**UNIT I**

**MINIMIZATION TECHNIQUES AND LOGIC GATES**

# Pre requisition:

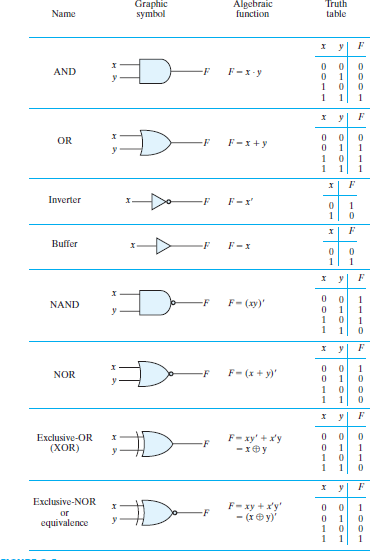
# i. Basic Electronics.

# ii. Basic Number System.

# iii. Basic Knowledge in Boolean Algebra.

# Digital Logic Gates

Boolean functions are expressed in terms of AND, OR, and NOT operations, it is easier to implement a Boolean function with these type of gates.



# Properties of XOR Gates

* + XOR (also ) : the “not-equal” function
  + XOR(X,Y) = X  Y = X’Y + XY’
  + Identities:

– X  0 = X

– X  1 = X’

– X  X = 0

– X  X’ = 1

* + Properties:

– X  Y = Y  X

– (X  Y)  W = X  ( Y  W)

# Universal Logic Gates

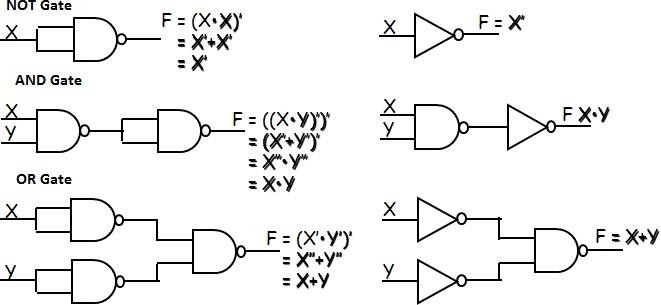
NAND and NOR gates are called Universal gates. All fundamental gates (NOT, AND, OR) can be realized by using either only NAND or only NOR gate. A universal gate provides flexibility and offers enormous advantage to logic designers.

### NAND as a Universal Gate

NAND Known as a “universal” gate because ANY digital circuit can be implemented with NAND

gates alone.

To prove the above, it suffices to show that AND, OR, and NOT can be implemented using NAND gates only.



**Boolean Algebra:** In 1854, George Boole developed an algebraic system now called Boolean algebra. In 1938, Claude E. Shannon introduced a two‐valued Boolean algebra called switching algebra that represented the properties of bistable electrical switching circuits. For the formal definition of Boolean algebra, we shall employ the postulates formulated by E. V. Huntington in 1904.

Boolean algebra is a system of mathematical logic. It is an algebraic system consisting of the set of elements (0, 1), two binary operators called OR, AND, and one unary operator NOT. It is the basic mathematical tool in the analysis and synthesis of switching circuits. It is a way to express logic functions algebraically.

Boolean algebra, like any other deductive mathematical system, may be defined with aset of elements, a set of operators, and a number of unproved axioms or postulates. A *set* of elements is anycollection of objects having a common property. If **S** is a set and ***x*** and ***y*** are certain objects, then **x** Î **S**denotes that ***x*** is a member of the set **S**, and ***y*** Ï**S** denotes that ***y*** is not an element of **S**. A set with adenumerable number of elements is specified by braces: **A** = {1,2,3,4}, *i.e.* the elements of set **A** are thenumbers 1, 2, 3, and 4. A *binary operator* defined on a set S of elements is a rule that assigns to each pair ofelements from S a unique element from S.\_ Example: In *a*\**b=c*, we say that \* is a binary operator if it specifies a rule for finding *c* from the pair (*a*,*b*)and also if *a*, *b*, *c* Î S.

**Axioms and laws of Boolean algebra**

Axioms or Postulates of Boolean algebra are a set of logical expressions that we accept without proof and upon which we can build a set of useful theorems.

|  |  |  |  |
| --- | --- | --- | --- |
|  | AND Operation | OR Operation | NOT Operation |
| Axiom1 : | 0.0=0 | 0+0=0 | 0=1 |
| Axiom2: | 0.1=0 | 0+1=1 | 1=0 |
| Axiom3: | 1.0=0 | 1+0=1 |  |
| Axiom4: | 1.1=1 | 1+1=1 |  |

|  |  |
| --- | --- |
| **AND Law** | **OR Law** |
| Law1: A.0=0 (Null law) | Law1: A+0=A |
| Law2: A.1=A (Identity law) | Law2: A+1=1 |
| Law3: A.A=A (Impotence law) | Law3: A+A=A (Impotence law) |

**CLOSURE:** The Boolean system is *closed* with respect to a binary operator if for every pair of Boolean values,it produces a Boolean result. For example, logical AND is closed in the Boolean system because it accepts only Boolean operands and produces only Boolean results.

\_ A set *S* is closed with respect to a binary operator if, for every pair of elements of *S*, the binary operator specifies a rule for obtaining a unique element of *S*.

\_ For example, the set of natural numbers N = {1, 2, 3, 4, … 9} is closed with respect to the binary operator plus (+) by the rule of arithmetic addition, since for any *a*, *b* Î N we obtain a unique *c* Î N by the operation *a* + *b* = c.

**ASSOCIATIVE LAW:**

A binary operator \* on a set *S* is said to be associative whenever (*x \* y*) \* *z* = *x* \* (*y* \* *z*) for all *x*, *y*, *z* Î S, forall Boolean values x, y and z.

**COMMUTATIVE LAW:**

A binary operator \* on a set *S* is said to be commutative whenever x *\* y* = *y* \* *x* for all *x*, *y*, *z* є S

**IDENTITY ELEMENT:**

A set ***S*** is said to have an identity element with respect to a binary operation \* on S if there exists an element

*e є* S with the property *e \* x* = *x* \* *e* = *x* for every *x* є S

**BASIC IDENTITIES OF BOOLEAN ALGEBRA**

* *Postulate 1(Definition)*: A Boolean algebra is a closed algebraic system containing a set *K* of two or more elements and the two operators · and + which refer to logical AND and logical OR •*x + 0 = x*
* *x* · *0 = 0*
* *x + 1 = 1*
* *x* · *1 = 1*
* *x + x = x*
* *x · x = x*
* *x + x’ = x*
* *x · x’ = 0*
* *x + y = y + x*
* *xy = yx*
* *x + ( y + z ) = ( x + y ) + z*
* *x (yz) = (xy) z*
* *x ( y + z ) = xy + xz*
* *x + yz = ( x + y )( x + z)*
* *( x + y )’ = x’ y’*
* *( xy )’ = x’ + y’*
* *(x’)’ = x*

**DeMorgan's Theorem**

**(a)** (*a + b*)' = *a*'*b*'

**(b)** (*ab*)' = *a*' + *b*'

**Generalized DeMorgan's Theorem**

(a) (*a + b + … z*)' = *a*'*b*' *… z*'

(b) (*a.b … z*)' = *a*' *+ b*' *+ … z*‘

### Basic Theorems and Properties of Boolean algebra Commutative law

Law1: A+B=B+A Law2: A.B=B.A

### Associative law

Law1: A + (B +C) = (A +B) +C Law2: A(B.C) = (A.B)C

### Distributive law

Law1: A.(B + C) = AB+ AC Law2: A + BC = (A + B).(A +C)

### Absorption law

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Law1: | A +AB =A |  | Law2: | A(A +B) = A |
| Solution: | A(1+B) |  | Solution: | A.A+A.B |
|  | A |  |  | A+A.B |
|  |  |  |  | A(1+B) |
|  |  |  |  | A |

**Consensus Theorem**

Theorem1. AB+ A’C + BC = AB + A’C Theorem2. (A+B). (A’+C).(B+C) =(A+B).( A’+C)

The BC term is called the consensus term and is redundant. The consensus term is formed from a PAIR OF TERMS in which a variable (A) and its complement (A’) are present; the consensus term is formed by multiplying the two terms and leaving out the selected variable and its complement

Consensus Theorem1 Proof:

AB+A’C+BC=AB+A’C+(A+A’)BC

=AB+A’C+ABC+A’BC

=AB(1+C)+A’C(1+B)

= AB+ A’C

### Principle of Duality

Each postulate consists of two expressions statement one expression is transformed into the other by interchanging the operations (+) and (⋅) as well as the identity elements 0 and 1.

Such expressions are known as duals of each other.

If some equivalence is proved, then its dual is also immediately true.

If we prove: (x.x)+(x’+x’)=1, then we have by duality: (x+x)⋅(x’.x’)=0

The Huntington postulates were listed in pairs and designated by part (a) and part (b) in below table.

### Table for Postulates and Theorems of Boolean algebra

|  |  |
| --- | --- |
| **Part-A** | **Part-B** |
| A+0=A | A.0=0 |
| A+1=1 | A.1=A |
| A+A=A (Impotence law) | A.A=A (Impotence law) |
| A+ A̅=1 | A. A̅=0 |
| A̅=A (double inversion law) | -- |
| **Commutative law:** A+B=B+A | A.B=B.A |
| **Associative law**: A + (B +C) = (A +B) +C | A(B.C) = (A.B)C |
| **Distributive law**: A.(B + C) = AB+ AC | A + BC = (A + B).(A +C) |
| **Absorption law**: A +AB =A | A(A +B) = A |
| **DeMorgan Theorem:**  (A+B) = A̅ . B̅ | (A.B) = = A̅ + B̅ |
| **Redundant Literal Rule:** A+ A̅. B=A+B | A.(A̅A+B)=AB |
| **Consensus Theorem:** AB+ A’C + BC = AB + A’C | (A+B). (A’+C).(B+C) =(A+B).( A’+C) |

**Boolean Function**

Boolean algebra is an algebra that deals with binary variables and logic operations.

A Boolean function described by an algebraic expression consists of binary variables, the constants 0 and 1, and the logic operation symbols.

For a given value of the binary variables, the function can be equal to either 1 or 0.

F(vars) = *expression*





Set of binary Variables Operators (+, •, ‘) Constants (0, 1) Groupings (parenthesis) Variables

Consider an example for the Boolean function

F1 = x + y’z

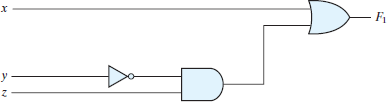
The function F1 is equal to 1 if x is equal to 1 or if both y’ and z are equal to 1. F1 is equal to 0 otherwise. The complement operation dictates that when y’ = 1, y = 0. Therefore, F1 = 1 if x = 1 or if y = 0 and z = 1.

A Boolean function expresses the logical relationship between binary variables and is evaluated by determining the binary value of the expression for all possible values of the variables.

A Boolean function can be represented in a truth table. The number of rows in the truth table is 2n, where n is the number of variables in the function. The binary combinations for the truth table are obtained from the binary numbers by counting from 0 through 2n - 1.

Truth Table for F1

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | z | F1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 |



Gate Implementation of F1 = x + y’z

### Note:

Q: Let a function F() depend on n variables. How many rows are there in the truth table of F() ? A: 2n rows, since there are 2n possible binary patterns/combinations for the n variables.

Truth Tables

* + Enumerates all possible combinations of variable values and the corresponding function value
  + Truth tables for some arbitrary functions

F1(x,y,z), F2(x,y,z), and F3(x,y,z) are shown to the below.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| x | y | z | F1 | F2 | F3 |
| 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |

* + Truth table: a unique representation of a Boolean function
  + If two functions have identical truth tables, the functions are equivalent(and vice- versa).
  + Truth tables can be used to prove equality theorems.
  + However, the size of a truth table grows exponentially with the number ofvariables involved, hence unwieldy. This motivates the use of Boolean Algebra.

**Boolean expressions-NOT unique** Unlike truth tables, expressions epresenting a Boolean function are NOT unique.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | y | z | F | G |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |

* Example:

– F(x,y,z) = x’•y’•z’ + x’•y•z’ +

x•y•z’

– G(x,y,z) = x’•y’•z’ + y•z’

* The corresponding truth tables for F() and G() are to the right. They are identical.
* Thus, F() = G()

# Algebraic Manipulation (Minimization of Boolean function)

## Boolean algebra is a useful tool for simplifying digital circuits.

* Why do it? Simpler can mean cheaper, smaller, faster.
* Example: Simplify F = x’yz + x’yz’ + xz.

F= x’yz + x’yz’ + xz

= x’y(z+z’) + xz

= x’y•1 + xz

= x’y + xz

* Example: Prove

x’y’z’ + x’yz’ + xyz’ = x’z’ + yz’

* **Proof:**

x’y’z’+ x’yz’+ xyz’

= x’y’z’ + x’yz’ + x’yz’ + xyz’

= x’z’(y’+y) + yz’(x’+x)

= x’z’•1 + yz’•1

= x’z’ + yz’

**Complement of a Function**

* + The complement of a function is derived by interchanging (• and +), and (1 and 0), and complementing each variable.
  + Otherwise, interchange 1s to 0s in the truth table column showing F.
  + The *complement* of a function IS NOT THE SAME as the *dual* of afunction. Example
* Find G(x,y,z), the complement of F(x,y,z) = xy’z’ + x’yz Ans: G = F’ = (xy’z’ + x’yz)’

= (xy’z’)’ • (x’yz)’ *DeMorgan*

= (x’+y+z) • (x+y’+z’) *DeMorgan* again

**Note:** The complement of a function can also be derived by finding the function’s *dual,* and then complementing all of the literals

# Canonical and Standard Forms

We need to consider formal techniques for the simplification of Boolean functions. Identical functions will have exactly the same canonical form.

* + Minterms and Maxterms
  + Sum-of-Minterms and Product-of- Maxterms
  + Product and Sum terms
  + Sum-of-Products (SOP) and Product-of-Sums (POS)

# Definitions

**Literal:** A variable or its complement **Product term:** literals connected by • **Sum term:** literals connected by +

**Minterm:** a product term in which all the variables appear exactly once, either complemented or uncomplemented.

**Maxterm:** a sum term in which all the variables appear exactly once, either complemented or uncomplemented.

**Canonical form:** Boolean functions expressed as a sum of Minterms or product of Maxterms are said to be in canonical form.

# Minterm

* + Represents exactly one combination in the truth table.
  + Denoted by mj, where j is the decimal equivalent of the minterm’s corresponding binary

combination (bj).

* + A variable in mj is complemented if its value in bj is 0, otherwise is uncomplemented.

Example: Assume 3 variables (A, B, C), and j=3. Then, bj = 011 and its corresponding minterm is denoted by mj = A’BC

# Maxterm

* + Represents exactly one combination in the truth table.
  + Denoted by *Mj*, where *j* is the decimal equivalent of the maxterm’s corresponding binary

combination *(bj*)*.*

* + A variable in *Mj* is complemented if its value in *bj* is 1, otherwise is uncomplemented.

Example: Assume 3 variables (A, B, C), and *j*=3. Then, b*j* = 011 and its corresponding maxterm is denoted by M*j* = A+B’+C’

# Truth Table notation for Minterms and Maxterms

* Minterms and Maxterms are easy to denote using a truthtable. Example: Assume 3 variables x,y,z (order is fixed)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| x | y | z | Minterm | Maxterm |
| 0 | 0 | 0 | x’y’z’ = m0 | x+y+z = M0 |
| 0 | 0 | 1 | x’y’z = m1 | x+y+z’ = M1 |
| 0 | 1 | 0 | x’yz’ = m2 | x+y’+z = M2 |
| 0 | 1 | 1 | x’yz = m3 | x+y’+z’= M3 |
| 1 | 0 | 0 | xy’z’ = m4 | x’+y+z = M4 |
| 1 | 0 | 1 | xy’z = m5 | x’+y+z’ = M5 |
| 1 | 1 | 0 | xyz’ = m6 | x’+y’+z = M6 |
| 1 | 1 | 1 | xyz = m7 | x’+y’+z’ = M7 |

# Canonical Forms

* Every function F() has two canonical forms:
  + Canonical Sum-Of-Products (sum of minterms)
  + Canonical Product-Of-Sums (product of maxterms) Canonical Sum-Of-Products:

The minterms included are those mj such that F( ) = 1 in row j of the truth table for F( ). Canonical Product-Of-Sums:

The maxterms included are those Mj such that F( ) = 0 in row j of the truth table for F( ).

# Example

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | c | f1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Consider a Truth table for f1(a,b,c) at right The canonical sum-of-products form for f1 is f1(a,b,c) = m1 + m2 + m4 + m6

= a’b’c + a’bc’ + ab’c’ + abc’

The canonical product-of-sums form for f1 is f1(a,b,c) = M0 • M3 • M5 • M7

= (a+b+c)•(a+b’+c’)• (a’+b+c’)•(a’+b’+c’).

* Observe that: mj = Mj’

# Shorthand: ∑ and ∏

* f1(a,b,c) = ∑ m(1,2,4,6), where ∑ indicates that this is a sum-of-products form, and m(1,2,4,6) indicates that the minterms to be included are m1, m2, m4, and m6.
* f1(a,b,c) = ∏ M(0,3,5,7), where ∏ indicates that this is a product-of-sums form, and M(0,3,5,7) indicates that the maxterms to be included are M0, M3, M5, and M7.
* Since mj = Mj’ for any *j*,

∑ m(1,2,4,6) = ∏ M(0,3,5,7) = f1(a,b,c)

•

# Conversion between Canonical Forms

* Replace ∑ with ∏ (or *vice versa*) and replace those *j’*s that appeared in the original form with those that do not.
* Example:

f1(a,b,c)= a’b’c + a’bc’ + ab’c’ + abc’

= m1 + m2 + m4 + m6

= ∑(1,2,4,6)

= ∏(0,3,5,7)

= (a+b+c)•(a+b’+c’)•(a’+b+c’)•(a’+b’+c’)

# Standard Forms

Another way to express Boolean functions is in standard form. In this configuration, the terms that form the function may contain one, two, or any number of literals.

There are two types of standard forms: the sum of products and products of sums.

The sum of products is a Boolean expression containing AND terms, called product terms, with one or more literals each. The sum denotes the ORing of these terms. An example of a function expressed as a sum of products is

F1 = y’ + xy + x’yz’

The expression has three product terms, with one, two, and three literals. Their sum is, in effect, an OR operation.

A product of sums is a Boolean expression containing OR terms, called sum terms. Each term may have any number of literals. The product denotes the ANDing of these terms. An example of a function expressed as a product of sums is

F2 = x(y’ + z)(x’ + y + z’)

This expression has three sum terms, with one, two, and three literals. The product is an AND operation.

# Conversion of SOP from standard to canonical form Example-1.

## Express the Boolean function F = A + B’C as a sum of minterms.

Solution: The function has three variables: A, B, and C. The first term A is missing two variables; therefore,

A = A(B + B’) = AB + AB’

This function is still missing one variable, so

A = AB(C + C’) + AB’ (C + C’)

= ABC + ABC’ + AB’C + AB’C’

The second term B’C is missing one variable; hence, B’C = B’C(A + A’) = AB’C + A’B’C

Combining all terms, we have

F = A + B’C

= ABC + ABC’ + AB’C + AB’C’+ A’B’C

But AB’C appears twice, and according to theorem (x + x = x), it is possible to remove one of those

occurrences. Rearranging the minterms in ascending order, we finally obtain

F = A’B’C + AB’C + AB’C + ABC’ + ABC

= m1 + m4 + m5 + m6 + m7

When a Boolean function is in its sum‐of‐minterms form, it is sometimes convenient to express the

function in the following brief notation:

F(A, B, C) = ∑m (1, 4, 5, 6, 7)

### Example-2.

Express the Boolean function F = xy + x’z as a product of maxterms.

Solution: First, convert the function into OR terms by using the distributive law:

F = xy + x’z = (xy + x’)(xy + z)

= (x + x’)(y + x’)(x + z)(y + z)

= (x’+ y)(x + z)(y + z)

The function has three variables: x, y, and z. Each OR term is missing one variable; therefore,

x’+ y = x’ + y + zz’ = (x’ + y + z)(x’ + y + z’) x + z = x + z + yy’ = (x + y + z)(x + y’ + z)

y + z = y + z + xx’ = (x + y + z)(x’ + y + z)

Combining all the terms and removing those which appear more than once, we finally obtain

F = (x + y + z)(x + y’ + z)(x’ + y + z)(x’ + y + z)

F= M0M2M4M5

A convenient way to express this function is as

follows: F(x, y, z) = πM(0, 2, 4, 5)

The product symbol, π, denotes the ANDing of maxterms; the numbers are the indices of the maxterms of the function.

**Two-variable k-map:**

A two-variable k-map can have 22=4 possible combinations of the input variables A and

1. each of these combinations (in the SOP form) is called a minterm. The minterm may be represented in terms of their decimal designations – m0 for A’B’ , m1 for A’ B,m2 for AB’ and m3 for AB, assuming that A represents the MSB. The letter m stands for minterm and the subscript represents the decimal designation of the minterm. The presence or absence of a minterm in the expression indicates that the output of the logic circuit assumes logic 1 or logic 0 level for that combination of input variables.

