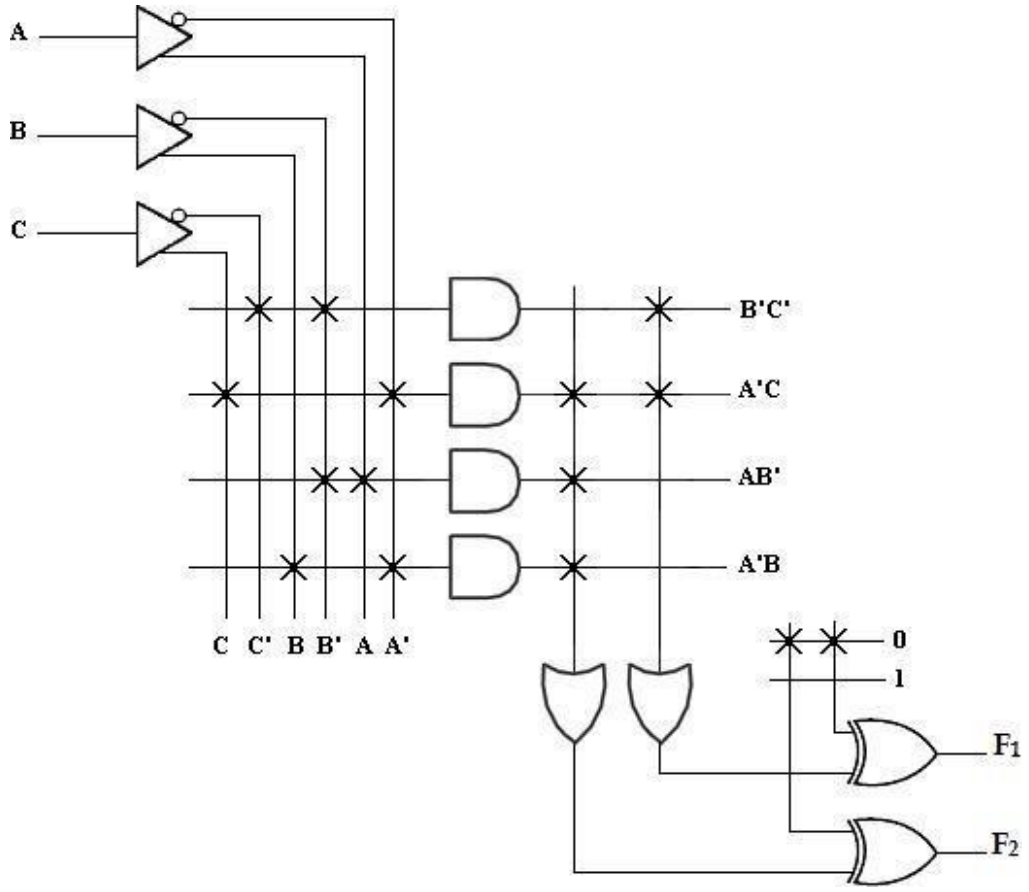


The product terms are **B'C', A'C, AB' and A'B**.

Step 3: PLA Program table

	Product term	Inputs			Outputs	
		A	B	C	F_1 (T)	F_2 (T)
B'C'	1	-	0	0	1	-
A'C	2	0	-	1	1	1
AB'	3	1	0	-	-	1
A'B	4	0	1	-	-	1

Step 4: PLA Diagram



7. **A combinational logic circuit is defined by the function,**

F (A, B, C, D) = ∑m (3, 4, 5, 7, 10, 14, 15)

G (A, B, C, D) = ∑m (1, 5, 7, 11, 15)

Implement the circuit with a PLA having 4 inputs, 6 product terms and 2 outputs.

Solution:

Step 1: Truth table for the given functions

A	B	C	D	F	G
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	0	0
0	0	1	1	1	0
0	1	0	0	1	0
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	1	1
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	0
1	0	1	1	0	1
1	1	0	0	0	0
1	1	0	1	0	0
1	1	1	0	1	0
1	1	1	1	1	1

Step 2: K-map Simplification

For F

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

F= A'BC'+ A'CD+ BCD+ ACD'

For G

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

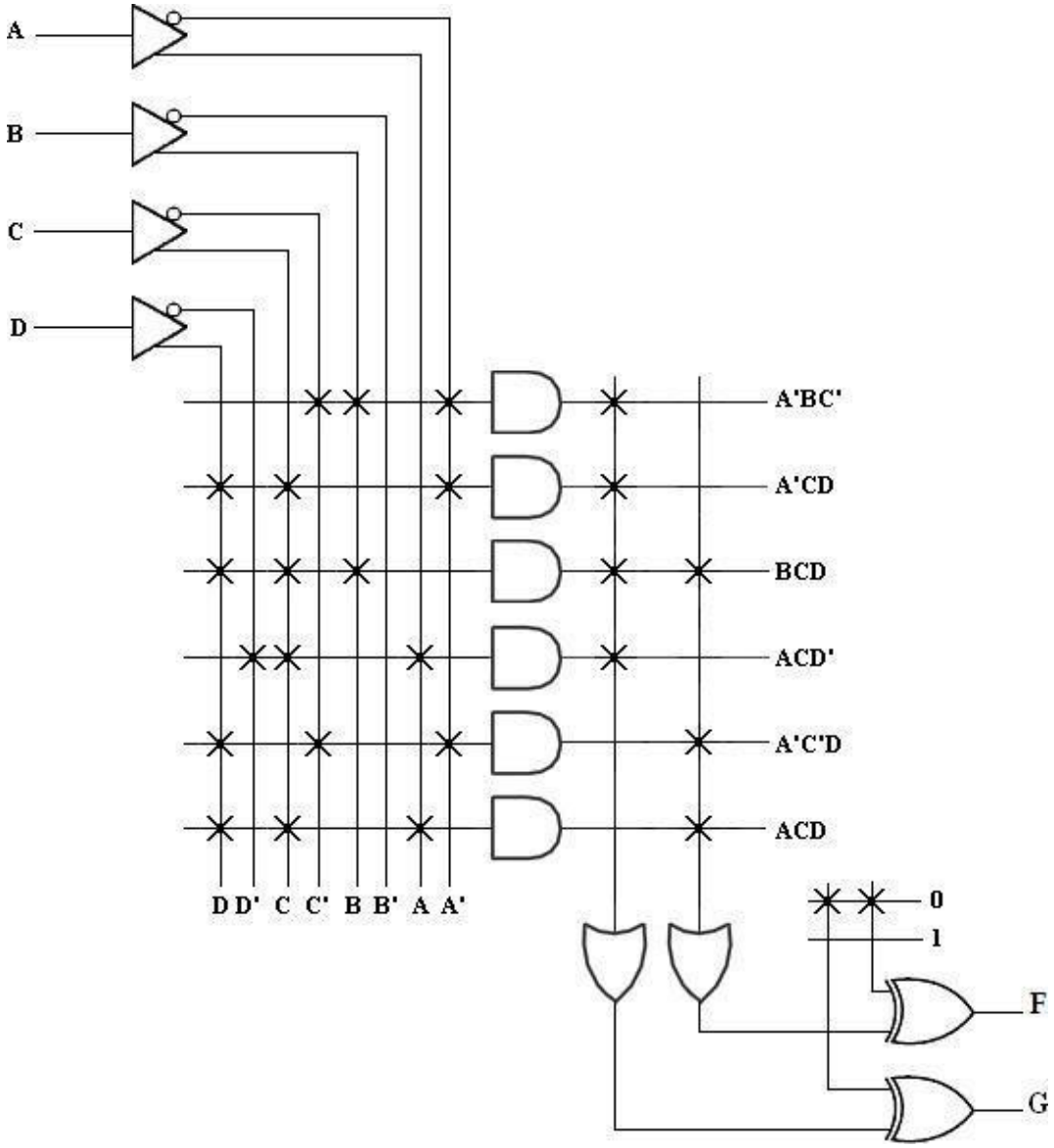
G= A'C'D+ BCD+ ACD

The product terms are A‘BC‘, A‘CD, BCD, ACD‘, A‘C‘D, ACD

Step 3: PLA Program table

	Product term	Inputs				Outputs	
		A	B	C	D	F (T)	G (T)
A‘BC‘	1	0	1	0	-	1	-
A‘CD	2	0	-	1	1	1	-
BCD	3	-	1	1	1	1	1
ACD‘	4	1	-	1	0	1	-
A‘C‘D	5	0	-	0	1	-	1
ACD	6	1	-	1	1	-	1