The expression f= A’B’+ AB’+AB , it can be expressed using minerm as F= m0+m2+m3=∑m(0,2,3)

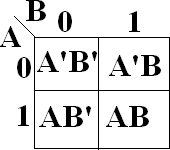
Using Truth Table:

|  |  |  |
| --- | --- | --- |
| Minterm | Inputs  A B | Output F |
| 0 | 0 0 | 1 |
| 1 | 0 1 | 0 |
| 2 | 1 0 | 1 |
| 3 | 1 1 | 1 |

A 1 in the output contains that particular minterm in its sum and a 0 in that column indicates that the particular minterm does not appear in the expression for output .This information can also be indicated by a two-variable k-map.

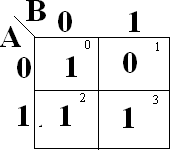
### Mapping of SOP Expresions:

A two-variable k-map has 22=4 squares .These squares are called cells. Each square on the k- map represents a unique minterm. The minterm designation of the squares are placed in any square, indicates that the corresponding minterm does output expressions. And a 0 or no entry in any square indicates that the corresponding minterm does not appear in the expression for output.

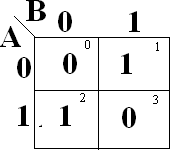


The minterms of a two-variable k-map

The mapping of the expressions =∑m(0,2,3)is



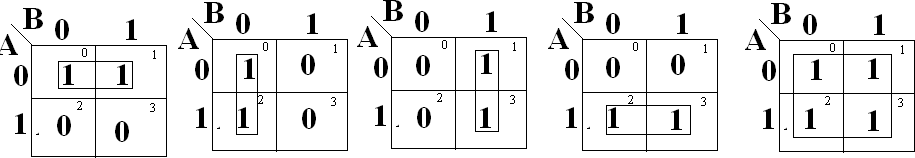
k-map of ∑m(0,2,3)

**EX**: Map the expressions f= B+A F= m1+m2=∑m(1,2)The k-map is

### Minimizations of SOP expressions:

To minimize Boolean expressions given in the SOP form by using the k-map, look for adjacent adjacent squares having 1‘s minterms adjacent to each other, and combine them to form larger squares to eliminate some variables. Two squares are said to be adjacent to each other, if their minterms differ in only one variable. (i.e, B & A differ only in one variable. so theymay be combined to form a 2-square to eliminate the variable B.similarly all other.

The necessary condition for adjacency of minterms is that their decimal designations must differ by a power of 2. A minterm can be combined with any number of minterms adjacent to it to form larger squares. Two minterms which are adjacent to each other can be combined to form a bigger square called a 2-square or a pair. This eliminates one variable – the variable that is not common to both the minterms.



f1=A’ f2=B’ f3=B f4=A f5=1

The possible minterm groupings in a two-variable k-map.

Two 2-squares adjacent to each other can be combined to form a 4-square. A 4-square eliminates 2 variables. A 4-square is called a quad. To read the squares on the map after minimization, consider only those variables which remain constant through the square, and ignore the variables which are varying. Write the non complemented variable if the variable is remaining constant as a 1, and the complemented variable if the variable is remaining constant as a 0, and write the variables as a product term. In the above figure f1 read as , because, along the square , A remains constant as a 0, that is , as , where as B is changing from 0 to 1.

The main criterion in the design of a digital circuit is that its cost should be as low as possible. For that the expression used to realize that circuit must be minimal.Since the cost is proportional to number of gate inputs in the circuit in the circuit, an expression is considered minimal only if it corresponds to the least possible number of gate inputs. & there is no guarantee for that k-map in SOP is the real minimal. To obtain real minimal expression, obtain the minimal expression both in SOP & POS form form by using k-maps and take the minimal of these two minimals.

The 1‘s on the k-map indicate the presence of minterms in the output expressions, where as the 0s indicate the absence of minterms .Since the absence of a minterm in the SOP expression means the presense of the corresponding maxterm in the POS expression of the same .when a SOP expression is plotted on the k-map, 0s or no entries on the k-map represent the maxterms. To obtain the minimal expression in the POS form, consider the 0s on the k-map and follow the procedure used for combining 1s. Also, since the absence of a maxterm in the POS expression means the presence of the corresponding minterm in the SOP expression of the same , when a POS expression is plotted on the k-map, 1s or no entries on the k-map represent theminterms.

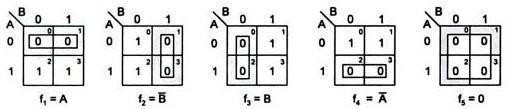
### Mapping of POS expressions:

Each sum term in the standard POS expression is called a maxterm. A function in two variables (A, B) has four possible maxterms, A+B, A+B’, A’+B, A’+B’

. They are represented as M0, M1, M2, and M3respectively. The uppercase letter M stands for maxterm and its subscript denotes the decimal designation of that maxterm obtained by treating the non-complemented variable as a 0 and the complemented variable as a 1 and putting them side by side for reading the decimal equivalent of the binary number so formed.

For mapping a POS expression on to the k-map, 0s are placed in the squares corresponding to the maxterms which are presented in the expression an d1s are placed in the squares corresponding to the maxterm which are not present in the expression. The decimal designation of the squares of the squares for maxterms is the same as that for the minterms. A two-variable k-map & the associated maxterms are asthe maxterms of a two-variable k-map

The possible maxterm groupings in a two-variable k-map



### Minimization of POS Expressions:

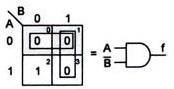
To obtain the minimal expression in POS form, map the given POS expression on to the K-map and combine the adjacent 0s into as large squares as possible. Read the squares putting the complemented variable if its value remains constant as a 1 and the non-complemented variable if its value remains constant as a 0 along the entire square ( ignoring the variables which do not remain constant throughout the square) and then write them as a sum term.

Various maxterm combinations and the corresponding reduced expressions are shown in figure. In this f1 read as A because A remains constant as a 0 throughout the square and B changes from a 0 to a 1. f2 is read as B‘ because B remains constant along the square as a 1 and A changes from a 0 to a 1. f5

Is read as a 0 because both the variables are changing along the square.

**Ex:** Reduce the expression f=(A+B)(A+B‘)(A‘+B‘) using mapping.

The given expression in terms of maxterms is f=πM(0,1,3). It requires two gates inputs for realization of the reduced expression as

F=AB‘

K-map in POS form and logic diagram

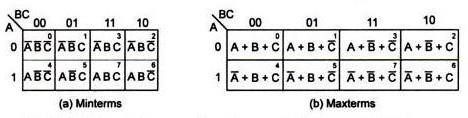
In this given expression ,the maxterm M2 is absent. This is indicated by a 1 on the k-map. The corresponding SOP expression is ∑m2 or AB‘. This realization is the same as that for the POS form.

### Three-variable K-map:

A function in three variables (A, B, C) expressed in the standard SOP form can have eight possible combinations: A’B’C’ A’B’C, A’BC’, A’BC, AB’C’, AB’C,, ABC’, and ABC,. Each one of these combinations designate d by m0,m1,m2,m3,m4,m5,m6, and m7, respectively, is called a minterm. A is the MSB of the minterm designator and C is the LSB.

In the standard POS form, the eight possible combinations are: A+B+C, A+B+C’, A+B’+C,A+B’+C’ ,A’+B+C,A’+B+C’,A’+B’+C,A’+B’+C’. Each one of these combinations designated by M0, M1, M2, M3, M4, M5, M6, and M7respectively is called a maxterm. A is the MSB of the maxterm designator and C is the LSB.

A three-variable k-map has, therefore, 8(=23) squares or cells, and each square on the map represents a minterm or maxterm as shown in figure. The small number on the top right corner of each cell indicates the minterm or maxterm designation.



The three-variable k-map.

The binary numbers along the top of the map indicate the condition of B and C for each column. The binary number along the left side of the map against each row indicates the condition of A for that row. For example, the binary number 01 on top of the second column in fig indicates that the variable B appears in complemented form and the variable C in non- complemented form in all the minterms in that column. The binary number 0 on the left of the first row indicates that the variable A appears in complemented form in all the minterms in that row, the binary numbers along the top of the k-map are not in normal binary order. They are, infact, in the Gray code. This is to ensure that twophysically adjacent squares are really adjacent, i.e., their minterms or maxterms differ by only one variable.

;

### Minimization of SOP and POS expressions:

For reducing the Boolean expressions in SOP (POS) form plotted on the k-map, look at the 1s (0s) present on the map. These represent the minterms (maxterms). Look for the minterms (maxterms) adjacent to each other, in order to combine them into larger squares. Combining of adjacent squares in a k-map containing 1s (or 0s) for the purpose of simplification of a SOP (or POS)expression is called *looping*. Some of the minterms (maxterms) may have many adjacencies. Always start with the minterms (maxterm) with the least number of adjacencies and try to form as large as large a square as possible. The larger must form a geometric square or rectangle. They can be formed even by wrapping around, but cannot be formed by using diagonal configurations. Next consider the minterm (maxterm) with next to the least number of adjacencies and form as large a square as possible. Continue this till all the minterms (maxterms) are taken care of . A minterm (maxterm) can be part of any number of squares if it is helpful in reduction. Read the minimal expression from the k-map, corresponding to the squares formed. There can be more than one minimal expression.

Two squares are said to be adjacent to each other (since the binary designations along the top of the map and those along the left side of the map are in Gray code), if they are physically adjacent to each other, or can be made adjacent to each other by wrapping around. For squares to be combinable into bigger squares it is essential but not sufficient that their minterm designations must differ by a power of two.

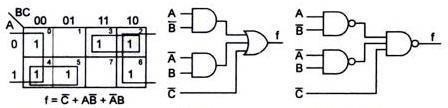
General procedure to simplify the Boolean expressions:

* 1. Plot the k-map and place 1s(0s) corresponding to the minterms (maxterms) of the SOP (POS) expression.
  2. Check the k-map for 1s(0s) which are not adjacent to any other 1(0). They are isolated minterms(maxterms) . They are to be read as they are because they cannot be combined even into a 2-square.
  3. Check for those 1s(0S) which are adjacent to only one other 1(0) and make them pairs (2 squares).
  4. Check for quads (4 squares) and octets (8 squares) of adjacent 1s (0s) even if they contain some 1s(0s) which have already been combined. They must geometrically form a square or a rectangle.
  5. Check for any 1s(0s) that have not been combined yet and combine them into bigger squares if possible.
  6. Form the minimal expression by summing (multiplying) the product the product (sum) terms of all the groups.

**Ex**:Reduce the expression f=∑m(0,2,3,4,5,6) using mapping and implement it in AOI logic as well as in NAND logic.The Sop k-map and its reduction , and the implementation of the minimal expression using AOI logic and the corresponding NAND logic are shown in figures below

In SOP k-map, the reduction is done as:

1. m5 has only one adjacency m4 , so combine m5 and m4 into a square. Along this 2-square A remains constant as 1 and B remains constant as 0 but C varies from 0 to 1. So read it as A .
2. m3 has only one adjacency m2 , so combine m3 and m2 into a square. Along this 2-square A remains constant as 0 and B remains constant as 1 but C varies from 1 to 0. So read it as B.
3. m6 can form a 2-square with m2 and m4 can form a 2-square with m0, but observe that by wrapping the map from left to right m0, m4 ,m2 ,m6 can form a 4-square. Out of these m2 andm4 have already been combined but they can be utilized again. So make it. Along this 4-square, A is changing from 0 to 1 and B is also changing from 0 to 1 but C is remaining constant as 0. so read it as .
4. Write all the product terms in SOP form. So the minimal SOP expression is

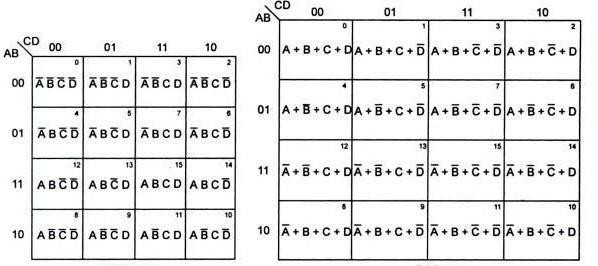
fmin= 

### k-map AOI logic NAND logic

**Four variable k-maps:**

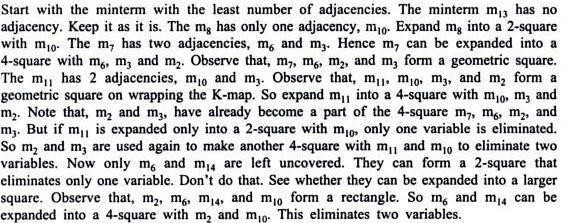
Four variable k-map expressions can have 24=16 possible combinations of input variables such as A.B.C.D with minterm designations m0,m1 m15 respectively

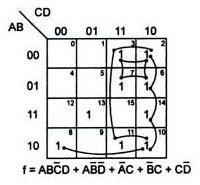
in SOP form & with maxterms M0,M1, ----------M15 respectively in POS form. It has 24=16 squares or cells.The binary number designations of rows & columns are in the gray code. Here follows 01 & 10 follows 11 called Adjacency ordering.



SOP form POS form

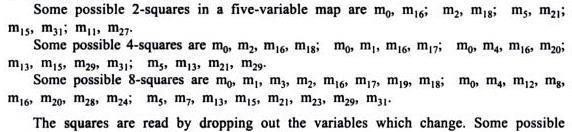
EX: 



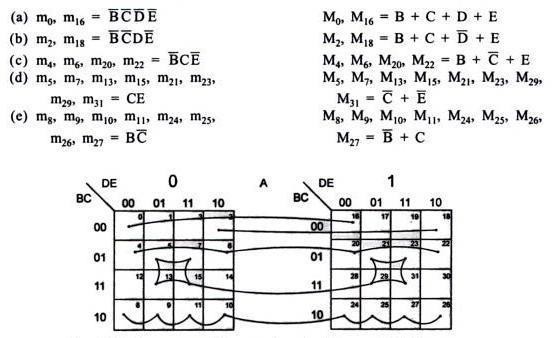


### Five variable k-map:

Five variable k-map can have 25 =32 possible combinations of input variable asA,B,C,D,E with minterms m0, m1-----m31 respectively in SOP with maxterms M0,M1, -----------M31 respectively in POS form. It has 25=32 squares or cells of the k-map are divided into 2 blocks of16 squares each.The left block represents minterms from m0 to m15 in which A is a 0, and the right block represents minterms from m16 to m31 in which A is 1.The 5-variable k-map may contain 2-squares, 4-squares , 8-squares , 16-squares or 32-squares involving these two blocks. Squares are also considered adjacent in these two blocks, if when superimposing one block on top of another, the squares coincide with one another.

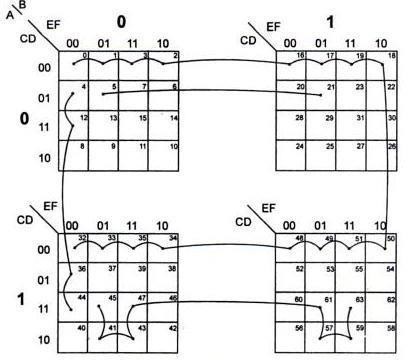


Grouping s is



### Six variable k-map:

Six variable k-map can have 26 =64 combinations as ABCDEF with minterms m0, m1-----m63 respectively in SOP & with maxterms M0,M1, -----------M63 respectively in POS form. It has 26=64 squares or cells of the k-map are divided into 4 blocks of 16 squares each.



Some possible groupings in a six variable k-map

**Don’t care combinations:** For certain input combinations, the value of the output is unspecified either because the input combinations are invalid or because the precise value of the output is of no consequence. The combinations for which the value of experiments are not specified are called don‘t care combinations are invalid or because the precise value of the output is of no consequence. The combinations for which the value of expressions is not specified are called don‘t care combinations or Optional Combinations, such expressions stand incompletely specified. The output is a don‘t care for these invalid combinations.

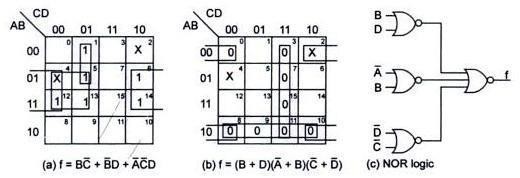
Ex:In XS-3 code system, the binary states 0000, 0001, 0010,1101,1110,1111 are unspecified. & never occur called don‘t cares.

A standard SOP expression with don‘t cares can be converted into a standard POS form by keeping the don‘t cares as they are & writing the missing minterms of the SOP form as the maxterms of the POS form viceversa.

Don‘t cares denoted by ‗X‘ or ‗φ‘

Ex:f=∑m(1,5,6,12,13,14)+d(2,4)

Or f=π M(0,3,7,9,10,11,15).πd(2,4)

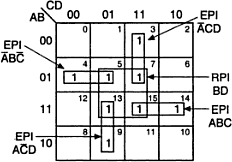


### Prime implicants, Essential Prime implicants, Redundant prime implicants:

Each square or rectangle made up of the bunch of adjacent minterms is called a subcube. Each of these subcubes is called a Prime implicant (PI). The PI which contains at leastone which cannot be covered by any other prime implicants is called as Essential Prime implicant (EPI).The PI whose each 1 is covered at least by one EPI is called a Redundant Prime implicant (RPI). A PI which is neither an EPI nor a RPI is called a Selective Prime implicant (SPI).

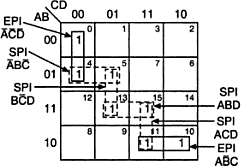
The function has unique MSP comprising EPI is F(A,B,C,D)= CD+ABC+A D + B

The RPI ‗BD‘ may be included without changing the function but the resulting expression would not be in minimal SOP(MSP) form.



Essential and Redundant Prime Implicants

F(A,B,C,D)=∑m(0,4,5,10,11,13,15) SPI are marked by dotted squares, shows MSP form of a function need not be unique.



Essential and Selective Prime Implicants

Here, the MSP form is obtained by including two EPI‘s & selecting a set of SPI‘s to cover remaining uncovered minterms 5,13,15. & these can be covered as

(A) (4,5) &(13,15) B +ABD

(B) (5,13) & (13,15) B D+ABD

(C) (5,13) & (15,11) B D+ACD

F(A,B,C,D)= +A C EPI‘s + B +ABD

|  |  |  |  |
| --- | --- | --- | --- |
| (OR) | F(A,B,C,D)= | +A C EPI‘s + | B D+ABD |
| (OR) | F(A,B,C,D)= | +A C EPI‘s + | B D+ACD |

### Mapping when the function is not expressed in minterms (maxterms):

An expression in k-map must be available as a sum (product) of minterms (maxterms). However if not so expressed, it is not necessary to expand the expression algebraically into its minterms (maxterms). Instead, expansion into minterms (maxterms) can be accomplished in the process of entering the terms of the expression on the k-map.

### Limitations of Karnaugh maps:

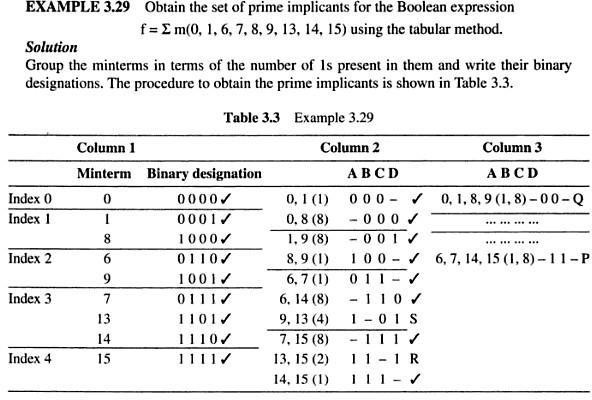
* + Convenient as long as the number of variables does not exceed six.
  + Manual technique, simplification process is heavily dependent on the human abilities.

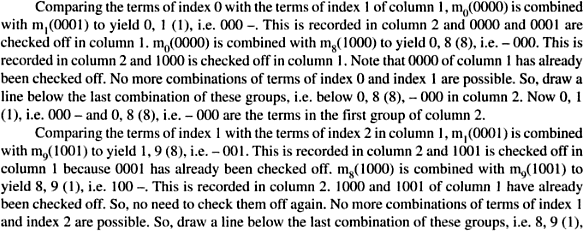
### Quine-Mccluskey Method:

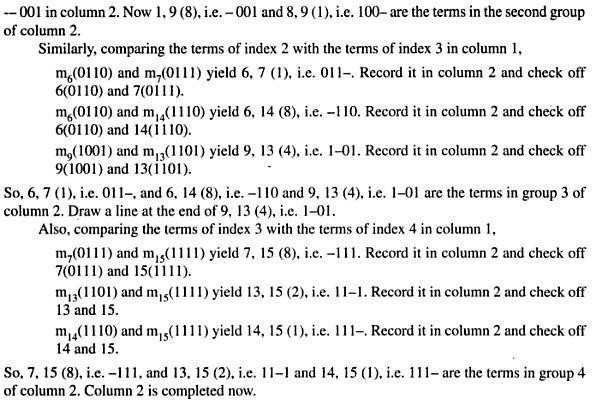
It also known as *Tabular method.* It is more systematic method of minimizing expressions of even larger number of variables. It is suitable for hand computation as well as computation by machines i.e., programmable. . The procedure is based on repeated application of the combining theorem.

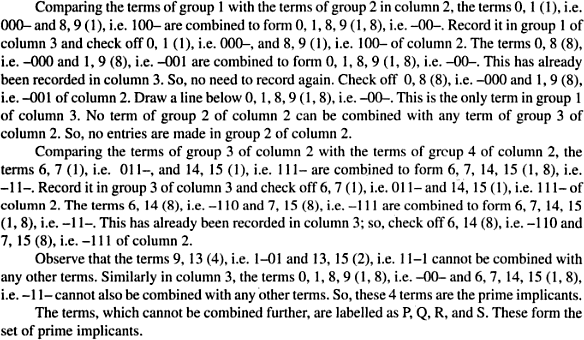
Procedure:

* + Decimal Representation
  + Don‘t cares
  + PI chart
  + EPI
  + Dominating Rows & Columns
  + Determination of Minimal expressions in complescases. Branching Method:

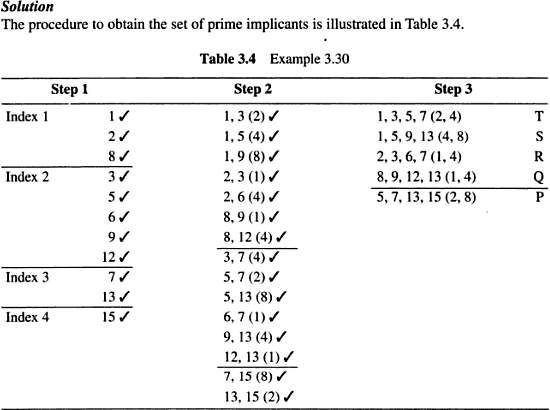


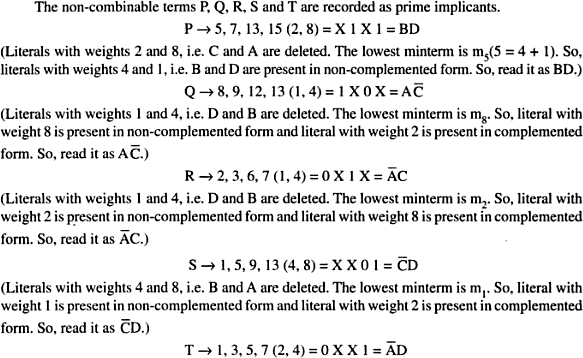


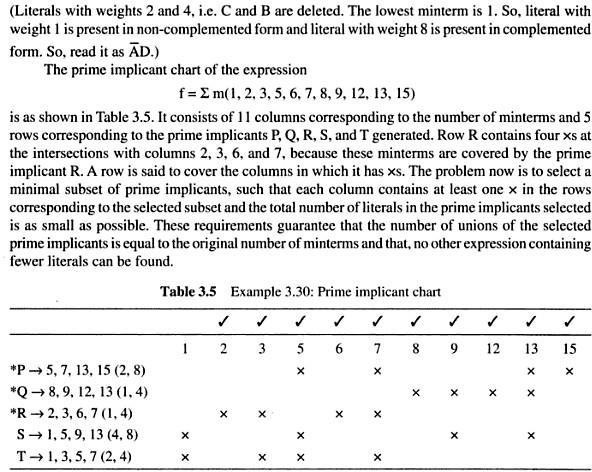


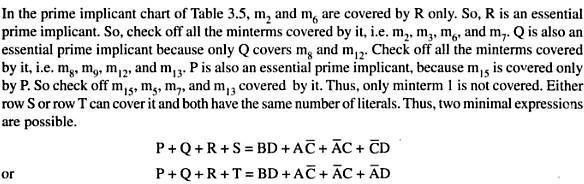


EX:





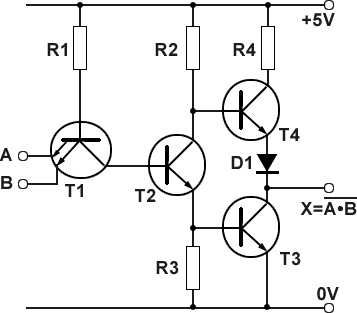




**TTL (Transistor Transistor Logic):**

TTL gates use a 5V (±0.25V) supply, and are capable of high-speed operation. Over 600 different logic ICs are available, covering a very wide range of digital functions. Due to the use of bipolar transistors, TTL has much higher power consumption than similar CMOS types, when working at relatively low frequencies. As the frequency of signals handled increases however, this difference decreases as the power consumption of CMOS increases and TTL power consumption remains nearly constant.

Notice that this circuit looks similar to those found in analogue push pull amplifiers, except that the transistors here are driven either into cut-off or saturation, rather than working in their linear operating condition. Also, being constructed within an IC, it can use a device not normally found in conventional analogue amplifiers, a multi emitter transistor.



Schematic diagram of a TTL NAND Gate

Fig. shows a typical schematic for a TTL NAND gate. R1 is a low value resistor (about 4K) and as the base current of T1 is small, the base voltage is about +5V. If both emitters of T1 are at logic 1, (also around +5V), there will be very little potential difference between base and emitter, and T1 will be turned off. As T1 is not conducting, its collector will also be at about 5V, and due to this high potential, T2 base will have a higher potential than its emitter, which will cause T2 to conduct heavily and go into saturation.

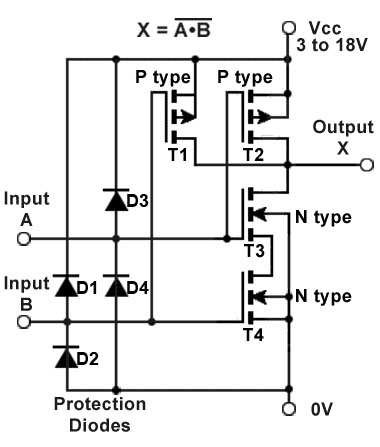
T2 collector will therefore fall to a low potential, and the emitter voltage of T2 will rise due to the current flow through R3. The voltage across R3 will rise to a sufficient level (about 0.7V) to fully turn on T3. As T3 saturates, its collector voltage will fall to about 0.2V, thus giving a logic 0 state at the output terminal.

T4 emitter voltage is made up of T3 VCE (about 0.2V) plus the forward voltage drop across D1, which will be about 0.7V

The base potential of T4 is made up of T3 base/emitter potential VBE (about 0.7V), plus the collector/emitter, potential (VCE) of T2, (about 0.2V), giving a base voltage for T4 of about 0.9V. Therefore the base and emitter voltages on T4 are approximately equal, so T4 will be turned off.

With BOTH input terminals at logic 1 therefore, the output terminal will be at logic 0, the correct operation for a NAND gate.

If either one of the inputs is taken to logic 0 however, this will make T1 conduct, as the emitter that is at logic 0 will be at a lower voltage than that supplied to the base by R1. This will cause T1 to saturate, taking its collector to a low potential (less than 0.8V) and as this is also connected to T2 base T2 will turn off, making its collector voltage and T4 base voltage, rise to very nearly +Vcc.

As virtually no current (ICE) is flowing through T2 collector/emitter circuit, practically no voltage is developed across the emitter resistor R3, reducing T3 base voltage to 0V, and so T3 is turned off. However, sufficient current will be flowing out of the output terminal (feeding the next gate input circuit) to cause T4 emitter to be held at about 4.1V. This is 0.9V below +Vcc, made up of the voltage across D1 (0.7V) plus the saturation voltage VCE of T4 (0.2V). This places about 4V or logic 1 (between 2.4V and 5V) on the output terminal.

## **CMOS:**

CMOS ICs can operate from a wide range of supply voltages (typically 3 to 18V, and lower with some sub families), with very low power consumption. The name CMOS (COMPLIMENTARY Metal Oxide Semiconductor) is used because opposite types, both P type and N type MOSFETs are used in the construction of these gates. Fig 3.2.3 shows a theoretical schematic circuit for a NAND gate.

Operation

T1 and T2 are P type MOSFETs and either of these transistors will be turned on when logic 0 is applied to its gate. T3 and T4 are N type MOSFETs and either of these transistors will be turned on by applying a logic 1 to its gate.

T1 and T2 are connected in parallel from supply to the output

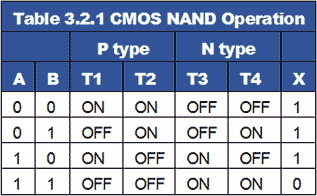
CMOS NAND Gate

X, so switching either of them on will result in a logic 1 at output X.

T3 and T4 are connected in series between X and ground so when both are switched on, a logic 0 will appear at output X. The eventual logic state at X depends of course on the on or off state of the combination of all four transistors, and these are controlled by the logic states applied to the inputs A and B as can be seen in Table 3.2.1.

Input A controls T2 and T3 so that when logic 0 is applied, T2 is on and T3 is off. Logic 1 on input A reverses this condition.

Input B controls T1 and T4 so that logic 0 applied to B turns T1 on and T4 off. Logic 1 on input B reverses the condition.



**Outcomes:**

i. Identifying different representation in Boolean expression.

ii. Analyze different methods used for simplification of Boolean expressions.

iii. Implement Logic Gate Circuits for Boolean expression.

## 

**UNIT II**

**COMBINATIONAL CIRCUITS**

# Pre requisition:

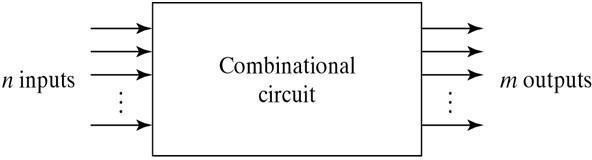
# i. Basic Knowledge in Boolean algebra.

# ii. Basic Knowledge in logic gates.

# iii. Basic Knowledge in minimization techniques.

**Combinational Logic**

* Logic circuits for digital systems may be combinational or sequential.
* A combinational circuit consists of input variables, logic gates, and output variables.



For n input variables,there are 2n possible combinations of binary input variables .For each possible input Combination ,there is one and only one possible output combination.A combinational circuit can be described by m Boolean functions one for each output variables.Usually the input s comes from flip-flops and outputs goto flip-flops.

# Design Procedure:

1. The problem is stated
2. The number of available input variables and required output variables is determined.

3.The input and output variables are assigned letter symbols.

4.The truth table that defines the required relationship between inputs and outputs is derived. 5.The simplified Boolean function for each output is obtained.

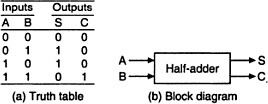
# Adders:

Digital computers perform variety of information processing tasks,the one is arithmetic operations.And the most basic arithmetic operation is the addition of two binary digits.i.e, 4 basic possible operations are:

0+0=0,0+1=1,1+0=1,1+1=10

The first three operations produce a sum whose length is one digit, but when augends and addend bits are equal to 1,the binary sum consists of two digits.The higher significant bit of this result is called a carry.A combinational circuit that performs the addition of two bits is called a half- adder. One that performs the addition of 3 bits (two significant bits & previous carry) is called a full adder.& 2 half adder can employ as a full-adder.

**The Half Adder**: A Half Adder is a combinational circuit with two binary inputs (augends and addend bits and two binary outputs (sum and carry bits.) It adds the two inputs (A and B) and produces the sum (S) and the carry (C) bits. It is an arithmetic operation of addition of two single bit words.

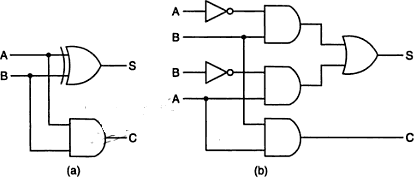


The Sum(S) bit and the carry (C) bit, according to the rules of binary addition, the sum (S) is the X-OR of A and B ( It represents the LSB of the sum). Therefore,

S=A𝐵+𝐴

The carry (C) is the AND of A and B (it is 0 unless both the inputs are 1).Therefore, C=AB

A half-adder can be realized by using one X-OR gate and one AND gate a

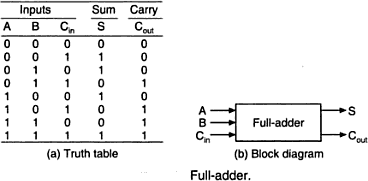


Logic diagrams of half-adder

# The Full Adder:

A Full-adder is a combinational circuit that adds two bits and a carry and outputs a sum bit and a carry bit. To add two binary numbers, each having two or more bits, the LSBs can be added by using a half-adder. The carry resulted from the addition of the LSBs is carried over to the next significant column and added to the two bits in that column. So, in the second and higher columns, the two data bits of that column and the carry bit generated from the addition in the previous column need to be added.

The full-adder adds the bits A and B and the carry from the previous column called the carry-in Cin and outputs the sum bit S and the carry bit called the carry-out Cout . The variable S gives the value of the least significant bit of the sum. The variable Cout gives the output carry. The eight rows under the input variables designate all possible combinations of 1s and 0s that these variables may have. The 1s and 0s for the output variables are determined from the arithmetic sum of the input bits. When all the bits are 0s , the output is 0. The S output is equal to 1 when only 1 input is equal to 1 or when all the inputs are equal to 1. The Cout has a carry of 1 if two or three inputs are equal to 1.



From the truth table, a circuit that will produce the correct sum and carry bits in response to every possible combination of A,B and Cin is described by

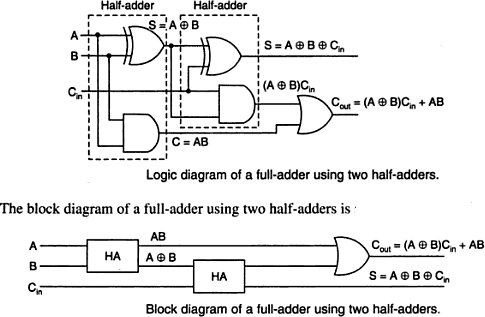
*S*  *ABCin*  *ABCin*  *ABCin*  *ABCin Cout*  *ABCin*  *ABCin*  *ABCin* *~~A~~BCin*

and

*S*  *A*  *B*  *Cin*

*Cout*  *ACin*  *BCin*  *AB*

The sum term of the full-adder is the X-OR of A,B, and Cin, i.e, the sum bit the modulo sum of the data bits in that column and the carry from the previous column. The logic diagram of the full-adder using two X-OR gates and two AND gates (i.e, Two half adders) and one OR gate is



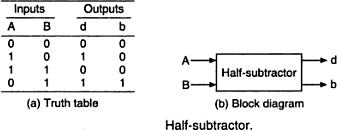
Even though a full-adder can be constructed using two half-adders, the disadvantage is that the bits must propagate through several gates in accession, which makes the total propagation delay greater than that of the full-adder circuit using AOI logic.

# Subtractors:

The subtraction of two binary numbers may be accomplished by taking the complement of the subtrahend and adding it to the minuend. By this, the subtraction operation becomes an addition operation and instead of having a separate circuit for subtraction, the adder itself can be used to perform subtraction. This results in reduction of hardware. In subtraction, each subtrahend bit of the number is subtracted from its corresponding significant minuend bit to form a difference bit. If the minuend bit is smaller than the subtrahend bit, a 1 is borrowed from the next significant position., that has been borrowed must be conveyed to the next higher pair of bits by means of a signal coming out (output) of a given stage and going into (input) the next higher stage.

# The Half-Subtractor:

A Half-subtractor is a combinational circuit that subtracts one bit from the other and produces the difference. It also has an output to specify if a 1 has been borrowed. . It is used to subtract the LSB of the subtrahend from the LSB of the minuend when one binary number is subtracted from the other.

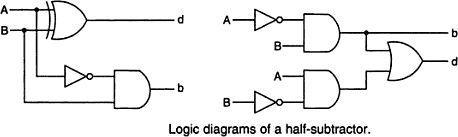
A Half-subtractor is a combinational circuit with two inputs A and B and two outputs d and b. d indicates the difference and b is the output signal generated that informs the next stage that a 1 has been borrowed. When a bit B is subtracted from another bit A, a difference bit (d) and a borrow bit (b) result according to the rules given as

The output borrow b is a 0 as long as A≥B. It is a 1 for A=0 and B=1. The d output is the result of the arithmetic operation2b+A-B.

A circuit that produces the correct difference and borrow bits in response to every possible combination of the two 1-bit numbers is , therefore ,

d=A𝐵+𝐴 and b=𝐴 B

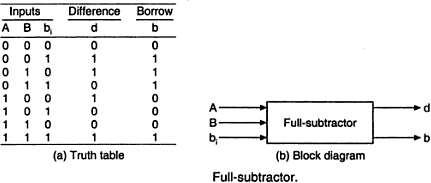
That is, the difference bit is obtained by X-OR ing the two inputs, and the borrow bit is obtained by ANDing the complement of the minuend with the subtrahend.Note that logic for this exactly the same as the logic for output S in the half-adder.



A half-substractor can also be realized using universal logic either using only NAND gates or using NOR gates as:

# The Full-Subtractor:

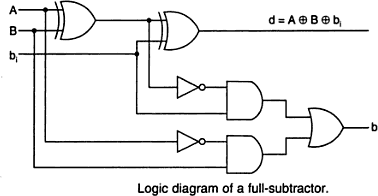
The half-subtractor can be only for LSB subtraction. IF there is a borrow during the subtraction of the LSBs, it affects the subtraction in the next higher column; the subtrahend bit is subtracted from the minuend bit, considering the borrow from that column used for the subtraction in the preceding column. Such a subtraction is performed by a full-subtractor. It subtracts one bit (B) from another bit (A) , when already there is a borrow bi from this column for the subtraction in the preceding column, and outputs the difference bit (d) and the borrow bit(b) required from the next d and b. The two outputs present the difference and output borrow. The 1s and 0s for the output variables are determined from the subtraction of A-B-bi.



From the truth table, a circuit that will produce the correct difference and borrow bits in response to every possible combinations of A,B and bi is

A full-subtractor can be realized using X-OR gates and AOI gates as

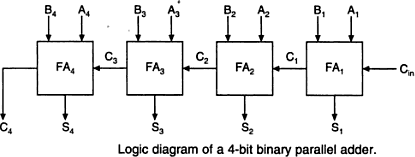
33



# Binary Parallel Adder:

A binary parallel adder is a digital circuit that adds two binary numbers in parallel form and produces the arithmetic sum of those numbers in parallel form. It consists of full adders connected in a chain , with the output carry from each full-adder connected to the input carry of the next full-adder in the chain.

The inter connection of four full-adder (FA) circuits to provide a 4-bit parallel adder. The augends bits of A and addend bits of B are designated by subscript numbers from right to left, with subscript 1 denoting the lower –order bit. The carries are connected in a chain through the full-adders. The input carry to the adder is Cin and the output carry is C4. The S output generates the required sum bits. When the 4-bit full-adder circuit is enclosed within an IC package, it has four terminals for the augends bits, four terminals for the addend bits, four terminals for the sum bits, and two terminals for the input and output carries. AN n-bit parallel adder requires n-full adders. It can be constructed from 4-bit, 2-bit and 1-bit full adder ICs by cascading several packages. The output carry from one package must be connected to the input carry of the one with the next higher –order bits. The 4-bit full adder is a typical example of an MSI function.



# Ripple carry adder:

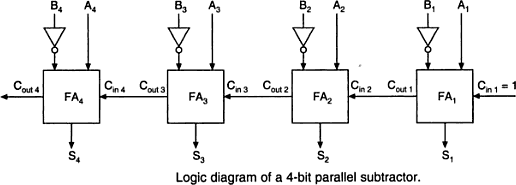
In the parallel adder, the carry –out of each stage is connected to the carry-in of the next stage. The sum and carry-out bits of any stage cannot be produced, until sometime after the carry-in of that stage occurs. This is due to the propagation delays in the logic circuitry,which lead to a time delay in the addition process. The carry propagation delay for each full- adder is the time between the application of the carry-in and the occurrence of the carry-out.

The 4-bit parallel adder, the sum (S1) and carry-out (C1) bits given by FA1 are not valid, until after the propagation delay of FA1. Similarly, the sum S2 and carry-out (C2) bits given by FA2 are not valid until after the cumulative propagation delay of two full adders (FA1 and FA2) , and so on. At each stage ,the sum bit is not valid until after the carry bits in all the preceding stages are valid. Carry bits must propagate or ripple through all stages before the most significant sum bit is valid. Thus, the total sum (the parallel output) is not valid until after the cumulative delay of all the adders.

The parallel adder in which the carry-out of each full-adder is the carry-in to the next most significant adder is called a ripple carry adder.. The greater the number of bits that a ripple carry adder must add, the greater the time required for it to perform a valid addition. If two numbers are added such that no carries occur between stages, then the add time is simply the propagation time through a single full-adder.

# Bit Parallel Subtractor:

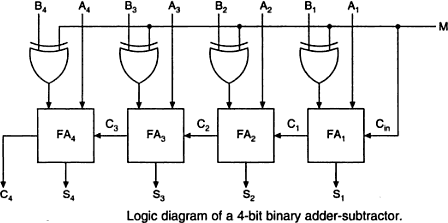
The subtraction of binary numbers can be carried out most conveniently by means of complements , the subtraction A-B can be done by taking the 2‘s complement of B and adding it to A . The 2‘s complement can be obtained by taking the 1‘s complement and adding 1 tothe least significant pair of bits. The 1‘s complement can be implemented with inverters as



# Binary-Adder Subtractor:

A 4-bit adder-subtractor, the addition and subtraction operations are combined into one circuit with one common binary adder. This is done by including an X-OR gate with each full-adder. 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 X-OR gate receives input M and one of the inputs of B. When M=0,The full-adder receives the value of B , the input carry is 0 and the circuit performs A+B. when and C1=1. The B inputs are complemented and a 1 is through the input carry. The circuit performs the operation A plus the 2‘s complement of B.