Step 4: PLA Diagram



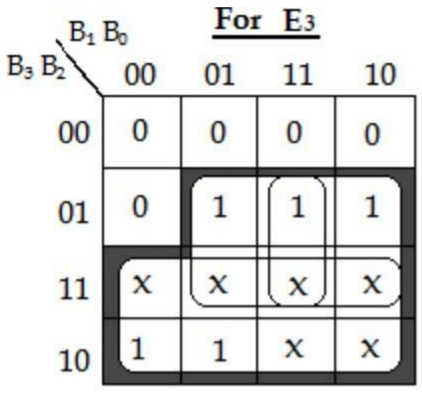
8. **Design a BCD to Excess-3 code converter and implement using suitable PLA.**

Solution:

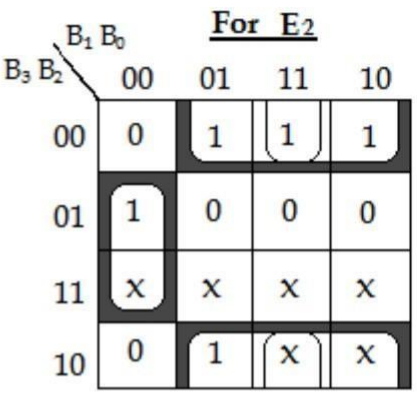
Step 1: Truth table of BCD to Excess-3 converter is shown below,

Decimal	BCD code				Excess-3 code			
	B_3	B_2	B_1	B_0	E_3	E_2	E_1	E_0
0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	0	0
6	0	1	1	0	1	0	0	1
7	0	1	1	1	1	0	1	0
8	1	0	0	0	1	0	1	1
9	1	0	0	1	1	1	0	0

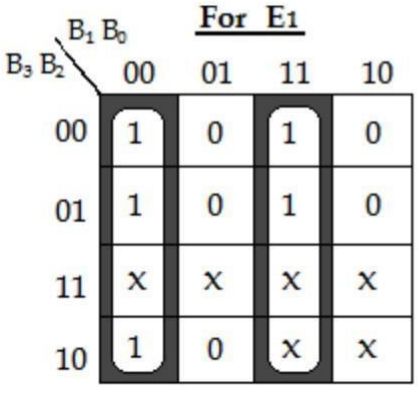
Step 2: K-map Simplification



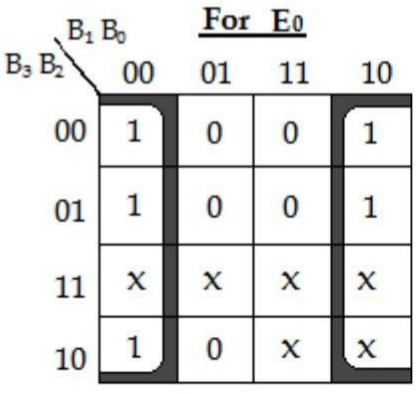
$E_3 = B_3 + B_2B_0 + B_2B_1$



$E_2 = B_2B_1'B_0' + B_2'B_0 + B_2'B_1$



$E_1 = B_1'B_0' + B_1B_0$



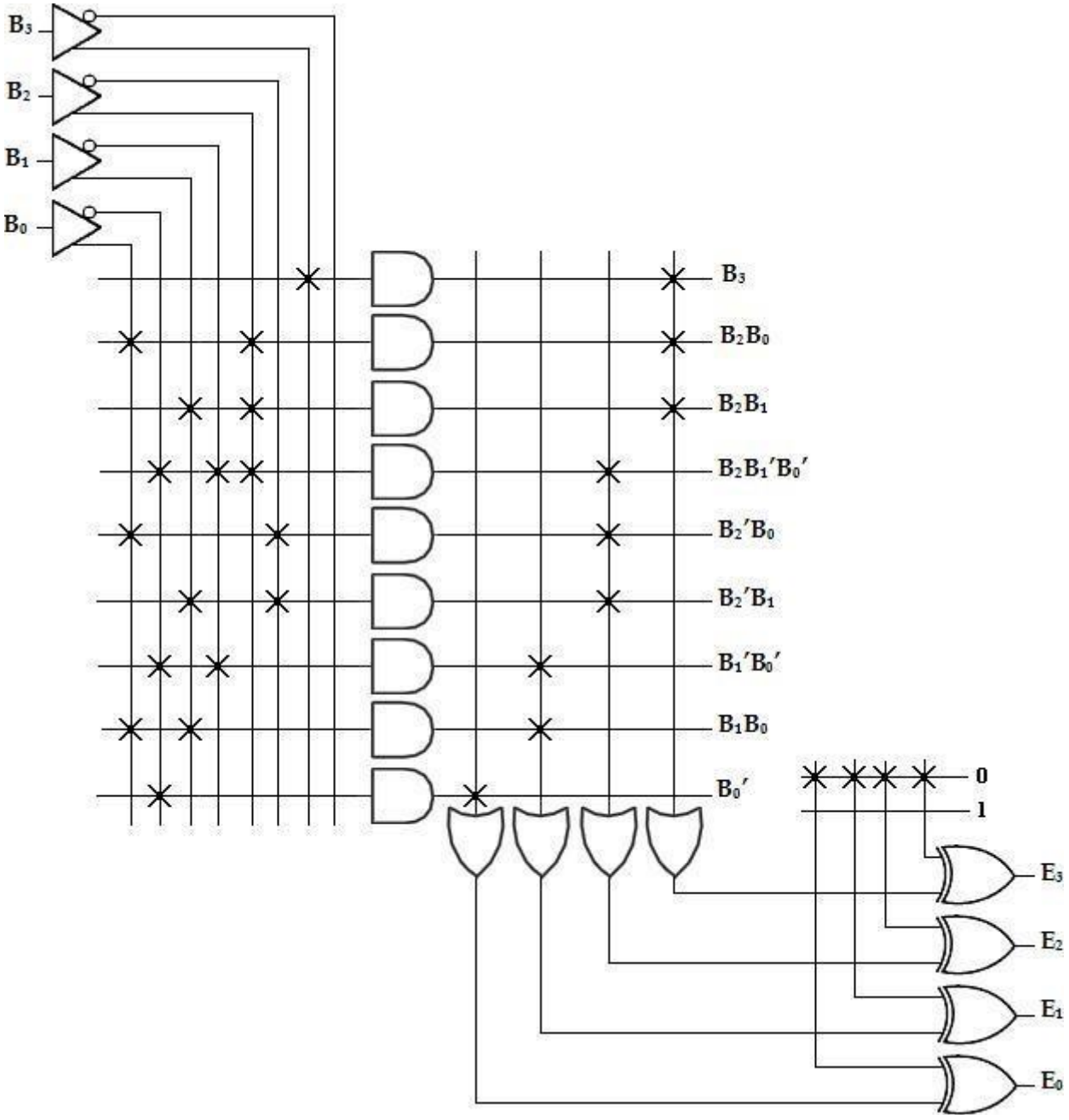
$E_0 = B_0'$

The product terms are B_3, B_2B_0, B_2B_1, $B_2B_1'B_0'$, $B_2'B_0$, $B_2'B_1$, $B_1'B_0'$, B_1B_0, B_0'