35

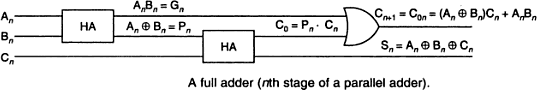


# The Look-Ahead –Carry Adder:

In parallel-adder,the speed with which an addition can be performed is governed by the time required for the carries to propagate or ripple through all of the stages of the adder. The look-ahead carry adder speeds up the process by eliminating this ripple carry delay. It examines all the input bits simultaneously and also generates the carry-in bits for all the stages simultaneously.

The method of speeding up the addition process is based on the two additional functions of the full-adder, called the carry generate and carry propagate functions.

Consider one full adder stage; say the nth stage of a parallel adder as shown in fig. we know that is made by two half adders and that the half adder contains an X-OR gate to produce the sum and an AND gate to produce the carry. If both the bits An and Bn are 1s, a carry has to be generated in this stage regardless of whether the input carry Cin is a 0 or a 1. This is called generated carry, expressed as Gn= An.Bn which has to appear at the output through the OR gate as shown in fig.



There is another possibility of producing a carry out. X-OR gate inside the half-adder

at the input produces an intermediary sum bit- call it Pn –which is expressed as . Next Pn and Cn are added using the X-OR gate inside the second half adder to produce the final

sum bit and and output carryC0= Pn.Cn=(  )Cn which becomes carry for the (n+1) thstage.

Consider the case of both Pn and Cn being 1. The input carry Cn has to be propagated to the output only if Pn is 1. If Pn is 0, even if Cn is 1, the and gate in the second half-adder will inhibit Cn . the carry out of the nth stage is 1 when either Gn=1 or Pn.Cn =1 or both Gn and Pn.Cn are equal to 1.

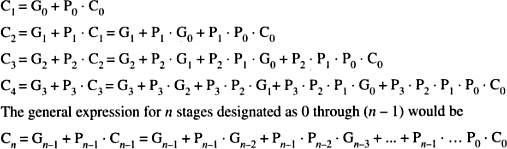
For the final sum and carry outputs of the nth stage, we get the following Boolean

expressions.

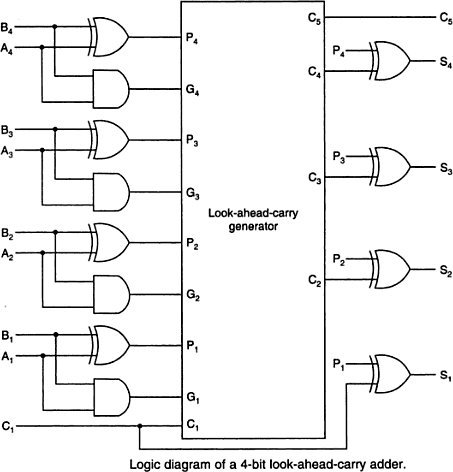


Observe the recursive nature of the expression for the output carry at the nth stage which becomes the input carry for the (n+1)st stage .it is possible to express the output carry of a higher significant stage is the carry-out of the previous stage.

Based on these , the expression for the carry-outs of various full adders are as follows,



Observe that the final output carry is expressed as a function of the input variables in SOP form. Which is two level AND-OR or equivalent NAND-NAND form. Observe that the full look-ahead scheme requires the use of OR gate with (n+1) inputs and AND gates with number of inputs varying from 2 to (n+1).



# Serial Adder:

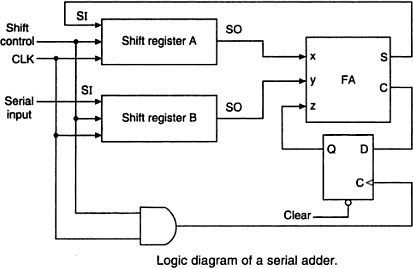
A serial adder is used to add binary numbers in serial form. The two binary numbers to be added serially are stored in two shift registers A and B. Bits are added one pair at a time through a single full adder (FA) circuit as shown. The carry out of the full-adder is transferred to a D flip- flop. The output of this flip-flop is then used as the carry input for the next pair of significant bits. The sum bit from the S output of the full-adder could be transferred to a third shift register. By shifting the sum into A while the bits of A are shifted out, it is possible to use one register for storing both augend and the sum bits. The serial input register B can be used to transfer a new binary number while the addend bits are shifted out during the addition.

The operation of the serial adder is:

Initially register A holds the augend, register B holds the addend and the carry flip-flop is cleared to 0. The outputs (SO) of A and B provide a pair of significant bits for the full-adder at x and y. The shift control enables both registers and carry flip-flop , so, at the clock pulse both registers are shifted once to the right, the sum bit from S enters the left most flip-flop of A , and the output carry is transferred into flip-flop Q . The shift control enables the registers for a number of clock pulses equal to the number of bits of the registers. For each succeeding clock pulse a new sum bit is transferred to A, a new carry is transferred to Q, and both registers are shifted once to the right. This process continues until the shift control is disabled. Thus the addition is accomplished by passing each pair of bits together with the previous carry through a single full adder circuit and transferring the sum, one bit at a time, into register A.

39

Initially, register A and the carry flip-flop are cleared to 0 and then the first number is added from B. While B is shifted through the full adder, a second number is transferred to it through its serial input. The second number is then added to the content of register A while a third number is transferred serially into register B. This can be repeated to form the addition of two, three, or more numbers and accumulate their sum in register A.



# Difference between Serial and Parallel Adders:

The parallel adder registers with parallel load, whereas the serial adder uses shift registers. The number of full adder circuits in the parallel adder is equal to the number of bits in the binary numbers, whereas the serial adder requires only one full adder circuit and a carry flip- flop. Excluding the registers, the parallel adder is a combinational circuit, whereas the serial adder is a sequential circuit. The sequential circuit in the serial adder consists of a full-adder and a flip-flop that stores the output carry.

# BCD Adder:

The BCD addition process:

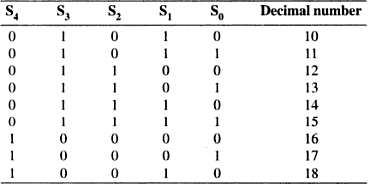
* 1. Add the 4-bit BCD code groups for each decimal digit position using ordinary binary addition.
  2. For those positions where the sum is 9 or less, the sum is in proper BCD form and no correction isneeded.
  3. When the sum of two digits is greater than 9, a correction of 0110 should be added to that sum, to produce the proper BCD result. This will produce a carry to be added to the next decimal position.

A BCD adder circuit must be able to operate in accordance with the above steps. In other words, the circuit must be able to do the following:

1. Add two 4-bit BCD code groups, using straight binary addition.
2. Determine, if the sum of this addition is greater than 1101 (decimal 9); if it is , add 0110 (decimal 6) to this sum and generate a carry to the next decimal position.

The first requirement is easily met by using a 4- bit binary parallel adder such as the 74LS83 IC .For example , if the two BCD code groups A3A2A1A0and B3B2B1B0 are applied to a 4-bit parallel adder, the adder will output S4S3S2S1S0 , where S4 is actually C4 , the carry –out of the MSB bits.

The sum outputs S4S3S2S1S0 can range anywhere from 00000 to 100109when both the BCD code groups are 1001=9). The circuitry for a BCD adder must include the logic needed to detect whenever the sum is greater than 01001, so that the correction can be added in. Those cases , where the sum is greater than 1001 are listed as:



Let us define a logic output X that will go HIGH only when the sum is greater than 01001 (i.e, for the cases in table). If examine these cases ,see that X will be HIGH for either of the following conditions:

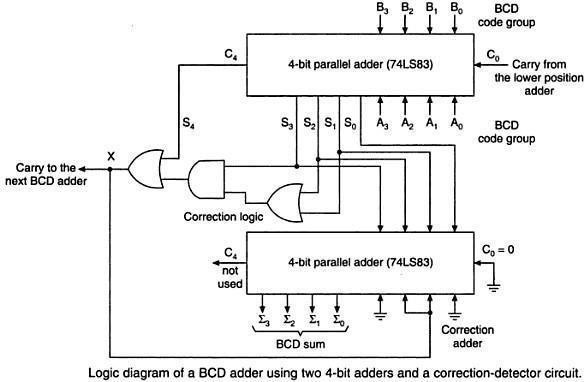
1. Whenever S4 =1(sum greater than15)
2. Whenever S3 =1 and either S2 or S1 or both are 1 (sum 10 to 15) This condition can be expressed as

X=S4+S3(S2+S1)

Whenever X=1, it is necessary to add the correction factor 0110 to the sum bits, and to generate a carry. The circuit consists of three basic parts. The two BCD code groups A3A2A1A0 and B3B2B1B0 are added together in the upper 4-bit adder, to produce the sum S4S3S2S1S0. The logic gates shown implement the expression for X. The lower 4-bit adder will add the correction 0110 to the sum bits, only when X=1, producing the final BCD sum output represented by ∑3∑2∑1∑0. The X is also the carry-out that is produced when the sum is greater than 01001. When X=0, there is no carry and no addition of 0110. In such cases, ∑3∑2∑1∑0= S3S2S1S0.

Two or more BCD adders can be connected in cascade when two or more digit decimal numbers are to be added. The carry-out of the first BCD adder is connected as the carry-in of the second BCD adder, the carry-out of the second BCD adder is connected as the carry-in of the third BCD adder and so on.

.



# Code converters:

The availability of a large variety of codes for the same discrete elements of information results in the use of different codes by different digital systems. It is sometimes necessary to use the output of one system as the input to another. A conversion circuit must be

45

inserted between the two systems if each uses different codes for the same information. Thus a

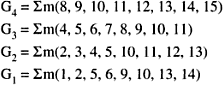
code converter is a logic circuit whose inputs are bit patterns representing numbers (or character) in one cod and whose outputs are the corresponding representation in a different code. Code converters are usually multiple output circuits.

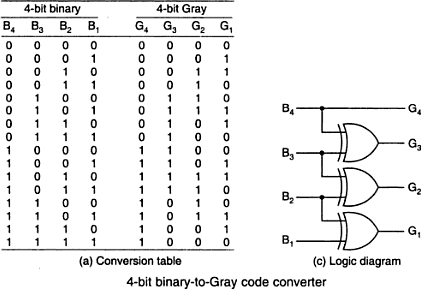
To convert from binary code A to binary code B, the input lines must supply the bit combination of elements as specified by code A and the output lines must generate the corresponding bit combination of code B. A combinational circuit performs this transformation by means of logic gates. For example, a binary –to-gray code converter has four binary input lines B4, B3,B2,B1 and four gray code output lines G4,G3,G2,G1. When the input is 0010, for instance, the output should be 0011 and so forth. To design a code converter, we use a code table treating it as a truth table to express each output as a Boolean algebraic function of all the inputs.

In this example, of binary –to-gray code conversion, we can treat the binary to the gray code table as four truth tables to derive expressions for G4, G3, G2, and G1. Each of these four expressions would, in general, contain all the four input variables B4, B3,B2,and B1. Thus,this code converter is actually equivalent to four logic circuits, one for each of the truth tables.

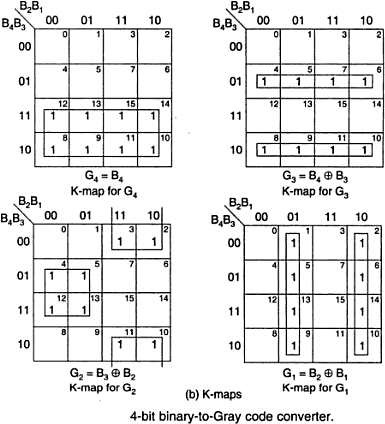
The logic expression derived for the code converter can be simplified using the usual techniques, including ‗don‘t cares‘ if present. Even if the input is an unweighted code, the same cell numbering method which we used earlier can be used, but the cell numbers --must correspond to the input combinations as if they were an 8-4-2-1 weighted code. s

# Design of a 4-bit binary to gray code converter:

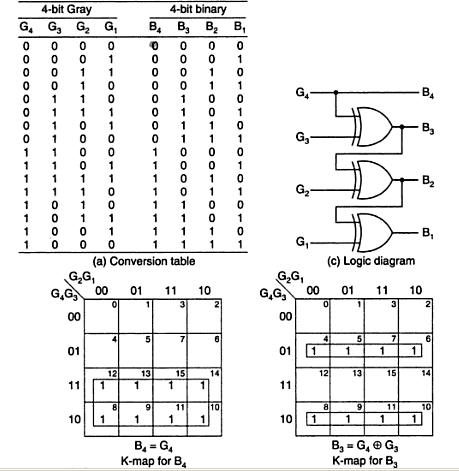
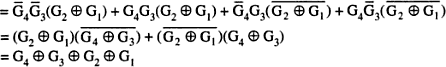
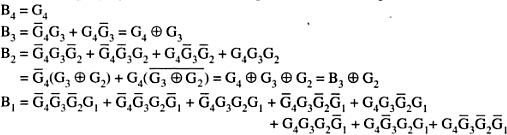
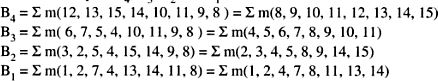




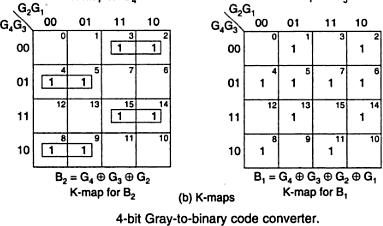
46



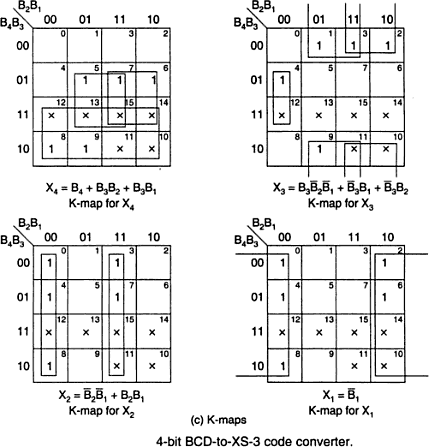
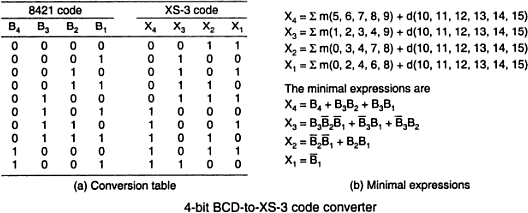
## Design of a 4-bit gray to Binary code converter:



47

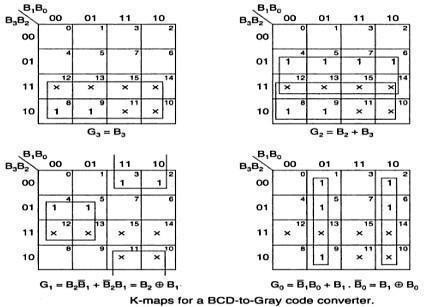
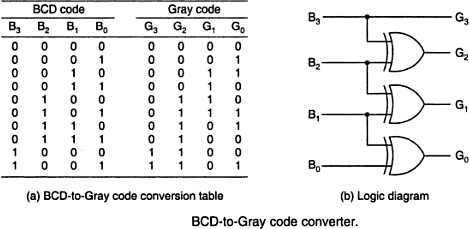


### Design of a 4-bit BCD to XS-3 code converter:



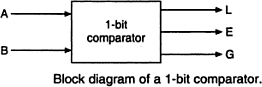
48

**Design of a BCD to gray code converter:**

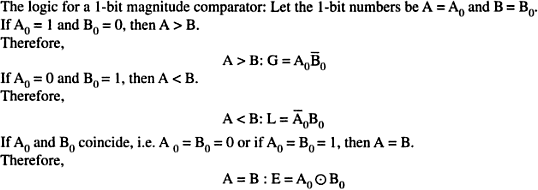


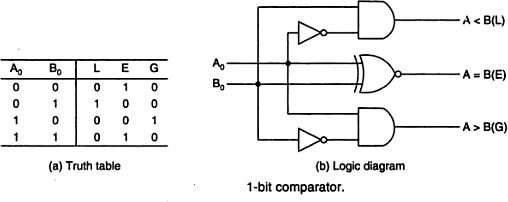
**Comparators:**

49

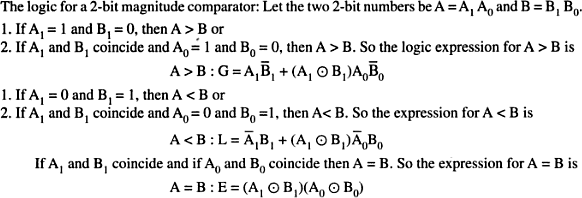


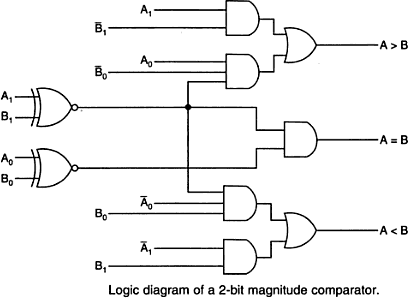
50





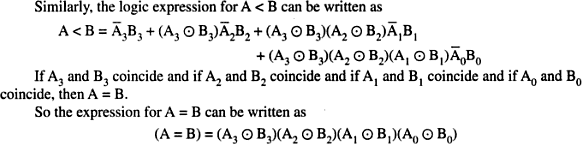
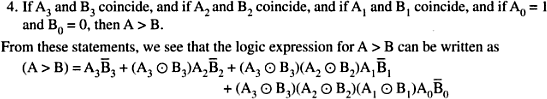
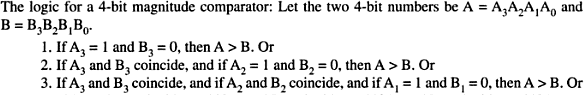
* 1. **Magnitude Comparator:**

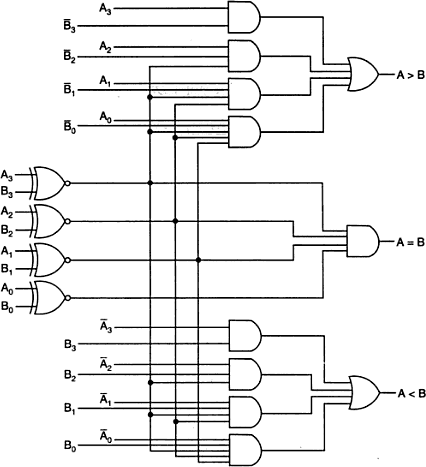
**1- bit Magnitude Comparator:**



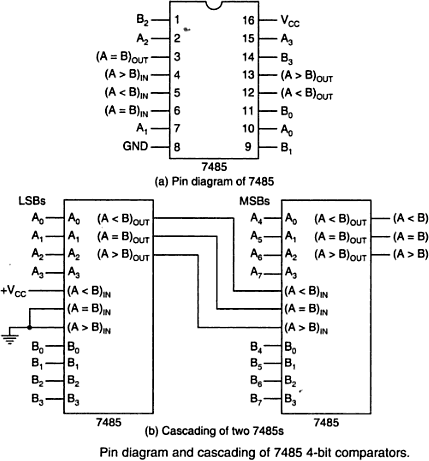
51

1. **Bit MagnitudeComparator**:





**IC Comparator:**



MULTIPLEXERS

**Multiplexer** is a combinational circuit that has maximum of 2n data inputs, ‘n’ selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines. Since there are ‘n’ selection lines, there will be 2n possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as **Mux**.

## **4x1 Multiplexer**

4x1 Multiplexer has four data inputs I3, I2, I1 & I0, two selection lines s1 & s0 and one output Y. The **block diagram** of 4x1 Multiplexer is shown in the following figure.

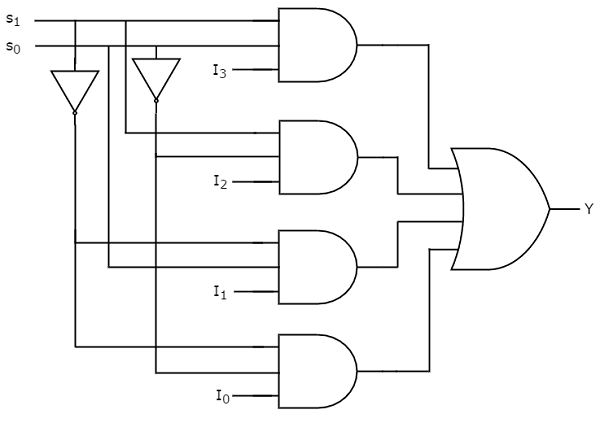
One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

|  |  |  |
| --- | --- | --- |
| **Selection Lines** | | **Output** |
| **S1** | **S0** | **Y** |
| 0 | 0 | I0 |
| 0 | 1 | I1 |
| 1 | 0 | I2 |
| 1 | 1 | I3 |

From Truth table, we can directly write the **Boolean function** for output, Y as

Y=S1′S0′I0+S1′S0I1+S1S0′I2+S1S0I3

We can implement this Boolean function using Inverters, AND gates & OR gate. The **circuit diagram** of 4x1 multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 8x1 Multiplexer and 16x1 multiplexer by following the same procedure.

## **Implementation of Higher-order Multiplexers.**

Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

* 8x1 Multiplexer
* 16x1 Multiplexer

### 8x1 Multiplexer

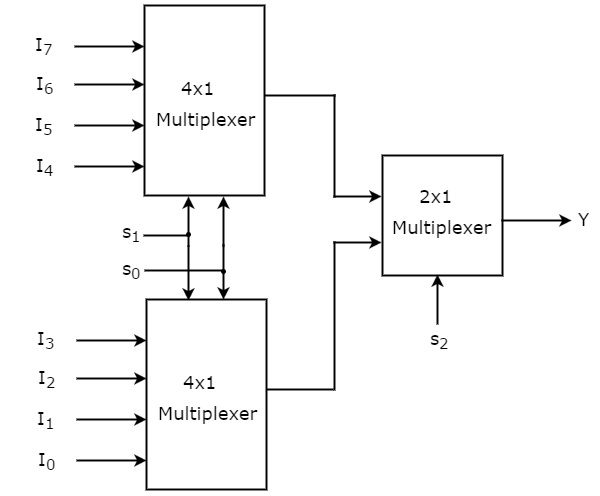
In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

So, we require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 8x1 Multiplexer has eight data inputs I7 to I0, three selection lines s2, s1 & s0 and one output Y. The **Truth table** of 8x1 Multiplexer is shown below.

|  |  |  |  |
| --- | --- | --- | --- |
| **Selection Inputs** | | | **Output** |
| **S2** | **S1** | **S0** | **Y** |
| 0 | 0 | 0 | I0 |
| 0 | 0 | 1 | I1 |
| 0 | 1 | 0 | I2 |
| 0 | 1 | 1 | I3 |
| 1 | 0 | 0 | I4 |
| 1 | 0 | 1 | I5 |
| 1 | 1 | 0 | I6 |
| 1 | 1 | 1 | I7 |

We can implement 8x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 8x1 Multiplexer is shown in the following figure.



The same **selection lines, s1 & s0** are applied to both 4x1 Multiplexers. The data inputs of upper 4x1 Multiplexer are I7 to I4 and the data inputs of lower 4x1 Multiplexer are I3 to I0. Therefore, each 4x1 Multiplexer produces an output based on the values of selection lines, s1 & s0.

The outputs of first stage 4x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s2** is applied to 2x1 Multiplexer.

* If s2 is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs I3 to I0 based on the values of selection lines s1 & s0.
* If s2 is one, then the output of 2x1 Multiplexer will be one of the 4 inputs I7 to I4 based on the values of selection lines s1 & s0.

Therefore, the overall combination of two 4x1 Multiplexers and one 2x1 Multiplexer performs as one 8x1 Multiplexer.

### 16x1 Multiplexer

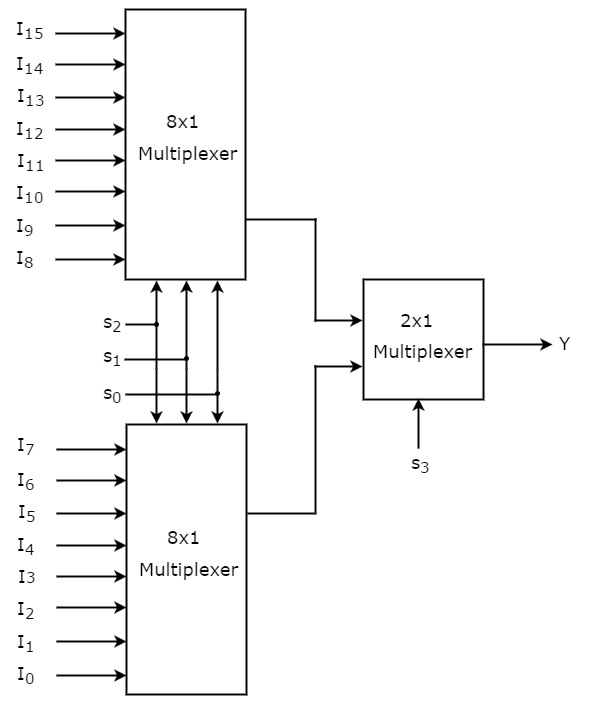
In this section, let us implement 16x1 Multiplexer using 8x1 Multiplexers and 2x1 Multiplexer. We know that 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output. Whereas, 16x1 Multiplexer has 16 data inputs, 4 selection lines and one output.

So, we require two **8x1 Multiplexers** in first stage in order to get the 16 data inputs. Since, each 8x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 16x1 Multiplexer has sixteen data inputs I15 to I0, four selection lines s3 to s0 and one output Y. The **Truth table** of 16x1 Multiplexer is shown below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Selection Inputs** | | | | **Output** |
| **S3** | **S2** | **S1** | **S0** | **Y** |
| 0 | 0 | 0 | 0 | I0 |
| 0 | 0 | 0 | 1 | I1 |
| 0 | 0 | 1 | 0 | I2 |
| 0 | 0 | 1 | 1 | I3 |
| 0 | 1 | 0 | 0 | I4 |
| 0 | 1 | 0 | 1 | I5 |
| 0 | 1 | 1 | 0 | I6 |
| 0 | 1 | 1 | 1 | I7 |
| 1 | 0 | 0 | 0 | I8 |
| 1 | 0 | 0 | 1 | I9 |
| 1 | 0 | 1 | 0 | I10 |
| 1 | 0 | 1 | 1 | I11 |
| 1 | 1 | 0 | 0 | I12 |
| 1 | 1 | 0 | 1 | I13 |
| 1 | 1 | 1 | 0 | I14 |
| 1 | 1 | 1 | 1 | I15 |

We can implement 16x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 16x1 Multiplexer is shown in the following figure.



The **same selection lines, s2, s1 & s0** are applied to both 8x1 Multiplexers. The data inputs of upper 8x1 Multiplexer are I15 to I8 and the data inputs of lower 8x1 Multiplexer are I7 to I0. Therefore, each 8x1 Multiplexer produces an output based on the values of selection lines, s2, s1 & s0.

The outputs of first stage 8x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s3** is applied to 2x1 Multiplexer.

* If s3 is zero, then the output of 2x1 Multiplexer will be one of the 8 inputs Is7 to I0 based on the values of selection lines s2, s1 & s0.
* If s3 is one, then the output of 2x1 Multiplexer will be one of the 8 inputs I15 to I8 based on the values of selection lines s2, s1 & s0.

Therefore, the overall combination of two 8x1 Multiplexers and one 2x1 Multiplexer performs as one 16x1 Multiplexer.

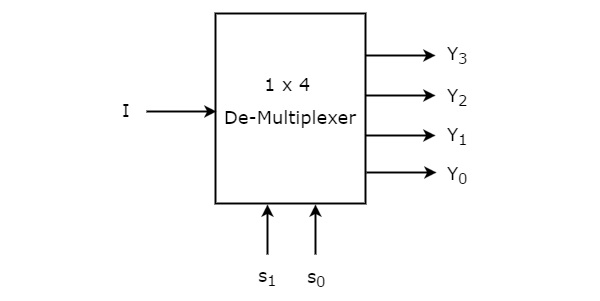
**Demultiplexers**

**De-Multiplexer** is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, ‘n’ selection lines and maximum of 2n outputs. The input will be connected to one of these outputs based on the values of selection lines.

Since there are ‘n’ selection lines, there will be 2n possible combinations of zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as **De-Mux**.

## **1x4 De-Multiplexer**

1x4 De-Multiplexer has one input I, two selection lines, s1 & s0 and four outputs Y3, Y2, Y1 &Y0. The **block diagram** of 1x4 De-Multiplexer is shown in the following figure.



The single input ‘I’ will be connected to one of the four outputs, Y3 to Y0 based on the values of selection lines s1 & s0. The **Truth table** of 1x4 De-Multiplexer is shown below

.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Selection Inputs** | | **Outputs** | | | |
| **S1** | **S0** | **Y3** | **Y2** | **Y1** | **Y0** |
| 0 | 0 | 0 | 0 | 0 | **I** |
| 0 | 1 | 0 | 0 | **I** | 0 |
| 1 | 0 | 0 | **I** | 0 | 0 |
| 1 | 1 | **I** | 0 | 0 | 0 |

From the above Truth table, we can directly write the **Boolean functions** for each output as

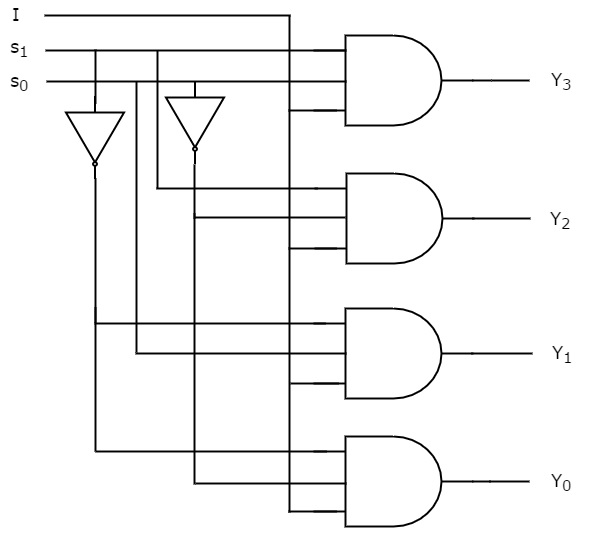
Y3=s1s0IY3=s1s0I

Y2=s1s0′IY2=s1s0′I

Y1=s1′s0IY1=s1′s0I

Y0=s1′s0′IY0=s1′s0′I

We can implement these Boolean functions using Inverters & 3-input AND gates. The **circuit diagram** of 1x4 De-Multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 1x8 De-Multiplexer and 1x16 De-Multiplexer by following the same procedure.

## **Implementation of Higher-order De-Multiplexers**

Now, let us implement the following two higher-order De-Multiplexers using lower-order De-Multiplexers.

* 1x8 De-Multiplexer
* 1x16 De-Multiplexer

### 1x8 De-Multiplexer

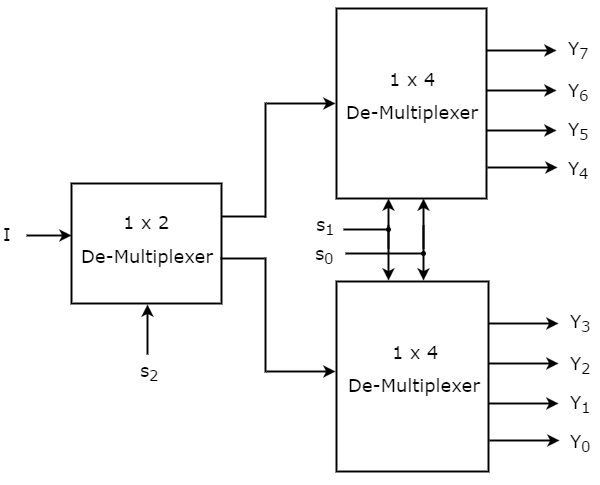
In this section, let us implement 1x8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x4 De-Multiplexer has single input, two selection lines and four outputs. Whereas, 1x8 De-Multiplexer has single input, three selection lines and eight outputs.

So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x8 De-Multiplexer.

Let the 1x8 De-Multiplexer has one input I, three selection lines s2, s1 & s0 and outputs Y7 to Y0. The **Truth table** of 1x8 De-Multiplexer is shown below.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Selection Inputs** | | | **Outputs** | | | | | | | |
| **s2** | **s1** | **s0** | **Y7** | **Y6** | **Y5** | **Y4** | **Y3** | **Y2** | **Y1** | **Y0** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | **I** |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | **I** | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | **I** | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 | **I** | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | **I** | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 0 | 0 | **I** | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | **I** | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | **I** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

We can implement 1x8 De-Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 1x8 De-Multiplexer is shown in the following figure.



The common **selection lines, s1 & s0** are applied to both 1x4 De-Multiplexers. The outputs of upper 1x4 De-Multiplexer are Y7 to Y4 and the outputs of lower 1x4 De-Multiplexer are Y3 to Y0.

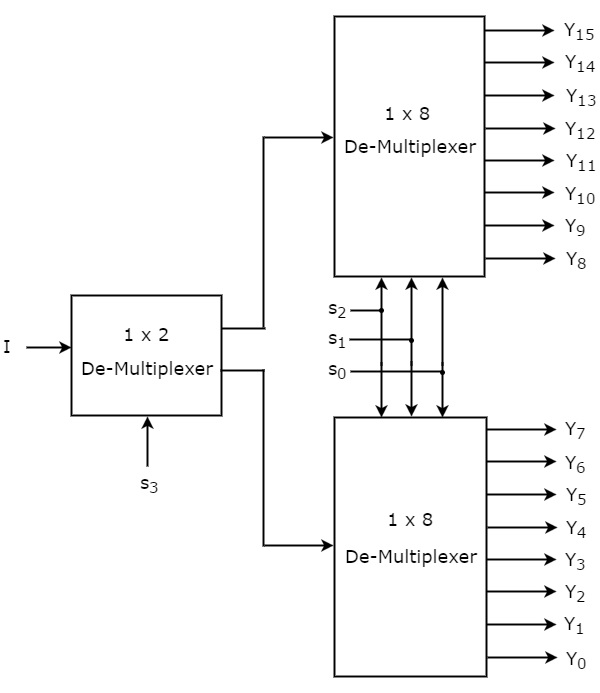
The other **selection line, s2** is applied to 1x2 De-Multiplexer. If s2 is zero, then one of the four outputs of lower 1x4 De-Multiplexer will be equal to input, I based on the values of selection lines s1 & s0. Similarly, if s2 is one, then one of the four outputs of upper 1x4 DeMultiplexer will be equal to input, I based on the values of selection lines s1 & s0.

### 1x16 De-Multiplexer

In this section, let us implement 1x16 De-Multiplexer using 1x8 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x8 De-Multiplexer has single input, three selection lines and eight outputs. Whereas, 1x16 De-Multiplexer has single input, four selection lines and sixteen outputs.

So, we require two **1x8 De-Multiplexers** in second stage in order to get the final sixteen outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x16 De-Multiplexer.

Let the 1x16 De-Multiplexer has one input I, four selection lines s3, s2, s1 & s0 and outputs Y15 to Y0. The **block diagram** of 1x16 De-Multiplexer using lower order Multiplexers is shown in the following figure.



The common **selection lines s2, s1 & s0** are applied to both 1x8 De-Multiplexers. The outputs of upper 1x8 De-Multiplexer are Y15 to Y8 and the outputs of lower 1x8 DeMultiplexer are Y7 to Y0.

The other **selection line, s3** is applied to 1x2 De-Multiplexer. If s3 is zero, then one of the eight outputs of lower 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s2, s1 & s0. Similarly, if s3 is one, then one of the 8 outputs of upper 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s2, s1 & s0.

DECODERS AND ENCODERS

The previous section began by discussing an application: Given 2n data signals, the problem is to select, under the control of n select inputs, sequences of these 2n data signals to send out serially on a communications link. The reverse operation on the receiving end of the communications link is to receive data serially on a single line and to convey it to one of 2n output lines. This again is controlled by a set of control inputs. It is this application that needs only one input line; other applications may require more than one.We will now investigate such a generalized circuit.

Conceivably, there might be a combinational circuit that accepts n inputs (not necessarily 1, but a small number) and causes data to be routed to one of many, say up to 2n, outputs. Such circuits have the generic name decoder.

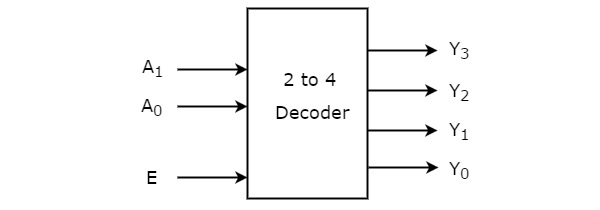
Semantically, at least, if something is to be decoded, it must have previously been encoded, the reverse operation from decoding. Like a multiplexer, an encoding circuit must accept data from a large number of input lines and convert it to data on a smaller number of output lines (not necessarily just one). This section will discuss a number of implementations of decoders and encoders.

**Decoder:**

Decoder is a combinational circuit that has ‘n’ input lines and maximum of 2n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min terms** of ‘n’ input variables lineslines, when it is enabled.

## **2 to 4 Decoder**

Let 2 to 4 Decoder has two inputs A1 & A0 and four outputs Y3, Y2, Y1 & Y0. The **block diagram** of 2 to 4 decoder is shown in the following figure.



One of these four outputs will be ‘1’ for each combination of inputs when enable, E is ‘1’. The **Truth table** of 2 to 4 decoder is shown below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Enable** | **Inputs** | | **Outputs** | | | |
| **E** | **A1** | **A0** | **Y3** | **Y2** | **Y1** | **Y0** |
| 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 |

From Truth table, we can write the Boolean functions for each output as

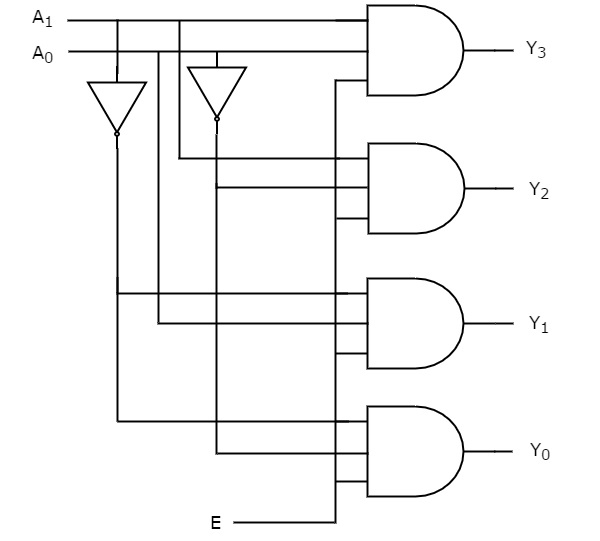
Y3=E.A1.A0Y3=E.A1.A0

Y2=E.A1.A0′Y2=E.A1.A0′

Y1=E.A1′.A0Y1=E.A1′.A0

Y0=E.A1′.A0′Y0=E.A1′.A0′

Each output is having one product term. So, there are four product terms in total. We can implement these four product terms by using four AND gates having three inputs each & two inverters. The circuit diagram of 2 to 4 decoder is shown in the following figure.



Therefore, the outputs of 2 to 4 decoder are nothing but the min terms of two input variables A1 & A0, when enable, E is equal to one. If enable, E is zero, then all the outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces eight min terms of three input variables A2, A1 & A0 and 4 to 16 decoder produces sixteen min terms of four input variables A3, A2, A1 & A0.

## **Implementation of Higher-order Decoders**

Now, let us implement the following two higher-order decoders using lower-order decoders.

* 3 to 8 decoder
* 4 to 16 decoder

### 3 to 8 Decoder

In this section, let us implement 3 to 8 decoder using 2 to 4 decoders. We know that 2 to 4 Decoder has two inputs, A1 & A0 and four outputs, Y3 to Y0. Whereas, 3 to 8 Decoder has three inputs A2, A1 & A0 and eight outputs, Y7 to Y0.

We can find the number of lower order decoders required for implementing higher order decoder using the following formula.

Requirednumberoflowerorderdecoders=m2m1Requirednumberoflowerorderdecoders=m2m1

Where,

m1:m1 is the number of outputs of lower order decoder.

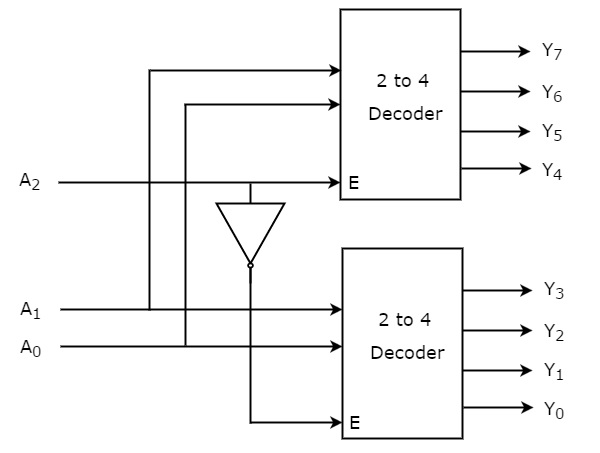
m2:m2 is the number of outputs of higher order decoder.

Here,  m1 = 4 and m2 = 8. Substitute,

these two values in the above formula.

Required number of 2 to 4 decoders=8/4=2

Therefore, we require two 2 to 4 decoders for implementing one 3 to 8 decoder. The block diagram of 3 to 8 decoder using 2 to 4 decoders is shown in the following figure.



The parallel inputs A1 & A0 are applied to each 2 to 4 decoder. The complement of input A2 is connected to Enable, E of lower 2 to 4 decoder in order to get the outputs, Y3 to Y0. These are the lower four min terms. The input, A2 is directly connected to Enable, E of upper 2 to 4 decoder in order to get the outputs, Y7 to Y4. These are the higher four min terms.