Step 3: PLA Program table

	Product terms	Inputs				Outputs			
		B_3	B_2	B_1	B_0	E_3 (T)	E_2 (T)	E_1 (T)	E_0 (T)
B_3	1	1	-	-	-	1	-	-	-
B_2B_0	2	-	1	-	1	1	-	-	-
B_2B_1	3	-	1	1	-	1	-	-	-
$B_2B_1'B_0'$	4	-	1	0	0	-	1	-	-
$B_2'B_0$	5	-	0	-	1	-	1	-	-
$B_2'B_1$	6	-	0	1	-	-	1	-	-
$B_1'B_0'$	7	-	-	0	0	-	-	1	-
B_1B_0	8	-	-	1	1	-	-	1	-
B_0'	9	-	-	-	0	-	-	-	1

Step 4: PLA Diagram



Comparison between PROM, PLA, and PAL:

S.No	PROM	PLA	PAL
1	AND array is fixed and OR array is programmable	Both AND and OR arrays are programmable	OR array is fixed and AND array is programmable
2	Cheaper and simpler to use	Costliest and complex	Cheaper and simpler

3	All minterms are decoded	AND array can be programmed to get desired minterms	AND array can be programmed to get desired minterms
4	Only Boolean functions in standard SOP form can be implemented using PROM	Any Boolean functions in SOP form can be implemented using PLA	Any Boolean functions in SOP form can be implemented using PLA

5. DIGITAL LOGIC FAMILIES

❁ Integrated circuits

Integrated Circuits are used for producing several different circuit configurations and production technologies. The semiconductor chip consists of electronic components used for constructing circuits. Integrated circuits are classified into

- ❁ Linear Integrated Circuits
- ❁ Digital Integrated Circuits

Both operate with continuous and discrete signals respectively and are used to construct various electronic circuits.

There are different levels of Integration, based on the number of logic gates in a single IC package.

❁ Small scale Integration (SSI)

These IC"s contain fewer logic gates (0 to 10).The input and output pins can be directly connected to the pins in the package.

❁ Medium Scale Integration (MSI)

These IC"s have approximately around 10 to 1000 gates in one package, which performs some specific functions.

(Ex: adders, multiplexers)

❁ Large Scale Integration (LSI)

These IC"s contain thousands of gates in a single package.

(Ex: processors, memory chips and programmable logic devices)

❁ Very Large Scale Integrated Devices (VLSI)

These IC"s contain hundreds of thousands of gates in a single package.

(Ex: memory arrays, computer chips)

❁ Classification of Logic Families

Based on the circuit technology digital IC"s are classified into various types

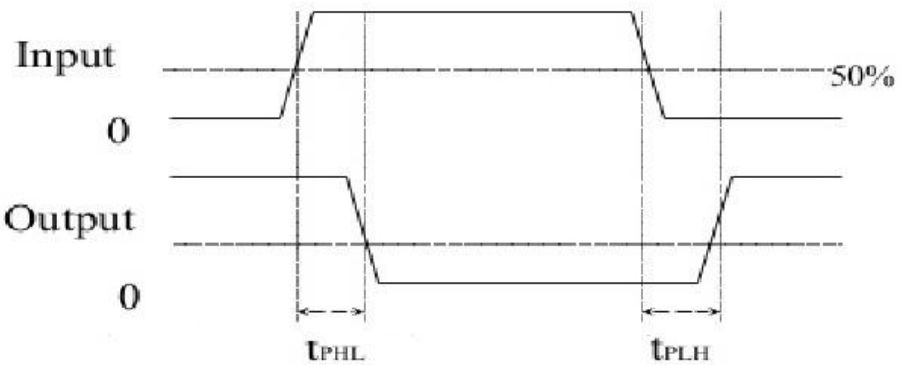
- ❁ RTL (Resistor Transistor Logic)
- ❁ DTL (Diode Transistor Logic)
- ❁ TTL (Transistor Transistor Logic)
- ❁ ECL (Emitter Coupled Logic)
- ❁ MOS (Metal Oxide Semiconductor Logic)

❁ Characteristics of Logic Families

The following are the important characteristics of digital IC"s

- ❁ Propagation Delay (or) Operating Speed
- ❁ Voltage and Current parameters
- ❁ Power Dissipation
- ❁ Fan-in
- ❁ Fan-out
- ❁ Noise Margin
- ❁ Operating Temperature
- ❁ Power Supply Requirements

❁ Propa



Propaga spond
when it:
Propaga

t_{PLH} : Propagation delay time from low level(0) to high level(1)

t_{PHL} : Propagation delay time from high level(1) to low level(0)

The delay in output is a measure of relative speed of logic circuits.

Average propagation delay $P_{D\ avg} = (t_{PLH} + t_{PHL})/2$

❁ Voltage and Current Parameters:

Digital logic gates have a certain range of voltage and current levels

corresponding to 0(low level)) and 1(high level)

❁ Voltage and current levels:

High level input voltage and current: The minimum input voltage recognised as logic 1(high level) by the logic gates (2V – 3V range) and the corresponding current is high level input current.

Low level input voltage and current: The maximum input voltage recognised as logic 0(low level) by the logic gates (around 0.8V) and the corresponding current is low level input current.

High level output voltage and current: The minimum voltage available at the output corresponding to logic 1 and the corresponding current is high level output current.

Low level output voltage and current: The minimum voltage available at the output corresponding to logic 0 and the corresponding current is low level output current.

❁ Power Dissipation:

Power dissipation is the measure of power consumed by the logic gate when fully driven by all its inputs. It is expressed in milliwatts or nanowatts.

Average power dissipation $P_{dc\ avg} = V_{cc} \times I_{avg}$

V_{cc} - DC supply voltage

I_{avg} – Average current taken from the supply

❁ Fan-in:

Fan-in is the number of inputs available in a gate

❁ Fan-out:

Fan-out is the number of similar logic gates that the output of a gate can drive without affecting the normal operation.

❁ Noise Margin:

Noise margin is the maximum external noise voltage added to the input signal that does not cause any undesirable change in the circuit operation.

❁ Operating Temperature:

All integrated circuits are semiconductor devices sensitive to temperature

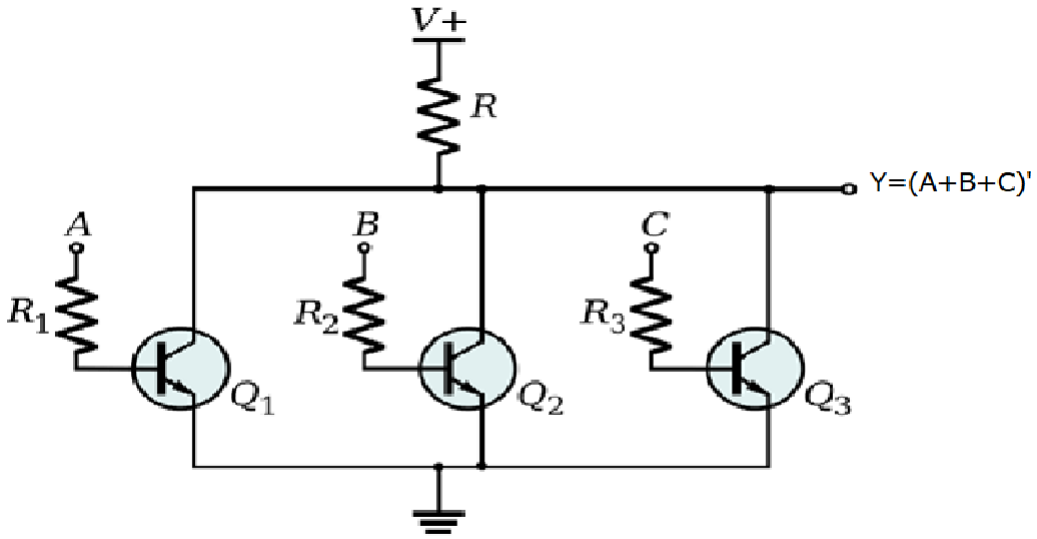
- ❁ Operating range
- ❁ 0ºC to +70º for consumer applications
- ❁ -55º to +125ºC for military applications

❁ Power Supply Requirements:

The amount of power required by the IC.

❁ 5.1 RESISTOR TRANSISTOR LOGIC (RTL)

❁ The following diagram shows the Resistor Transistor Logic (RTL) of NOR logic function.



❁ Basic diagram of RTL NOR consists of transistor and resistors.

Here it is three input NOR gates logic diagram using RTL [i.e., A, B, C].

This follows the NOR gates truth table in its operation. i.e., whenever any one of the input is "HIGH" then it produce low output (or) all inputs are low, it produces "Low" output. This is similar to NOR logic truth table shown below.

INPUT		OUTPUT
A	B	Y=(A+B)'
0	0	1
0	1	0
1	0	0
1	1	0

❁ Operation:

When all inputs are zero (or) low, then output Q= 0.

Since, all these transistors Q1, Q2, Q3 are in OFF condition; its collector output is high.

When any one of the input is high (ort all inputs are high, then its corresponding transistor is going to ON condition. Also, it is connected with ground and collector potential which is approximately zero.

Anyhow, the base current is practically independent of the emitter junction characteristic. When the resistors increase the input resistance and reduce the switching speed of the circuit. This reduces the rise and fall times of any input pulse.

In practice, this approach to increase the speed of an RTL is to connect a capacitor called a speed-up capacitor which is parallel to resistance connected in base.

❁ Operation Table

Inputs			Transistor Status			Output (Y=A+B+C)'
A	B	C	Q1	Q2	Q3	
0	0	0	OFF	OFF	OFF	1(HIGH)
1	0 or 1	0 or 1	ON	ON or OFF	ON or OFF	0(LOW)

❁ Drawbacks:

In this logic family, some disadvantages are there, they are:

(i) It reduces current-hogging by load transistors, which is purely because of mismatch of junction voltages. Hence it permits large fan-out.

(ii) One more problem is that load transistor in a RTL gate are driven heavily into saturation. Hence it results in long-turn-off delays.

❁ Characteristics of RTL Family

1. Speed of operation is low, i.e., Propagation delay is of the order 500 ns. Hence it cannot operate the system speed above 4 MHz.

2. For switching delay of 50 ns, the fan-in is 4 (or) 5 and fan-out is 4.

3. Because of Base resistor in transistor, the power dissipation is more. This can be reduced by introducing DCTL (Direct coupled transistor logic).

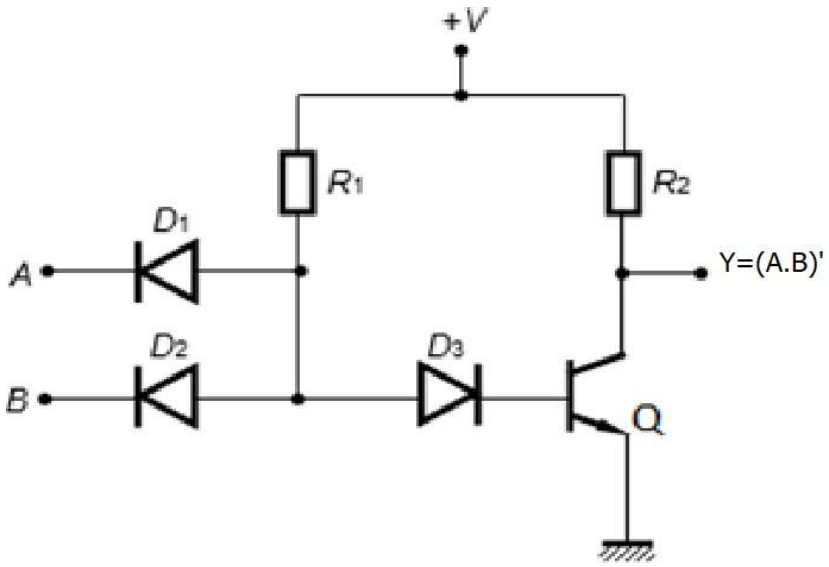
4. It is highly sensitive to temperature.

5. Poor in nose immunity

❁ 5.2 DIODE TRANSISTOR LOGIC (DTL)

This DTL logic family reduces the problem of decreasing output voltage with increasing load.

The following diagram shows the DTL NAND logic circuit using diode and transistor.



- A and B are the inputs.
- D1 D2 forms AND equivalent circuit and transistor (Q) acts as a inverter. Therefore, the combinations of AND and NOT gates forms a logical NAND circuit hence it follows the following NAND truth table.

INPUT		OUTPUT
A	B	Y=(A.B)′
0	0	1
0	1	1
1	0	1
1	1	0

- Operation:

As per the above circuit diagram, the operation is as follows:

When A = B = 1

Diodes D1 D2 are in reverse biased conditions [i.e., acts as open circuit].

Therefore D3 conduct. Hence the transistor base gets current flow and which is turned ON.

Q output in low cut-off

When A = B = 0 (or) A = 0 and B = * (0 or 1)

When all inputs are zero, the D1 and D2 is in ON condition. Hence there is no input current to base of the transistor. Q, hence it is in OFF condition. Thereby the output in collector terminal of transistor Q is high, called saturated state.

Similarly if any one of the input is low (0) that makes the above operation. So output is high (1)

Therefore, the final expression is Y = (A.B)“ Then above operation is tabulated by using functional operation table

❁ Operation table

Inputs		Device Status				Output (Y=A.B)′
A	B	D1	D2	D3	Q	
0	0	ON	ON	OFF	OFF	1(HIGH)
0	1	ON	OFF	OFF	OFF	1(HIGH)
1	0	OFF	ON	OFF	OFF	1(HIGH)
1	1	OFF	OFF	ON	ON	0(LOW)

❁ Characteristics of DTL Family

❁ Propagation Delay:

The turn-off delay is considerably more than the turn-on-delay. Hence propagation delay is 25 ns.

❁ Fan-in and Fan-out.

Fan-in is less than 8.

Fan-out is high i.e., upto 8.

❁ Noise immunity:

Noise margin is high. This is due to the additional diodes.