### 4 to 16 Decoder

In this section, let us implement 4 to 16 decoder using 3 to 8 decoders. We know that 3 to 8 Decoder has three inputs A2, A1 & A0 and eight outputs, Y7 to Y0. Whereas, 4 to 16 Decoder has four inputs A3, A2, A1 & A0 and sixteen outputs, Y15 to Y0

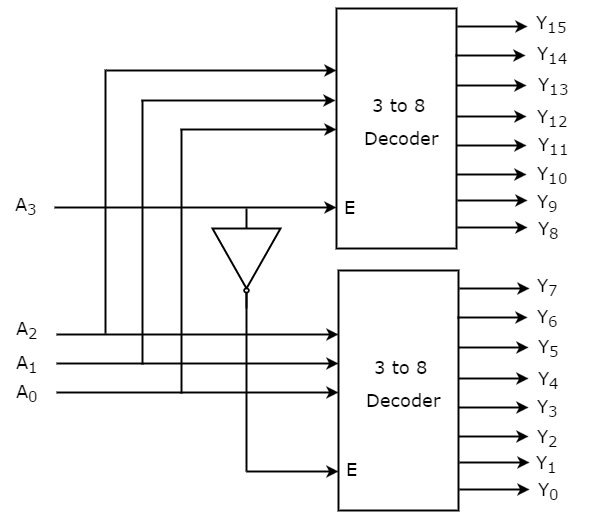
We know the following formula for finding the number of lower order decoders required.

Required number of lower order decoders=m2/m1

Substitute,  m1 = 8 and m2 = 16 in the above formula.

Required number of 3to8 decoders=16/8=2

Therefore, we require two 3 to 8 decoders for implementing one 4 to 16 decoder. The block diagram of 4 to 16 decoder using 3 to 8 decoders is shown in the following figure.



The parallel inputs A2, A1 & A0 are applied to each 3 to 8 decoder. The complement of input, A3 is connected to Enable, E of lower 3 to 8 decoder in order to get the outputs, Y7 to Y0. These are the lower eight min terms. The input, A3 is directly connected to Enable, E of upper 3 to 8 decoder in order to get the outputs, Y15 to Y8. These are the higher eight min terms.

ENCODER

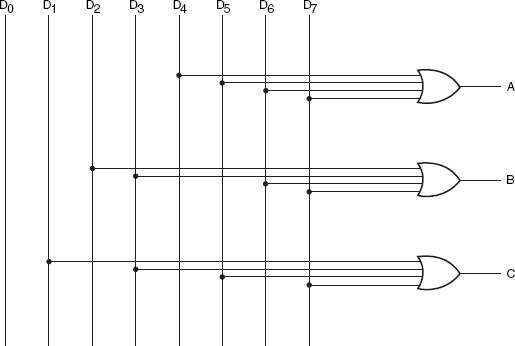
An encoder is a combinational circuit that performs the inverse operation of a decoder. If a device output code has fewer bits than the input code has, the device is usually called an encoder. e.g. 2n-to-n, priority encoders.

The simplest encoder is a 2n-to-n binary encoder, where it has only one of 2n inputs = 1 and the output is the n-bit binary number corresponding to the active input.

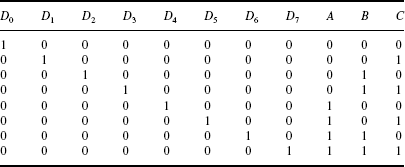
Priority Encoder

A priority encoder is a practical form of an encoder. The encoders available in IC form are all priority encoders. In this type of encoder, a priority is assigned to each input so that, when more than one input is simultaneously active, the input with the highest priority is encoded. We will illustrate the concept of priority encoding with the help of an example. Let us assume that the octal to-binary encoder described in the previous paragraph has an input priority for higher-order digits. Let us also assume that input lines D2, D4 and D7 are all simultaneously in logic ‗1‘ state. In that case, only D7 will be encoded and the output will be

111. The truth table of such a priority

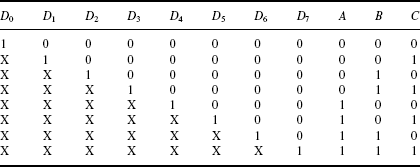


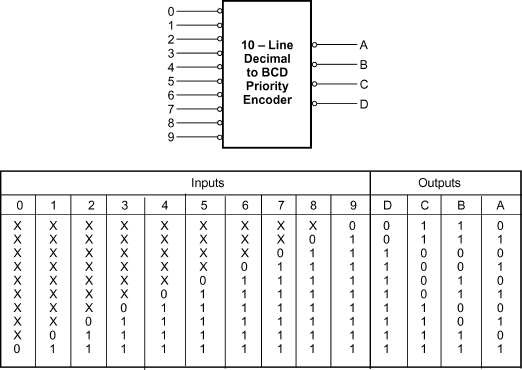
Octal to binary encoder



Truth table of encoder

encoder will then be modified to what is shown above in truth table. Looking at the last row of the table, it implies that, if D7 = 1, then, irrespective of the logic status of other inputs, the output is 111 as D7 will only be encoded. As another example, Fig. 8.16 shows the logic symbol and truth table of a 10-line decimal to four-line BCD encoder providing priority encoding for higher-order digits, with digit 9 having the highest priority. In the functional table shown, the input line with highest priority having a LOW on it is encoded irrespective of the logic status of the other input lines.

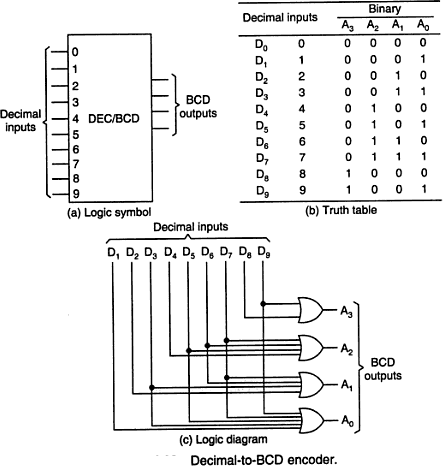




10 line decimal to four line BCD priority encoder

Some of the encoders available in IC form provide additional inputs and outputs to allow expansion. IC 74148, which is an eight-line to three -line priority encoder, is an example. ENABLE-IN (EI) and ENABLE-OUT (EO) terminals on this IC allow expansion. For instance, two 74148s can be cascaded to build a 16-line to four-line priority encoder.

Decimal to BCD Encoder:



## **Parity Bit Generator**

There are two types of parity bit generators based on the type of parity bit being generated. Even parity generator generates an even parity bit. Similarly, odd parity generator generates an odd parity bit.

### Even Parity Generator

Now, let us implement an even parity generator for a 3-bit binary input, WXY. It generates an even parity bit, P. If odd number of ones present in the input, then even parity bit, P should be ‘1’ so that the resultant word contains even number of ones. For other combinations of input, even parity bit, P should be ‘0’. The following table shows the **Truth table** of even parity generator.

|  |  |
| --- | --- |
| **Binary Input WXY** | **Even Parity bit P** |
| 000 | 0 |
| 001 | 1 |
| 010 | 1 |
| 011 | 0 |
| 100 | 1 |
| 101 | 0 |
| 110 | 0 |
| 111 | 1 |

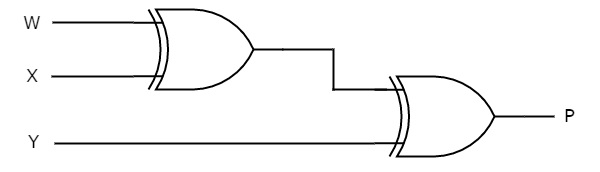
From the above Truth table, we can write the **Boolean function** for even parity bit as

P=W′X′Y+W′XY′+WX′Y′+WXYP

P=W′(X′Y+XY′)+W(X′Y′+XY

P=W′(X⊕Y)+W(X⊕Y)′=W⊕X⊕Y

The following figure shows the circuit diagram of even parity generator.

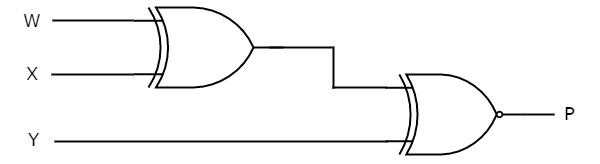


This circuit consists of two Exclusive-OR gates having two inputs each. First Exclusive OR gate having two inputs W & X and produces an output W ⊕ X. This output is given as one input of second Exclusive-OR gate. The other input of this second Exclusive-OR gate is Y and produces an output of W ⊕ X ⊕ Y.

### Odd Parity Generator

If even number of ones present in the input, then odd parity bit, P should be ‘1’ so that the resultant word contains odd number of ones. For other combinations of input, odd parity bit, P should be ‘0’.

Follow the same procedure of even parity generator for implementing odd parity generator. The circuit diagram of odd parity generator is shown in the following figure.



The above circuit diagram consists of Ex-OR gate in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity generator. In that case, the first and second levels contain an ExOR gate in each level and third level consist of an inverter.

## **Parity Checker**

There are two types of parity checkers based on the type of parity has to be checked. Even parity checker checks error in the transmitted data, which contains message bits along with even parity. Similarly, odd parity checker checks error in the transmitted data, which contains message bits along with odd parity.

### Even parity checker

Now, let us implement an even parity checker circuit. Assume a 3-bit binary input, WXY is transmitted along with an even parity bit, P. So, the resultant word data contains 4 bits, which will be received as the input of even parity checker.

It generates an even parity check bit, E. This bit will be zero, if the received data contains an even number of ones. That means, there is no error in the received data. This even parity check bit will be one, if the received data contains an odd number of ones. That means, there is an error in the received data.

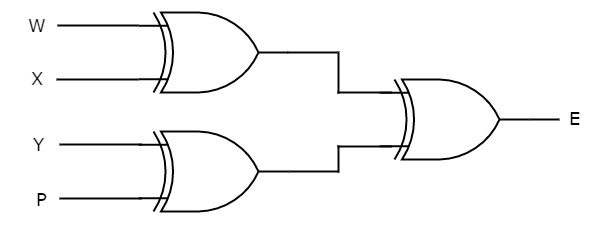
The following table shows the Truth table of an even parity checker.

|  |  |
| --- | --- |
| **4-bit Received Data WXYP** | **Even Parity Check bit E** |
| 0000 | 0 |
| 0001 | 1 |
| 0010 | 1 |
| 0011 | 0 |
| 0100 | 1 |
| 0101 | 0 |
| 0110 | 0 |
| 0111 | 1 |
| 1000 | 1 |
| 1001 | 0 |
| 1010 | 0 |
| 1011 | 1 |
| 1100 | 0 |
| 1101 | 1 |
| 1110 | 1 |
| 1111 | 0 |

From the above Truth table, we can observe that the even parity check bit value is ‘1’, when odd number of ones present in the received data. That means the Boolean function of even parity check bit is an odd function. Exclusive-OR function satisfies this condition. Hence, we can directly write the Boolean function of even parity check bit as

E=W⊕X⊕Y⊕ P

The following figure shows the circuit diagram of even parity checker.



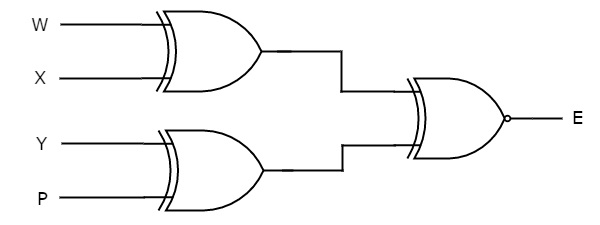
This circuit consists of three Exclusive-OR gates having two inputs each. The first level gates produce outputs of W⊕X & Y⊕P. The Exclusive-OR gate, which is in second level produces an output of W⊕X⊕Y⊕P

### Odd Parity Checker

Assume a 3-bit binary input, WXY is transmitted along with odd parity bit, P. So, the resultant word datadata contains 4 bits, which will be received as the input of odd parity checker.

It generates an **odd parity check bit, E**. This bit will be zero, if the received data contains an odd number of ones. That means, there is no error in the received data. This odd parity check bit will be one, if the received data contains even number of ones. That means, there is an error in the received data.

Follow the same procedure of an even parity checker for implementing an odd parity checker. The **circuit diagram** of odd parity checker is shown in the following figure.



The above circuit diagram consists of Ex-OR gates in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity checker. In that case, the first, second and third levels contain two Ex-OR gates, one Ex-OR gate and one inverter respectively.

**Outcomes:**

i. Able to understand the concept of combinational logic circuit.

ii. Design and Implement combinational circuit using Logic Gate Circuits.

**UNIT III**

**SEQUENTIAL CIRCUITS**

# Pre requisition:

# i. Basic Knowledge in Boolean Algebra.

# ii. Basic Knowledge in logic gates and combinational circuits.

# iii. Basic Knowledge in clock signal.

**The Basic Latch**

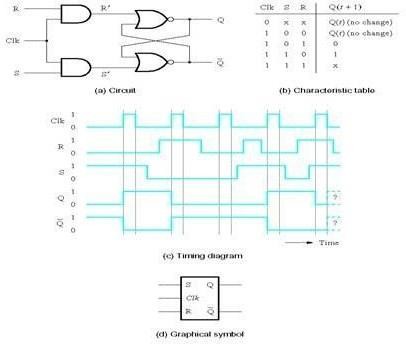
* + **Basic latch** is a feedback connection of two NOR gates or two NAND gates
  + It can store one bit of information

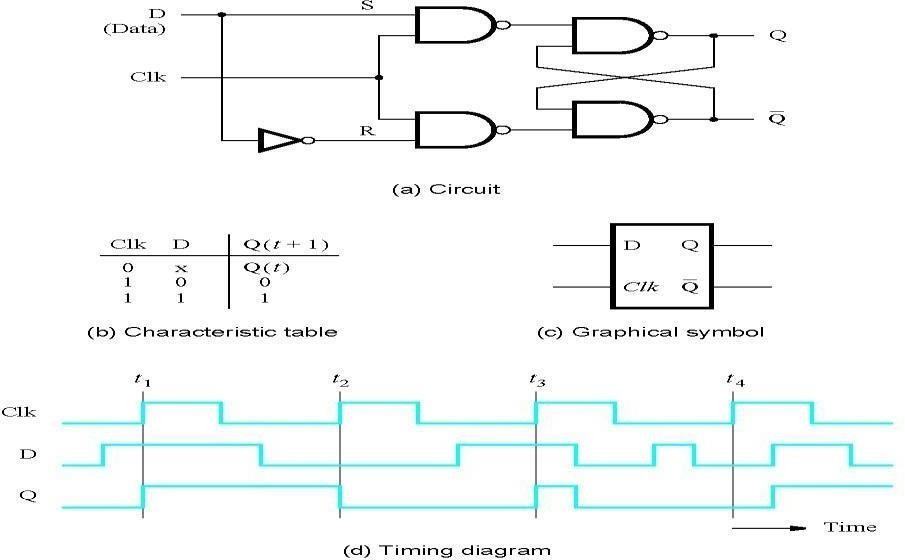
It can be set to 1 using the *S* input and reset to 0 using the *R* input

**The Gated Latch**

* + **Gated latch** is a basic latch that includes input gating and a control signal
  + The latch retains its existing state when the control input is equal to0
  + Its state may be changed when the control signal is equal to 1. In our discussion we referred to the control input as the clock
  + We consider two types of gated latches:
    - **Gated SR latch** uses the *S* and *R* inputs to set the latch to 1 or reset it to 0, respectively.
    - **Gated D latch** uses the *D* input to force the latch into a state that has the same logic value as the *D* input.

**Gated S/R Latch**



**Gated D Latch**

**Setup and Hold Times**

* + Setup Time tsu

The minimum time that the input signal must be stable prior to the edge of the clock signal.

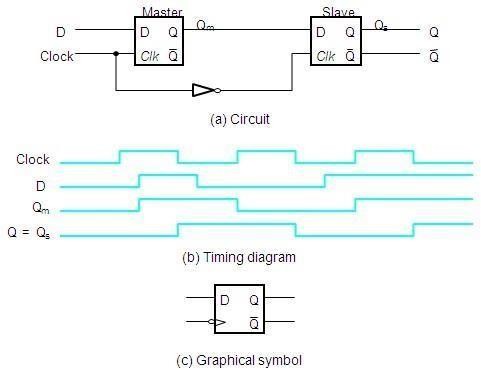
* + Hold Time th

The minimum time that the input signal must be stable after the edge of the clock signal.

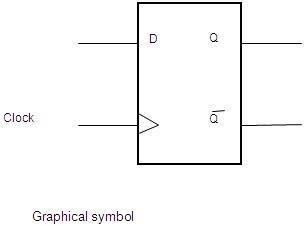
**Flip-Flops**

* A **flip-flop** is a storage element based on the gated latch principle
* It can have its output state changed only on the edge of the controlling clocksignal
* We consider two types:
* **Edge-triggered flip-flop** is affected only by the input values present when the active edge of the clock occurs
* **Master-slave flip-flop** is built with two gated latches
* The master stage is active during half of the clock cycle, and the slave stage is active during the other half.
* The output value of the flip-flop changes on the edge of the clock that activates the transfer into the slave stage.

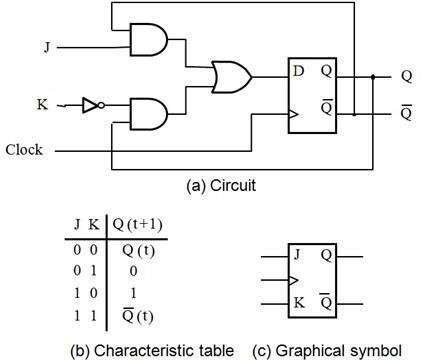
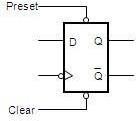
**Master-Slave D Flip-Flop**



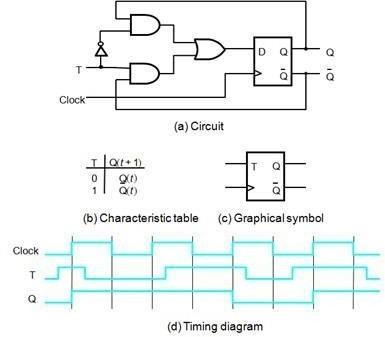
**A Positive-Edge-Triggered D Flip-Flop**



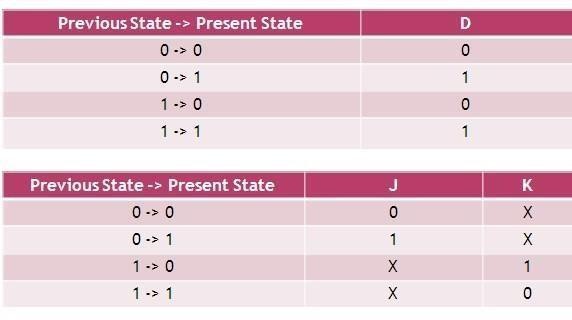
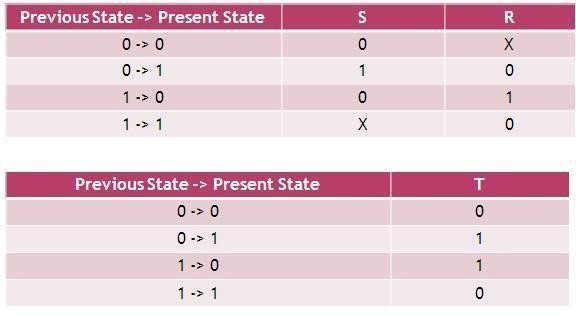
**Master-Slave D Flip-Flop with *Clear* and *Preset***



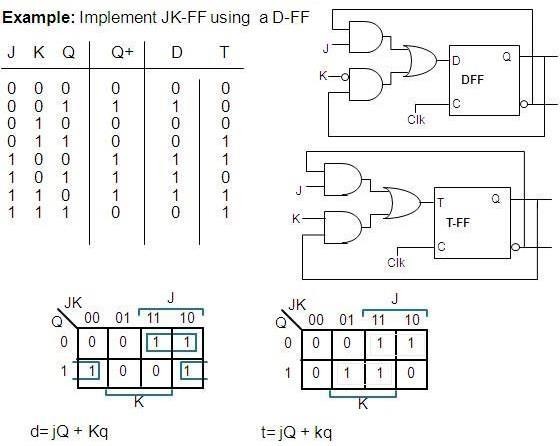
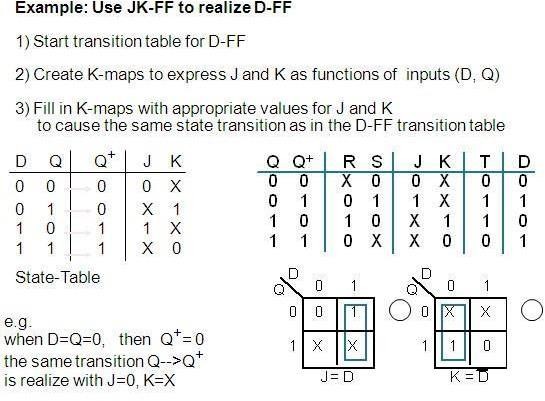
**T Flip-Flop**



**Excitation Tables**

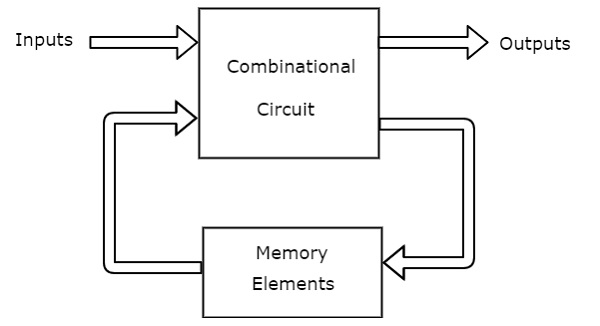


**Conversions of flip-flops**



**SEQUENTIAL CIRCUIT:**

We discussed various combinational circuits in earlier chapters. All these circuits have a set of outputss, which depends only on the combination of present inputs. The following figure shows the **block diagram** of sequential circuit.