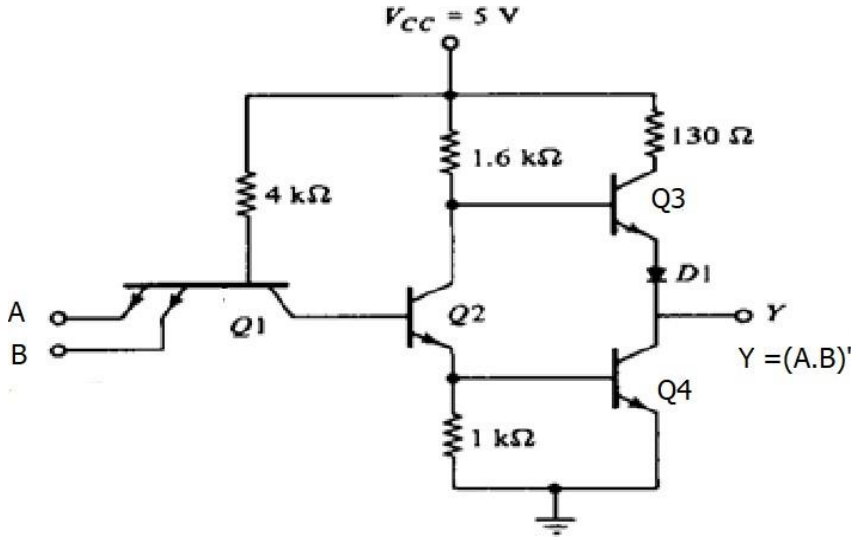
❁ Anyhow, whatever the drawbacks, can be reduced (or) improved in TTL family.

❁ 5.3 TRANSISTOR TRANSISTOR LOGIC (TTL)

The speed limitation of DTL is overcome by TTL family. It is the commonly used saturating family and hence operating speed is high.

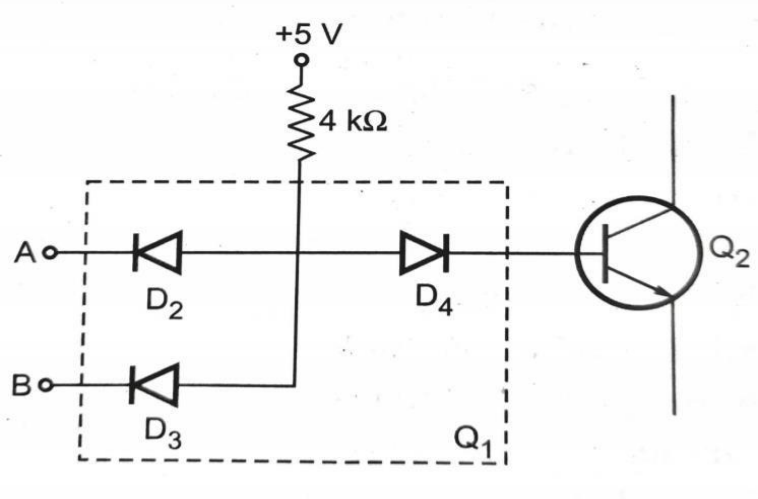
Basic gate for TTL logic is NAND gate

- 2-Input TTL NAND Gate

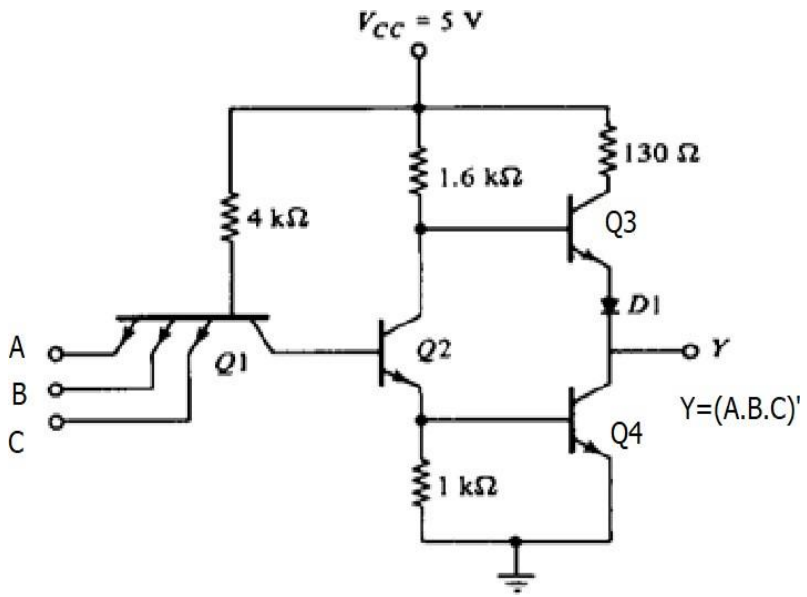


- The figure shows the circuit diagram of 2-input NAND gate. Its input structure consists of multiple-emitter transistor and output structure consists of totem-pole output. Here, Q1 is an NPN transistor having two emitters, one for each input to the gate. Although this circuit looks complex, we can simplify its analysis by using the diode equivalent of the multiple-emitter transistor Q1, as shown in figure. Diodes D2 and D3 represent the two E-B junctions of Q1 an d4 is the collector- base (C-B) junction.

- Diode equival



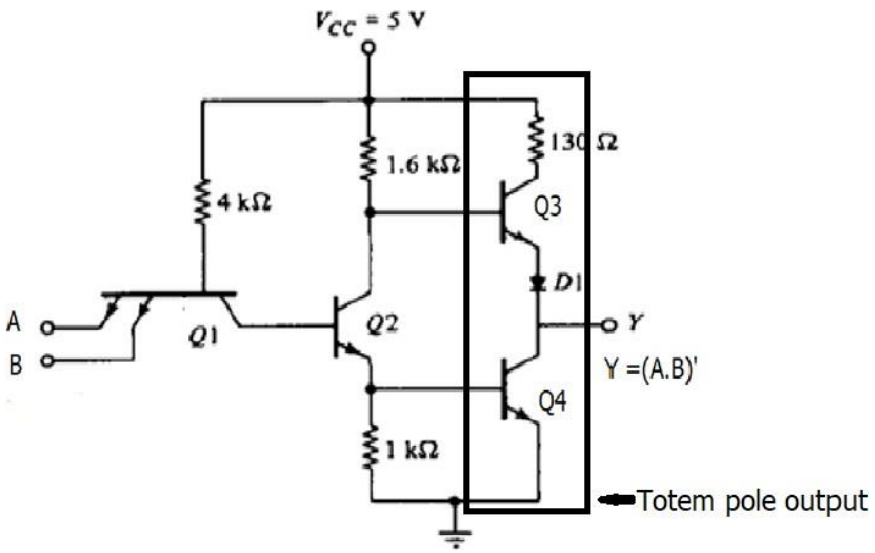
- The input voltages A and B are either LOW (ideally grounded) or HIGH (ideally + 5 volts). If either A or B or both are low, the corresponding diode conducts and the base of Q1 is pulled down to approximately 0.7 V. This reduces the base voltage of Q2 to almost zero. Therefore, Q2 cuts off. With Q2 open, Q4 goes into cut-off and the Q3 base is pulled HIGH. Since Q3 acts as an emitter follower, the Y output is pulled up to a HIGH voltage. On the other hand, when A and B both are HIGH, the emitter diode of Q1 are reversed biased making them off. This causes the collector diode D4 to go into forward conduction. This forces Q2 base to go HIGH. In turn, Q4 goes into saturation producing a low output.
- Without diode D1 in the circuit, Q3 will conduct slightly when the output is low. To prevent this diode is used; its voltage keeps the base-emitter diode of Q3 revere biased only Q4 conducts when output is low.
- 3-input TTL NAND Gate



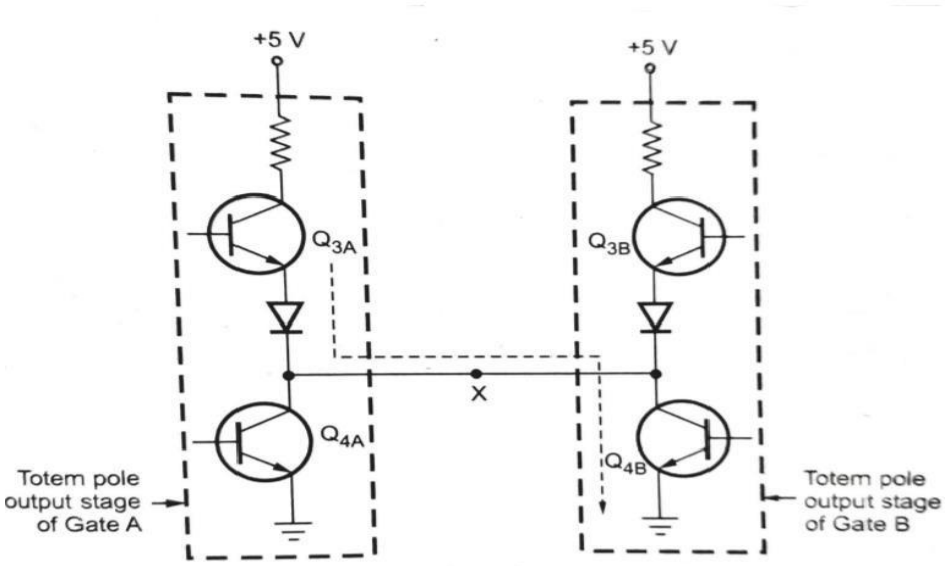
- The figure shows the three input TTL NAND gate. The operation of three input TTL NAND is same as that of two output TTL NAND gate except that is Q1 (NPN) transistor has three emitters instead of two. For three input NAND gate if all the inputs are logic 1 then and then only output is logic 0; otherwise output is logic 1. The operation is similar to the 2-input NAND gate. The table show the truth table for 3-input NAND gate.

Inputs			Device Status				Output (Y=A.B.C)'
A	B	C	Q1	Q2	Q3	Q4	
0	0	0	ON	OFF	ON	OFF	1(HIGH)
0	0 or 1	0 or 1	ON	OFF	ON	OFF	1(HIGH)
1	1	1	OFF	ON	OFF	ON	0(LOW)

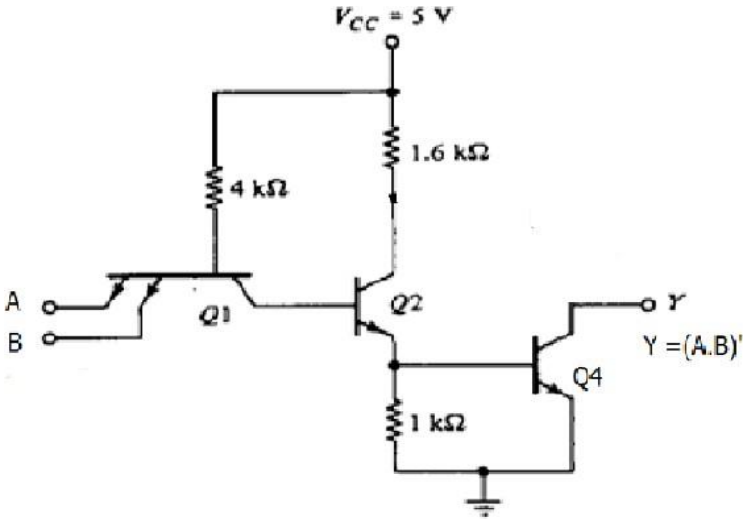
- Totem-Pole Output



- The figure shows a highlighted output configuration.
- Transistor Q3 and Q4 form a totem-pole. Such a configuration is known as active pull-up or totem pole output.
- The active pull-up formed by Q3 and Q4 has specific advantage.
- Totem-pole transistors are used because they produce LOW output impedance
- Either Q3 acts as a emitter follower (HIGH output) or Q4 is saturated (LOW output)
- When Q3 is conducting, the output impedance is approximately 70Ω; when Q4 is saturated, the output impedance is only 12Ω.
- Either way, the output impedance is low. This means that the output voltage can change quickly from one state to the other because any stray output capacitance is rapidly charged or discharged through the low output impedance. Thus the propagation delay is low in totem-pole TTL logic.
- Open-Collector Output
- One problem with totem pole output is that two outputs cannot be tied together. See the figure, where the totem pole outputs of two separate gates are connected together at point X. Suppose that the output of gate A is high (Q3A ON and Q4A OFF) and the output of gate B is low (Q3B OFF and Q4B ON). In this situation transistor Q4B acts as a load for Q3A. Since Q4B is a low resistance load, it draws high current around 55 mA. This current might not damage Q3A or Q4B immediately, but over a period of time can cause overheating and deterioration in performance and eventual device failure.



- Some TTL devices provide another type of output called open collector output. The outputs of two difference gates with open collector output can be tied together. This known as wired logic. Figure shows a 2-input NAND gate with an open-collector output eliminates the pull-up transistor Q3, D1 and R4. The output is taken from the open collector terminal of transistor Q4.



- Because the collector of Q4 is open, a gate like this will not work properly until you connect an external pull-up resistor, as shown in fig. When Q4 is ON, output is low and when Q4 is OFF output is tied to VCC through an external pull up resistor.

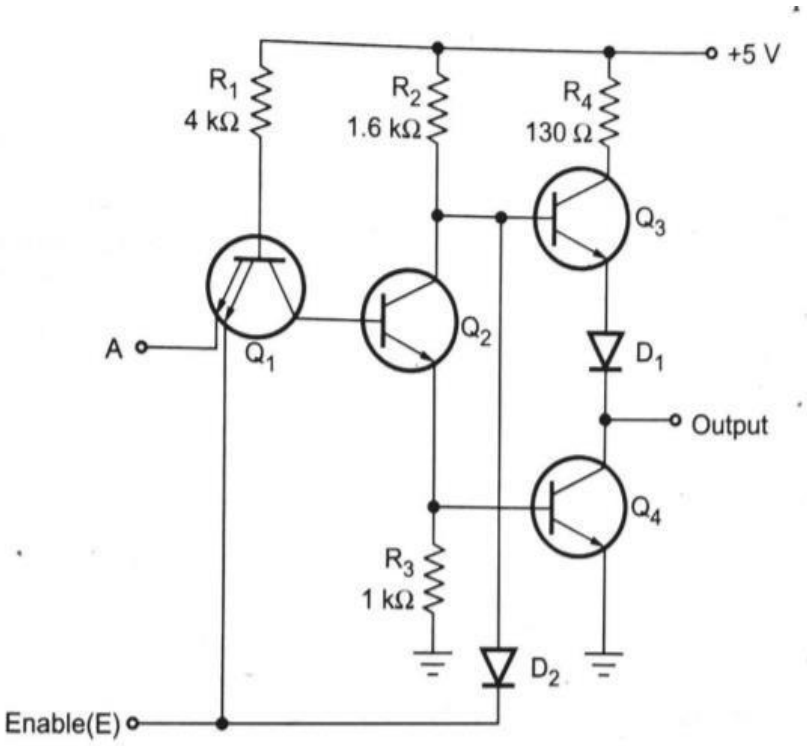
- **Comparison between Totem-Pole and Open-Collector Outputs**

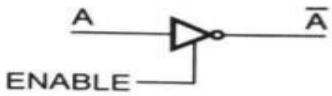
Totem-pole	Open collector
Output stage consists of pull-up transistor (Q3), diode resistor and pull-down transistor (Q4)	Output stage consists of only pull-down transistor.
External pull-up resistor is not required.	External pull-up resistor is required for proper operation of gate.
Output of two gates cannot be tied together.	Output of two gates can be tied together using wired AND technique.
Operating speed is high.	Operating speed is low.