This sequential circuit contains a set of inputs and outputs. The outputs of sequential circuit depends not only on the combination of present inputs but also on the previous outputs. Previous output is nothing but the **present state**. Therefore, sequential circuits contain combinational circuits along with memory storage elements. Some sequential circuits may not contain combinational circuits, but only memory elements.

Following table shows the **differences** between combinational circuits and sequential circuits.

|  |  |
| --- | --- |
| **Combinational Circuits** | **Sequential Circuits** |
| Outputs depend only on present inputs. | Outputs depend on both present inputs and present state. |
| Feedback path is not present. | Feedback path is present. |
| Memory elements are not required. | Memory elements are required. |
| Clock signal is not required. | Clock signal is required. |
| Easy to design. | Difficult to de |

## **Types of Sequential Circuits**

Following are the two types of sequential circuits −

* Asynchronous sequential circuits
* Synchronous sequential circuits

### Asynchronous sequential circuits

If some or all the outputs of a sequential circuit do not change affect with respect to active transition of clock signal, then that sequential circuit is called as **Asynchronous sequential circuit**. That means, all the outputs of asynchronous sequential circuits do not change affect at the same time. Therefore, most of the outputs of asynchronous sequential circuits are **not in synchronous** with either only positive edges or only negative edges of clock signal.

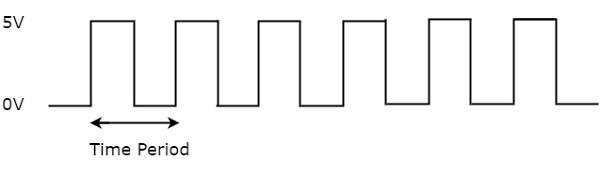
### Synchronous sequential circuits

If all the outputs of a sequential circuit change affect with respect to active transition of clock signal, then that sequential circuit is called as **Synchronous sequential circuit**. That means, all the outputs of synchronous sequential circuits change affect at the same time. Therefore, the outputs of synchronous sequential circuits are in synchronous with either only positive edges or only negative edges of clock signal.

## **Clock Signal and Triggering**

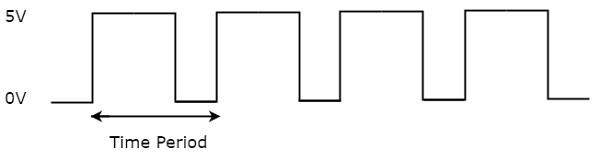
### Clock signal

Clock signal is a periodic signal and its ON time and OFF time need not be the same. We can represent the clock signal as a **square wave**, when both its ON time and OFF time are same. This clock signal is shown in the following figure.



n the above figure, square wave is considered as clock signal. This signal stays at logic High 5V5V for some time and stays at logic Low 0V0V for equal amount of time. This pattern repeats with some time period. In this case, the **time period** will be equal to either twice of ON time or twice of OFF time.

We can represent the clock signal as **train of pulses**, when ON time and OFF time are not same. This clock signal is shown in the following figure.



In the above figure, train of pulses is considered as clock signal. This signal stays at logic High 5V5V for some time and stays at logic Low 0V0V for some other time. This pattern repeats with some time period. In this case, the **time period** will be equal to sum of ON time and OFF time.

The reciprocal of the time period of clock signal is known as the **frequency** of the clock signal. All sequential circuits are operated with clock signal. So, the frequency at which the sequential circuits can be operated accordingly the clock signal frequency has to be chosen.

## **Types of Triggering**

Following are the two possible types of triggering that are used in sequential circuits.

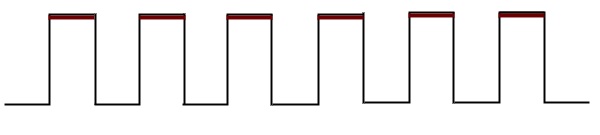
* Level triggering
* Edge triggering

### Level triggering

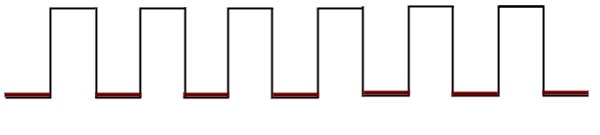
There are two levels, namely logic High and logic Low in clock signal. Following are the two **types of level triggering**.

* Positive level triggering
* Negative level triggering

If the sequential circuit is operated with the clock signal when it is in **Logic High**, then that type of triggering is known as **Positive level triggering**. It is highlighted in below figure.



If the sequential circuit is operated with the clock signal when it is in **Logic Low**, then that type of triggering is known as **Negative level triggering**. It is highlighted in the following figure.



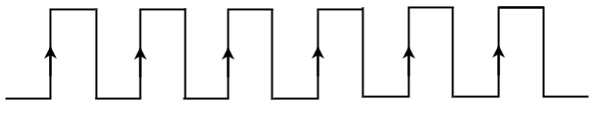
### Edge triggering

There are two types of transitions that occur in clock signal. That means, the clock signal transitions either from Logic Low to Logic High or Logic High to Logic Low.

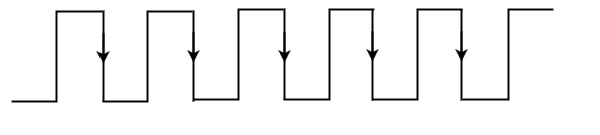
Following are the two **types of edge triggering** based on the transitions of clock signal.

* Positive edge triggering
* Negative edge triggering

If the sequential circuit is operated with the clock signal that is transitioning from Logic Low to Logic High, then that type of triggering is known as **Positive edge triggering**. It is also called as rising edge triggering. It is shown in the following figure.



If the sequential circuit is operated with the clock signal that is transitioning from Logic High to Logic Low, then that type of triggering is known as **Negative edge triggering**. It is also called as falling edge triggering. It is shown in the following figure.



In coming chapters, we will discuss about various sequential circuits based on the type of triggering that can be used in it.

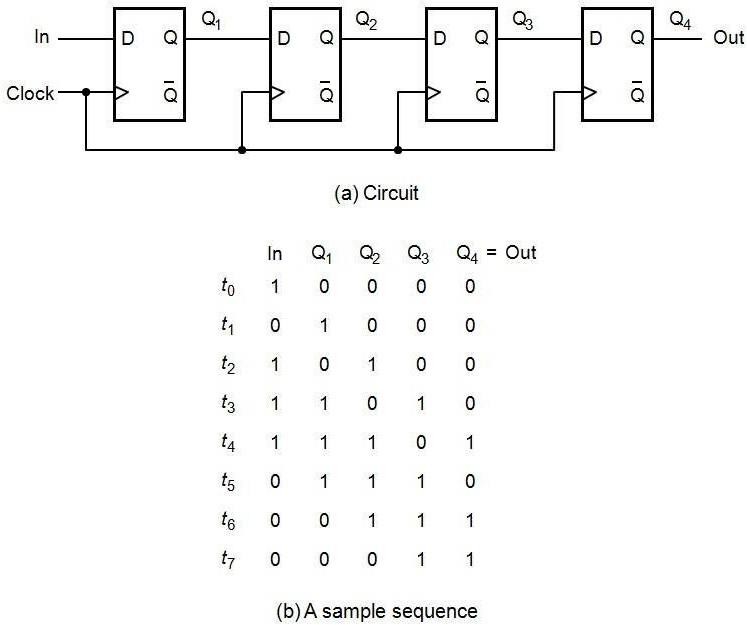
**Sequential Circuit Design**

* Steps in the design process for sequential circuits
* State Diagrams and State Tables Examples
* Steps in Design of a Sequential Circuit
  + 1. Specification – A description of the sequential circuit. Should include a detailing of the inputs, the outputs, and the operation. Possibly assumes that you have knowledge of digital system basics.
  + 2. Formulation: Generate a state diagram and/or a state table from the statement of the problem.
  + 3. State Assignment: From a state table assign binary codes to thestates.
  + 4. Flip-flop Input Equation Generation: Select the type of flip-flop for the circuit and generate the needed input for the required state transitions
  + 5. Output Equation Generation: Derive output logic equations for generation of the output from the inputs and current state.
  + 6. Optimization: Optimize the input and output equations. Today, CAD systemsare typically used for this in real systems.
  + 7. Technology Mapping: Generate a logic diagram of the circuit using ANDs, ORs, Inverters, and F/Fs.
  + 8. Verification: Use a HDL to verify the design

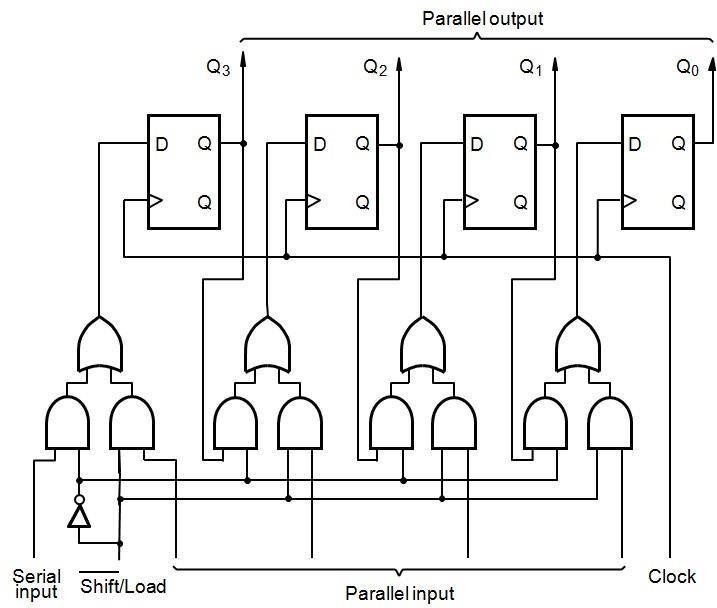
**Registers and Counters**

* An *n*-bit register is a cascade of *n* flip-flops and can store an *n*-bit binary data
* A counter can count occurrences of events and can generate timing intervals for control purposes

**A Simple Shift Register**



**Parallel-Access Shift Register**



**Counters**

* Counters are a specific type of sequential circuit.
* Like registers, the state, or the flip-flop values themselves, serves as the “output.”
* The output value increases by one on each clock cycle.
* After the largest value, the output “wraps around” back to 0.
* Using two bits, we’d get something like this:

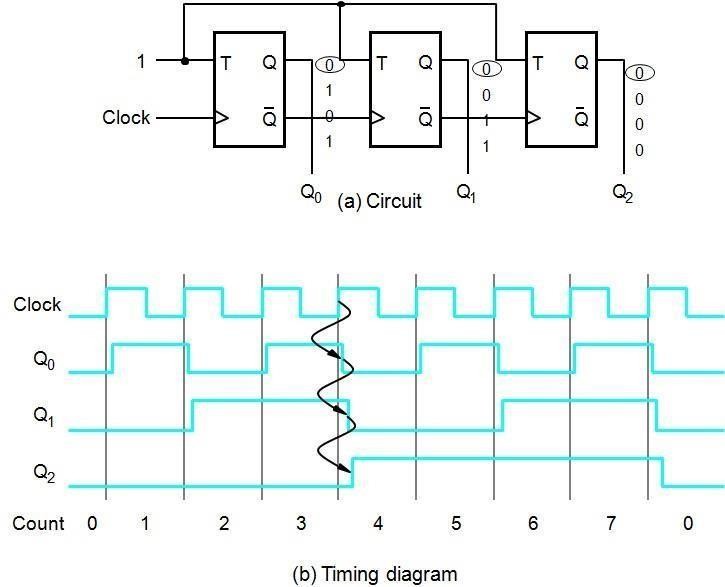
|  |  |
| --- | --- |
| Present State | Next State |
| A B | A B |
| 0 0 | 0 1 |
| 0 1 | 1 0 |
| 1 0 | 1 1 |
| 1 1 | 0 0 |

**Benefits of counters**

* Counters can act as simple clocks to keep track of “time.”
* You may need to record how many times something has happened.
  + How many bits have been sent or received?
  + How many steps have been performed in some computation?
* All processors contain a program counter, or PC.
  + Programs consist of a list of instructions that are to be executed one after another (for the most part).
  + The PC keeps track of the instruction currently being executed.
  + The PC increments once on each clock cycle, and the next program instruction is then executed.

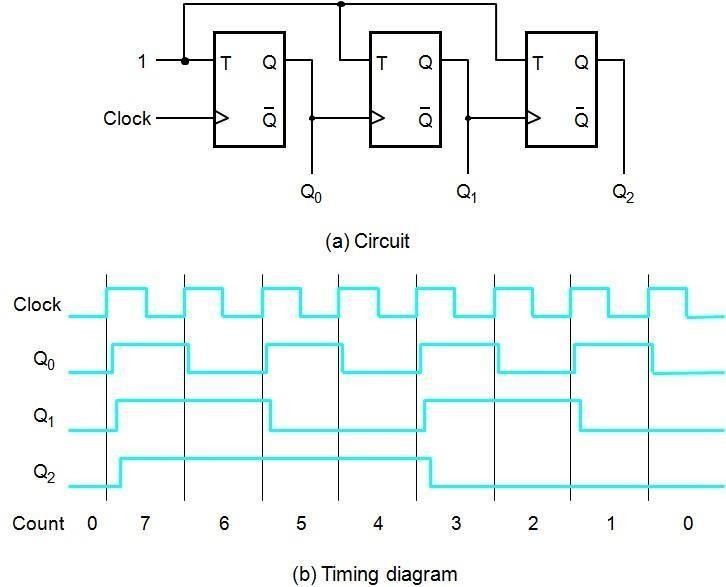
**A Three-Bit Up-Counter**

Q1 is connected to clk, Q2 and Q3 are clocked by Q’ of the preceding stage (hence called asynchronous or ripple counter



\

**A Three-Bit Down-Counter**



### Shift registers:

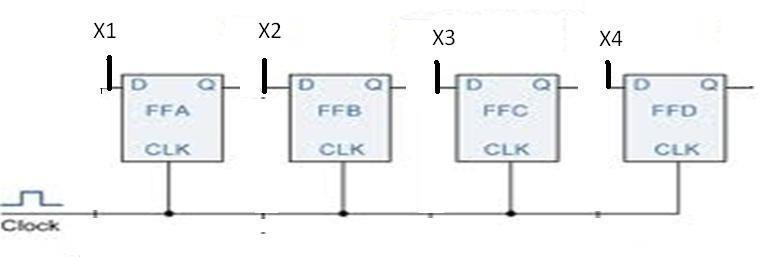
In digital circuits, a **shift register** is a cascade of flip-flops sharing the same clock, in which the output of each flip-flop is connected to the "data" input of the next flip-flop in the chain, resulting in a circuit that shifts by one position the "bit array" stored in it, *shifting in* the data present at its input and *shifting out* the last bit in the array, at each transition of the clock input. More generally, a **shift register** may be multidimensional, such that its "data in" and stage outputs are themselves bit arrays: this is implemented simply by running several shift registers of the same bit-length in parallel.

Shift registers can have both parallel and serial inputs and outputs. These are often configured as **serial-in, parallel-out** (SIPO) or as **parallel-in, serial-out** (PISO). There are also types that have both serial and parallel input and types with serial and parallel output. There are also **bi- directional** shift registers which allow shifting in both directions: L→R or R→L. The serial input and last output of a shift register can also be connected to create a **circular shift register**

Shift registers are a type of logic circuits closely related to counters. They are basically for the storage and transfer of digital data.

### Buffer register:

The buffer register is the simple set of registers. It is simply stores the binary word. The buffer may be controlled buffer. Most of the buffer registers used D Flip-flops.



### Figure: logic diagram of 4-bit buffer register

The figure shows a 4-bit buffer register. The binary word to be stored is applied to the data terminals. On the application of clock pulse, the output word becomes the same as the word applied at the terminals. i.e., the input word is loaded into the register by the application of clock pulse.

When the positive clock edge arrives, the stored word becomes: Q4Q3Q2Q1=X4X3X2X1

Q=X

### Controlled buffer register:

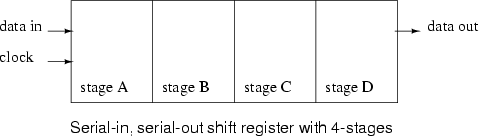
If goes LOW, all the FFs are RESET and the output becomes, Q=0000.

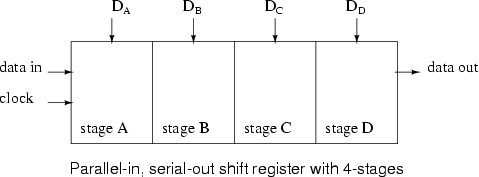
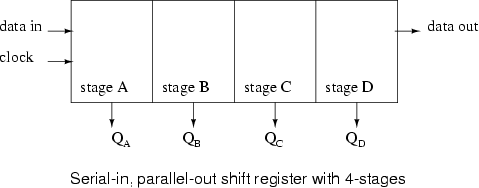
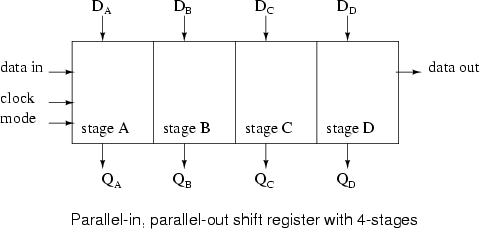
When is HIGH, the register is ready for action. LOAD is the control input. When LOAD is HIGH, the data bits X can reach the D inputs of FF‘s.

Q4Q3Q2Q1=X4X3X2X1 Q=X

When load is low, the X bits cannot reach the FF‘s.

### Data transmission in shift registers:

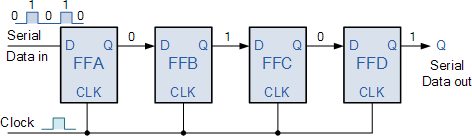




A number of ff‘s connected together such that data may be shifted into and shifted out of them is called shift register. Data may be shifted into or out of the register in serial form or in parallel form. There are four basic types of shift registers.

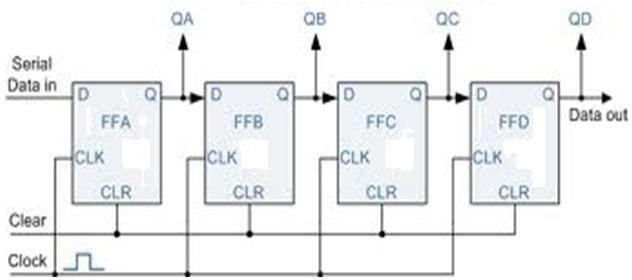
1. Serial in, serial out, shift right, shift registers
2. Serial in, serial out, shift left, shift registers
3. Parallel in, serial out shift registers
4. Parallel in, parallel out shift registers

### Serial IN, serial OUT, shift right, shift left register:

The logic diagram of 4-bit serial in serial out, right shift register with four stages. The register can store four bits of data. Serial data is applied at the input D of the first FF. the Q output of the first FF is connected to the D input of another FF. the data is outputted from the Q terminal of the last FF.

When serial data is transferred into a register, each new bit is clocked into the first FF at the positive going edge of each clock pulse. The bit that was previously stored by the first FF is transferred to the second FF. the bit that was stored by the Second FF is transferred to the third FF.

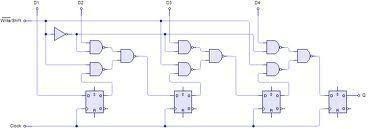
### Serial-in, parallel-out, shift register:



In this type of register, the data bits are entered into the register serially, but the data stored in the register is shifted out in parallel form.

Once the data bits are stored, each bit appears on its respective output line and all bits are available simultaneously, rather than on a bit-by-bit basis with the serial output. The serial-in, parallel out, shift register can be used as serial-in, serial out, shift register if the output is taken from the Q terminal of the last FF.

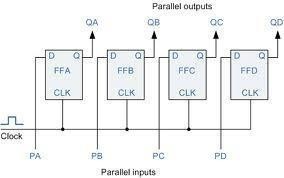
### Parallel-in, serial-out, shift register:



For a parallel-in, serial out, shift register, the data bits are entered simultaneously into their respective stages on parallel lines, rather than on a bit-by-bit basis on one line as with serial data bits are transferred out of the register serially. On a bit-by-bit basis over a single line.

There are four data lines A, B, C, D through which the data is entered into the register in parallel form. The signal shift/ load allows the data to be entered in parallel form into the register and the data is shifted out serially from terminalQ4

### Parallel-in, parallel-out, shift register

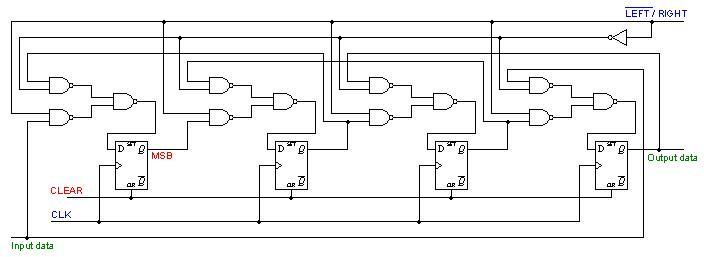


In a parallel-in, parallel-out shift register, the data is entered into the register in parallel form, and also the data is taken out of the register in parallel form. Data is applied to the D input terminals of the FF‘s. When a clock pulse is applied, at the positive going edge of the pulse, the D inputs are shifted into the Q outputs of the FFs. The register now stores the data. The stored data is available instantaneously for shifting out in parallel form.