- Table summarizes the difference between totem-pole and open collector outputs.

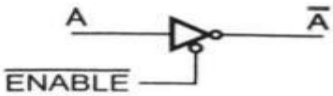
- Tri-Slate TTL Inverter

- The tristate configuration is a third type of TTL output configuration. It utilizes the high-speed operation of the totem-pole arrangement while permitting outputs to be wired-ANDed (connected together). It is called tristate TTL because it allows three possible output stages: HIGH, LOW and high impedance. We know the transistor Q3 is ON when output is HIGH and Q4 is ON when output is LOW. In the high impedance state both transistors, transistors Q3 and Q4 in the totem-pole arrangement are turned OFF. As a result, the output is open or floating, it is neither LOW nor HIGH.





Logic symbol for active high enable input



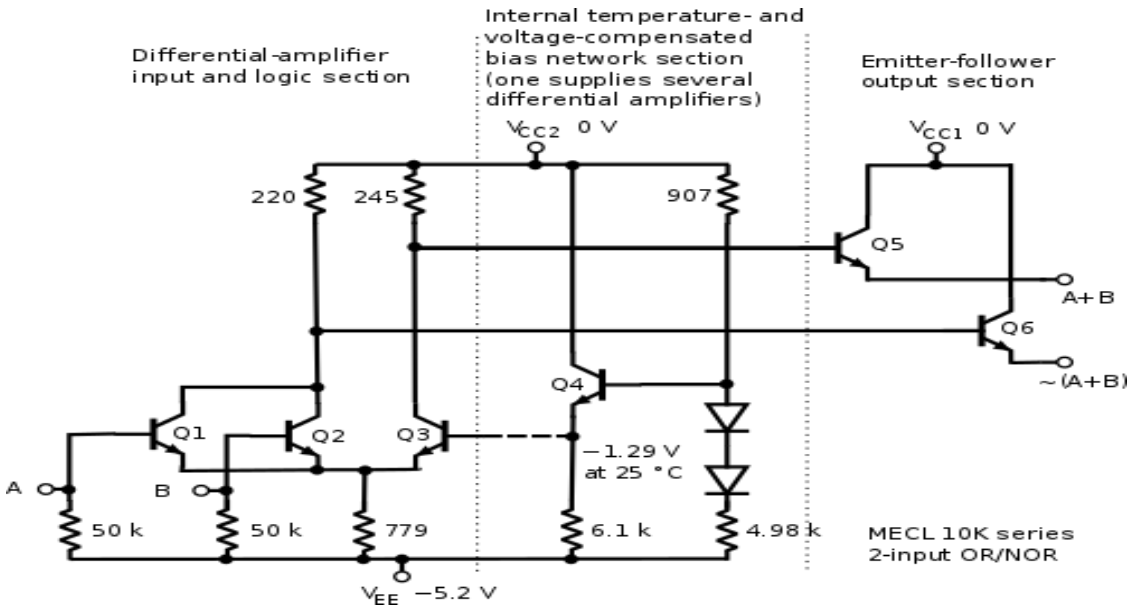
Logic symbol for active low enable input

- The figure shows the simplified circuit for tristate inverter. It has two inputs A and E.
- A is the normal logic output whereas E is an ENABLE input. When ENABLE input is HIGH, the circuit works as a normal inverter. Because when E is HIGH, the state of the transistor Q1 (either ON or OFF) depends on the logic input A, and the additional component diode is open circuited as its cathode is at logic HIGH. When ENABLE input is LOW, regardless of the state of logic input A, the base-emitter junction of Q1 is forward biased and as a result it turns ON. This shunts the current through R1 away from Q2 making it OFF. As Q2 is OFF, there is no sufficient drive for Q4 to conduct and hence Q4 turns off. The LOW at ENABLE input also forward-biases diode D2 which shunt the current away from the base of Q3, making it OFF. In this way, when ENABLE input is LOW, both transistors are OFF and output is at high impedance state. Fig shows the logic symbols for tristate inverter. In above case circuit operation is enabled when ENABLE input is HIGH. Therefore, ENBLE input is active high. The logic symbol for high enable input is shown in figure. In some circuits ENABLE input can be active LOW, i.e. circuit operates when ENABLE input is LOW. The logic symbol for active low ENABLE input is shown in the figure.

- The internal temperature – and voltage -compensated bias network supplies a reference voltage (Bias voltage V_{BB} = -1.3 V) to the differential amplifier. The best noise immunity is obtained by connecting V_{CC} to ground and V_{EE} to -5.2 V.

5.4 ECL – Emitter coupled logic:

- Emitter Coupled Logic (ECL) is a non saturated digital logic family. It achieves the propagation delay of 2ns. Its required high speed system operation. The output provides both OR and NOR functions. Each input is connected to the base of transistor. The two voltage levels are about -0.8 V for the high state and -1.8V for the low state.



- The circuit consists of
  - Differential amplifier
  - Temperature – and voltage -compensated bias network
  - Emitter follower
- The Emitter follower output requires a pull down resistor for current to flow. This is obtained from the input resistor, R_p of other similar gates or from an external resistor connected to a negative voltage supply.

- Working:
- If any of the input is high the corresponding input transistor is turned ON and transistor Q_3 is OFF.
- Ex: if V_A = -0.8V, the transistor Q_1 starts conducting, So the $V_{BE}(Q_1)$ = 0.8V. Now $V_E(Q1) = V_A - V_{BE}(Q_1)$ = -0.8V – 0.8V = -1.6V
- Next to find the $V_{BE}(Q_3)$. $V_{BE}(Q_3) = V_B(Q_3) – V_E(Q_3)$ = -1.3 –(-1.6) =0.3V. Thus the transistor Q_3 is OFF. So the transistor Q_1 is remains ON. The output voltage of the transistor Q1 is low. So the input voltage of transistor Q6 is Low. Since the transistor Q6 is a Emitter follower so the output of the transistor Q6 is also Low. This output produce the NOR output of the circuit.
- The transistor Q_3 is OFF. The output voltage of the transistor Q3 is high. So the input voltage of transistor Q5 is high. Since the transistor Q5 is a Emitter follower so the output of the transistor Q5 is also high. This output produce the OR output of the circuit.
- 2. If both the inputs are low, transistors Q1 and Q2 are turned OFF and transistor Q_3 is ON.
- Ex: if $V_A = V_B$ = -1.8V, the transistor Q_1 and Q_2 = OFF, the transistor Q_3 is ON. So the $V_{BE}(Q_3)$ = 0.8V. Now $V_E(Q_3) = V_{BB} - V_{BE}(Q_1)$ = -1.3V – 0.8V = -2.1V
- Next to find the $V_{BE}(Q_1)$ or $V_{BE}(Q_2)$. $V_{BE}(Q_1) = V_B(Q_1) – V_E(Q_1)$ = -1.8 –(-2.1) =0.3V. Thus the transistor Q_1 is OFF. So the transistor Q_3 is remains ON. The output voltage of the transistor Q_3 is low. So the input voltage of transistor Q_3 is Low. Since the transistor Q5 is a Emitter follower so the output of the transistor Q5 is also Low. This output produce the OR output of the circuit.
- The transistor Q_1 is OFF. The output voltage of the transistor Q1 is high. So the input voltage of transistor Q6 is high. Since the transistor Q6 is a Emitter follower so the output of the transistor Q6 is also high. This output produce the NOR output of the circuit.

❁ Operation Table

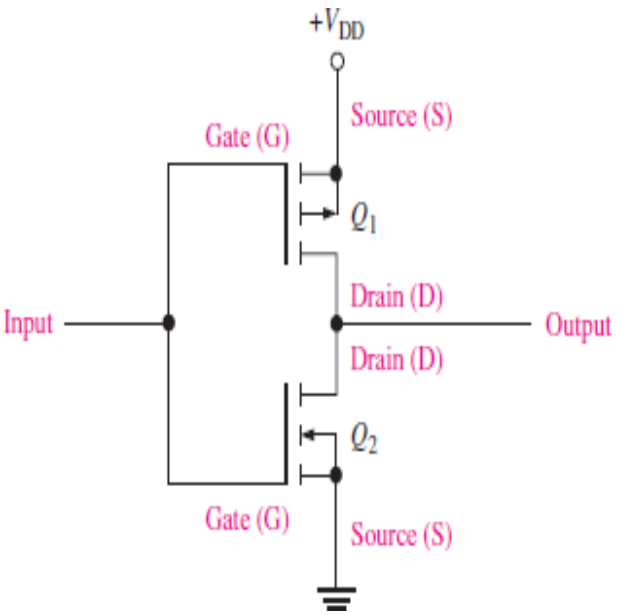
INPUT		Q1	Q2	Q3	Q5	Q6	NOR OUTPUT	OR OUTPUT
V_A	V_B							
-1.8V	-1.8V	OFF	OFF	ON	OFF	ON	HIGH	LOW
-1.8V	-0.8V	OFF	ON	OFF	ON	OFF	LOW	HIGH
-0.8V	-1.8V	ON	OFF	OFF	ON	OFF	LOW	HIGH
-0.8V	-0.8V	ON	ON	OFF	ON	OFF	LOW	HIGH

❁ 5.5 CMOS Families

❁ Complementary MOS (CMOS) logic uses the MOSFET in complementary pairs as its basic element.

❁ A complementary pair uses both p-channel and n-channel enhancement MOSFETs

❁ CMOS AS INVERTER



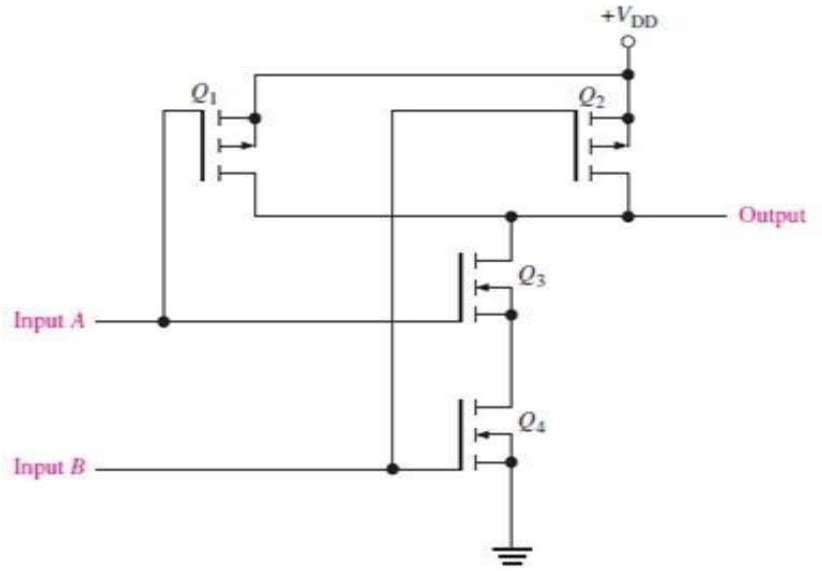
A	Q1 PMOS	Q2 NMOS	Y
0	ON	OFF	1
1	OFF	ON	0

❁ CMOS AS NAND LOGIC

❁ When both inputs are LOW, Q1 and Q2 are on, and Q3 and Q4 are off.

❁ The output is pulled HIGH through the on resistance of Q1 and Q2 in parallel.

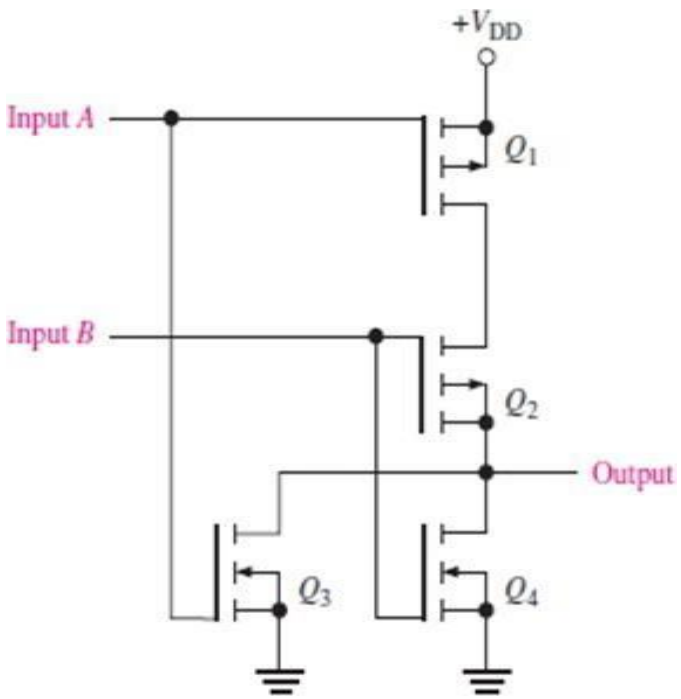
❁ When input A is LOW and input B is HIGH, Q1 and Q4 are on, and Q2 and Q3 are off.



A (Q1 & Q3)	B (Q2 & Q4)	Q1 PMOS	Q2 PMOS	Q3 NMOS	Q4 NMOS	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	1
1	0	OFF	ON	ON	OFF	1
1	1	OFF	OFF	ON	ON	0

- The output is pulled HIGH through the low on resistance of Q1.
- When input A is HIGH and input B is LOW, Q1 and Q4 are off, and Q2 and Q3 are on.
- The output is pulled HIGH through the low on resistance of Q2.
- When both inputs are HIGH, Q1 and Q2 are off, and Q3 and Q4 are on.
- The output is pulled LOW through the on resistance of Q3 and Q4 in series to ground.

- **CMOS AS NOR LOGIC**
- When both inputs are LOW, Q1 and Q2 are on, and Q3 and Q4 are off.
- The output is pulled HIGH through the on resistance of Q1 and Q2 in series.



A (Q1 & Q3)	B (Q2 & Q4)	Q1 PMOS	Q2 PMOS	Q3 NMOS	Q4 NMOS	Y
0	0	ON	ON	OFF	OFF	1
0	1	ON	OFF	OFF	ON	0
1	0	OFF	ON	ON	OFF	0
1	1	OFF	OFF	ON	ON	0

- When input A is LOW and input B is HIGH, Q1 and Q4 are on, and Q2 and Q3 are off.
- The output is pulled LOW through the low on resistance of Q4 to ground.
- When input A is HIGH and input B is LOW, Q1 and Q4 are off, and Q2 and Q3 are on.
- The output is pulled LOW through the on resistance of Q3 to ground.
- When both inputs are HIGH, Q1 and Q2 are off, and Q3 and Q4 are on. The output is
- pulled LOW through the on resistance of Q3 and Q4 in parallel to ground.