### Bidirectional shift register:

A bidirectional shift register is one which the data bits can be shifted from left to right or from right to left. A fig shows the logic diagram of a 4-bit serial-in, serial out, bidirectional shift register. Right/left is the mode signal, when right /left is a 1, the logic circuit works as a shift-register. The bidirectional operation is achieved by using the mode signal and two NAND gates and one OR gate for each stage.

A HIGH on the right/left control input enables the AND gates G1, G2, G3 and G4 and disables the AND gates G5, G6, G7 and G8, and the state of Q output of each FF is passed through the gate to the D input of the following FF. when a clock pulse occurs, the data bits are then effectively shifted one place to the right. A LOW on the right/left control inputs enables the AND gates G5, G6, G7 and G8 and disables the And gates G1, G2, G3 and G4 and the Q output of each FF is passed to the D input of the preceding FF. when a clock pulse occurs, the data bits are then effectively shifted one place to the left. Hence, the circuit works as a bidirectional shift register



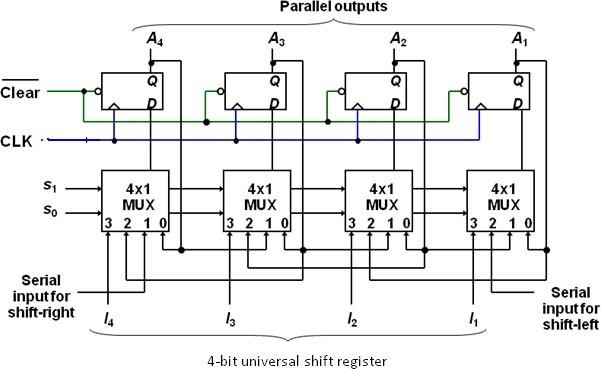
### Figure: logic diagram of a 4-bit bidirectional shift register Universal shift register:

A register is capable of shifting in one direction only is a unidirectional shift register. One that can shift both directions is a bidirectional shift register. If the register has both shifts and parallel load capabilities, it is referred to as a universal shift registers. Universal shift register is a bidirectional register, whose input can be either in serial form or in parallel form and whose output also can be in serial form or I parallel form.

The most general shift register has the following capabilities.

* 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 serial input and output lines associated with the shift-right
  4. A shift-left control to enable the shift-left operation and serial input and output lines associated with the shift-left
  5. A parallel loads control to enable a parallel transfer and the n input lines associated with the parallel transfer
  6. N parallel output lines
  7. A control state that leaves the information in the register unchanged in the presence of the clock.

A universal shift register can be realized using multiplexers. The below fig shows the logic diagram of a 4-bit universal shift register that has all capabilities. It consists of 4 D flip-flops and four multiplexers. The four multiplexers have two common selection inputs s1 and s0. Input 0 in each multiplexer is selected when S1S0=00, input 1 is selected when S1S0=01 and input 2 is selected when S1S0=10 and input 4 is selected when S1S0=11. The selection inputs control the mode of operation of the register according to the functions entries. When S1S0=0, the present value of the register is applied to the D inputs of flip-flops. The condition forms a path from the output of each flip-flop into the input of the same flip-flop. The next clock edge transfers into each flip-flop the binary value it held previously, and no change of state occurs. When S1S0=01, terminal 1 of the multiplexer inputs have a path to the D inputs of the flip-flop. This causes a shift-right operation, with serial input transferred into flip-flopA4. When S1S0=10, a shift left operation results with the other serial input going into flip-flop A1. Finally when S1S0=11, the binary information on the parallel input lines is transferred into the register simultaneously during the next clock cycle



### Figure: logic diagram 4-bit universal shift register

**Function table for the register**

|  |  |  |
| --- | --- | --- |
| **mode control** | | |
| **S0** | **S1** | **register operation** |
|  |  |  |
| 0 | 0 | No change |
| 0 | 1 | Shift Right |
| 1 | 0 | Shift left |
| 1 | 1 | Parallel load |

**Counters:**

**Counter** is a device which stores (and sometimes displays) the number of times particular event or process has occurred, often in relationship to a clock signal. A Digital counter is a set of flip flops whose state change in response to pulses applied at the input to the counter. Counters may be asynchronous counters or synchronous counters. Asynchronous counters are also called ripple counters

In electronics counters can be implemented quite easily using register-type circuits such as the flip-flops and a wide variety of classifications exist:

* Asynchronous (ripple) counter – changing state bits are used as clocks to subsequent state flip-flops
* Synchronous counter – all state bits change under control of a single clock
* Decade counter – counts through ten states per stage
* Up/down counter – counts both up and down, under command of a control input
* Ring counter – formed by a shift register with feedback connection in a ring
* Johnson counter – a *twisted* ring counter Cascaded counter

Modulus counter.

Each is useful for different applications. Usually, counter circuits are digital in nature, and count in natural binary Many types of counter circuits are available as digital building blocks, for example a number of chips in the 4000 series implement different counters.

Occasionally there are advantages to using a counting sequence other than the natural binary sequence such as the binary coded decimal counter, a linear feed-back shift register counter, or a gray-code counter.

Counters are useful for digital clocks and timers, and in oven timers, VCR clocks, etc.

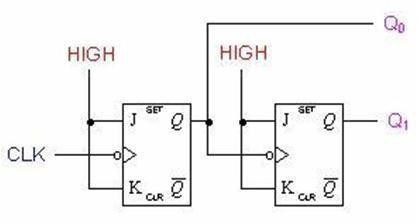
**Asynchronous counters:**

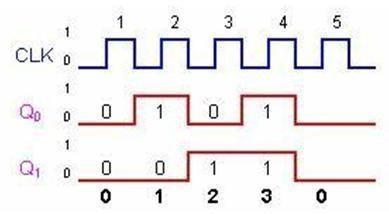
An asynchronous (ripple) counter is a single [JK-type flip-flop,](http://en.wikipedia.org/wiki/Flip-flop_(electronics)#JK_flip-flop) with its J (data) input fed from its own inverted output. This circuit can store one bit, and hence can count from zero to one before it overflows (starts over from 0). This counter will increment once for every clock cycle and takes two clock cycles to overflow, so every cycle it will alternate between a transition from 0 to 1 and a transition from 1 to 0. Notice that this creates a new clock with a 50% [duty cycle](http://en.wikipedia.org/wiki/Duty_cycle) at exactly half the frequency of the input clock. If this output is then used as the clock signal for a similarly arranged D flip-flop (remembering to invert the output to the input), one will get another 1 bit counter that counts half as fast. Putting them together yields a two-bit counter:

**Two-bit ripple up-counter using negative edge triggered flip flop:**

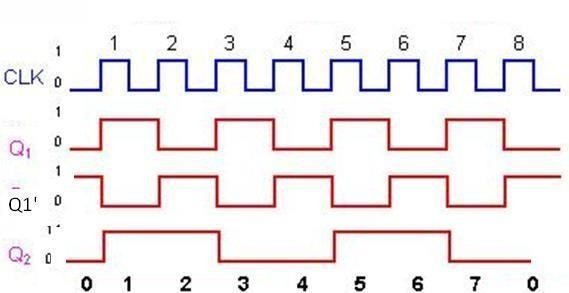
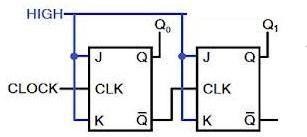
Two bit ripple counter used two flip-flops. There are four possible states from 2 – bit up- counting I.e. 00, 01, 10 and 11.

* The counter is initially assumed to be at a state 00 where the outputs of the tow flip-flops are noted as Q1Q0. Where Q1 forms the MSB and Q0 forms the LSB.
* For the negative edge of the first clock pulse, output of the first flip-flop FF1 toggles its state. Thus Q1 remains at 0 and Q0 toggles to 1 and the counter state are now read as 01.
* During the next negative edge of the input clock pulse FF1 toggles and Q0 = 0. The output Q0 being a clock signal for the second flip-flop FF2 and the present transition acts as a negative edge for FF2 thus toggles its state Q1 = 1. The counter state is now read as 10.
* For the next negative edge of the input clock to FF1 output Q0 toggles to 1. But this transition from 0 to 1 being a positive edge for FF2 output Q1 remains at 1. The counter state is now read as 11.
* For the next negative edge of the input clock, Q0 toggles to 0. This transition from 1 to 0 acts as a negative edge clock for FF2 and its output Q1 toggles to 0. Thus the starting state 00 is attained. Figure shown below





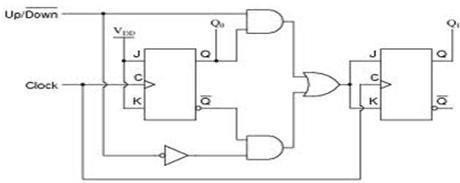
**Two-bit ripple down-counter using negative edge triggered flip flop:**



A 2-bit down-counter counts in the order 0,3,2,1,0,1…….,i.e, 00,11,10,01,00,11 …..,etc. the above fig. shows ripple down counter, using negative edge triggered J-K FFs and its timing diagram.

* + For down counting, Q1‘ of FF1 is connected to the clock of Ff2. Let initially all the FF1 toggles, so, Q1 goes from a 0 to a 1 and Q1‘ goes from a 1 to a 0.
  + The negative-going signal at Q1‘ is applied to the clock input of FF2, toggles Ff2 and, therefore, Q2 goes from a 0 to a 1.so, after one clock pulse Q2=1 and Q1=1, I.e., the state of the counter is 11.
  + At the negative-going edge of the second clock pulse, Q1 changes from a 1 to a 0 and Q1‘ from a 0 to a 1.
  + This positive-going signal at Q1‘ does not affect FF2 and, therefore, Q2 remains at a 1. Hence , the state of the counter after second clock pulse is 10
  + At the negative going edge of the third clock pulse, FF1 toggles. So Q1, goes from a 0 to a 1 and Q1‘ from 1 to 0. This negative going signal at Q1‘ toggles FF2 and, so, Q2 changes from 1 to 0, hence, the state of the counter after the third clock pulse is 01.
  + At the negative going edge of the fourth clock pulse, FF1 toggles. So Q1, goes from a 1 to a 0 and Q1‘ from 0 to 1. . This positive going signal at Q1‘ does not affect FF2 and, so, Q2 remains at 0, hence, the state of the counter after the fourth clock pulse is 00.

**Two-bit ripple up-down counter using negative edge triggered flip flop:**



**Figure: asynchronous 2-bit ripple up-down counter using negative edge triggered flip flop:**

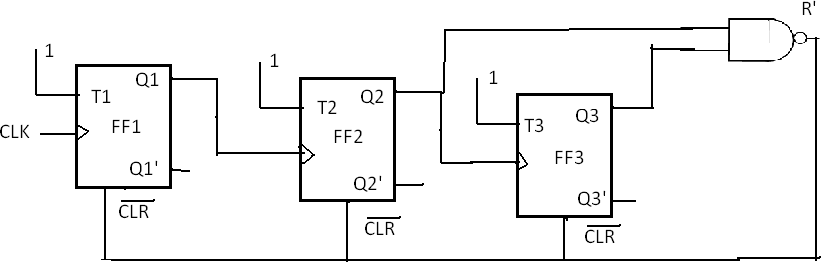
* + As the name indicates an up-down counter is a counter which can count both in upward and downward directions. An up-down counter is also called a forward/backward counter or a bidirectional counter. So, a control signal or a mode signal M is required to choose the direction of count. When M=1 for up counting, Q1 is transmitted to clock of FF2 and when M=0 for down counting, Q1‘ is transmitted to clock of FF2. This is achieved by using two AND gates and one OR gates. The external clock signal is applied to FF1.
  + Clock signal to FF2= (Q1.Up)+(Q1‘. Down)= Q1m+Q1‘M‘

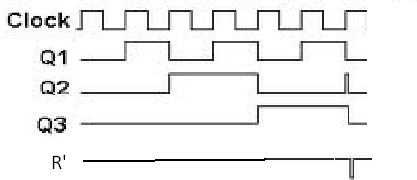
### Design of Asynchronous counters:

To design a asynchronous counter, first we write the sequence , then tabulate the values of reset signal R for various states of the counter and obtain the minimal expression for R and R‘ using K-Map or any other method. Provide a feedback such that R and R‘ resets all the FF‘s after the desired count

### Design of a Mod-6 asynchronous counter using T FFs:

A mod-6 counter has six stable states 000, 001, 010, 011, 100, and 101. When the sixth clock pulse is applied, the counter temporarily goes to 110 state, but immediately resets to 000 because of the feedback provided. it is ―divide by-6-counter‖, in the sense that it divides the input clock frequency by 6.it requires three FFs, because the smallest value of n satisfying the conditionN≤2n is n=3; three FFs can have 8 possible states, out of which only six are utilized and the remaining two states 110and 111, are invalid. If initially the counter is in 000 state, then after the sixth clock pulse, it goes to 001, after the second clock pulse, it goes to 010, and so on.





**After s**ixth clock pulse it goes to 000. For the design, write the truth table with present state outputs Q3, Q2 and Q1 as the variables, and reset R as the output and obtain an expression for R in terms of Q3, Q2, and Q1that decides the feedback into be provided. From the truth table, R=Q3Q2. For active-low Reset, R‘ is used. The reset pulse is of very short duration, of the order of nanoseconds and it is equal to the propagation delay time of the NAND gate used. The expression for R can also be determined as follows.

Therefore,

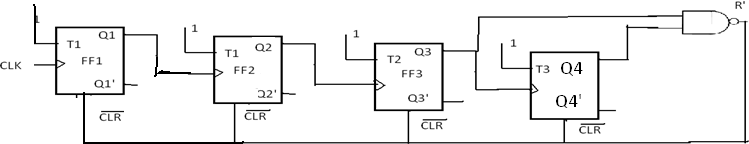
R=0 for 000 to 101, R=1 for 110, and R=X=for111 R=Q3Q2Q1‘+Q3Q2Q1=Q3Q2

The logic diagram and timing diagram of Mod-6 counter is shown in the above fig. The truth table is as shown in below.

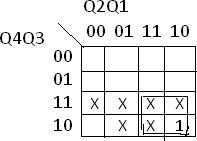
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| After pulses | States | |  |  |
| Q3 | Q2 | Q1 | R |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 1 | 0 |
| 4 | 1 | 0 | 0 | 0 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 1 | 1 | 0 | 1 |
|  | 0 | 0 | 0 | 0 |
| 7 | 0 | 0 | 0 | 0 |

### Design of a mod-10 asynchronous counter using T-flip-flops:

A mod-10 counter is a decade counter. It also called a BCD counter or a divide-by-10 counter. It requires four flip-flops (condition 10 ≤2n is n=4). So, there are 16 possible states, out of which ten are valid and remaining six are invalid. The counter has ten stable state, 0000 through 1001, i.e., it counts from 0 to 9. The initial state is 0000 and after nine clock pulses it goes to 1001. When the tenth clock pulse is applied, the counter goes to state 1010 temporarily, but because of the feedback provided, it resets to initial state 0000. So, there will be a glitch in the waveform of Q2. The state 1010 is a temporary state for which the reset signal R=1, R=0 for 0000 to 1001, and R=C for 1011 to 1111.



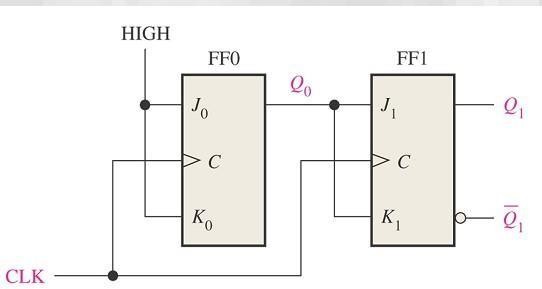
The count table and the K-Map for reset are shown in fig. from the K-Map R=Q4Q2. So, feedback is provided from second and fourth FFs. For active –HIGH reset, Q4Q2 is applied to the clear terminal. For active-LOW reset 4 2 is connected isof all Flip=flops.

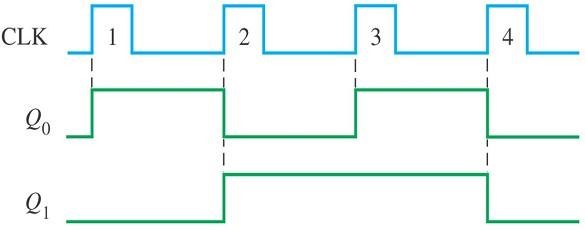


|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| After pulses | Count | |  |  |
| Q4 | Q3 | Q2 | Q1 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 |
| 6 | 0 | 1 | 1 | 0 |
| 7 | 0 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 |
| 9 | 0 | 1 | 0 | 1 |
| 10 | 0 | 0 | 0 | 0 |

### Synchronous counters:

Asynchronous counters are serial counters. They are slow because each FF can change state only if all the preceding FFs have changed their state. if the clock frequency is very high, the asynchronous counter may skip some of the states. This problem is overcome in synchronous counters or parallel counters. Synchronous counters are counters in which all the flip flops are triggered simultaneously by the clock pulses Synchronous counters have a common clock pulse applied simultaneously to all flip- -Bit Synchronous Binary Counter





### Design of synchronous counters:

For a systematic design of synchronous counters. The following procedure is used.

**Step 1:**State Diagram: draw the state diagram showing all the possible states state diagram which also be called nth transition diagrams, is a graphical means of depicting the sequence of states through which the counter progresses.

**Step2:** number of flip-flops: based on the description of the problem, determine the required number n of the flip-flops- the smallest value of n is such that the number of states N≤2n--- and the desired counting sequence.

**Step3:** choice of flip-flops excitation table: select the type of flip-flop to be used and write the excitation table. An excitation table is a table that lists the present state (ps) , the next state(ns) and required excitations.

**Step4**: minimal expressions for excitations: obtain the minimal expressions for the excitations of the FF using K-maps drawn for the excitation of the flip-flops in terms of the present states and inputs.

**Step5**: logic diagram: draw a logic diagram based on the minimal expressions

### Design of a synchronous 3-bit up-down counter using JK flip-flops:

**Step1:** determine the number of flip-flops required. A 3-bit counter requires three FFs. It has 8 states (000,001,010,011,101,110,111) and all the states are valid. Hence no don‘t cares. For selecting up and down modes, a control or mode signal M is required. When the mode signal M=1 and counts down when M=0. The clock signal is applied to all the FFs simultaneously.

**Step2:** draw the state diagrams: the state diagram of the 3-bit up-down counter is drawn as

**Step3:** select the type of flip flop and draw the excitation table: JK flip-flops are selected and the excitation table of a 3-bit up-down counter using JK flip-flops is drawn as shown in fig.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PS | | | mode | NS | | | required excitations | | | | |  |
| Q3 | Q2 | Q1 | M | Q3 | Q2 | Q1 | J3 | K3 | J2 | K2 | J1 | K1 |
| 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | x | 1 | x | 1 | x |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | x | 0 | x | 1 | x |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | x | 0 | x | x | 1 |
| 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | x | 1 | x | x | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | x | x | 1 | 1 | x |
| 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | x | x | 0 | 1 | x |
| 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | x | x | 0 | x | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | x | x | 1 | x | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | x | 1 | 1 | x | 1 | x |
| 1 | 0 | 0 | 1 | 1 | 0 | 1 | x | 0 | 0 | x | 1 | x |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 | x | 0 | 0 | x | x | 1 |
| 1 | 0 | 1 | 1 | 1 | 1 | 0 | x | 0 | 1 | x | x | 1 |
| 1 | 1 | 0 | 0 | 1 | 0 | 1 | x | 0 | x | 1 | 1 | x |
| 1 | 1 | 0 | 1 | 1 | 1 | 1 | x | 0 | x | 0 | 1 | x |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | x | 0 | x | 0 | x | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | x | 1 | x | 1 | x | 1 |

**Step4:** obtain the minimal expressions: From the excitation table we can conclude that J1=1 and K1=1, because all the entries for J1and K1 are either X or 1. The K-maps for J3, K3,J2 and K2 based on the excitation table and the minimal expression obtained from them are shown in fig.

00 01 11 10

Q3Q2 Q1M

1

X

X

X

1

X

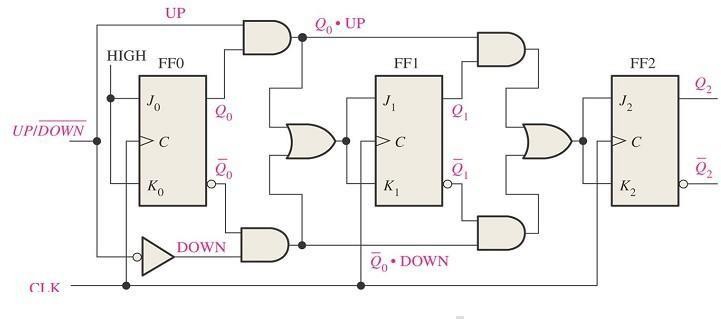
X

**Step5:** draw the logic diagram: a logic diagram using those minimal expressions can be drawn as shown in fig.

X

X

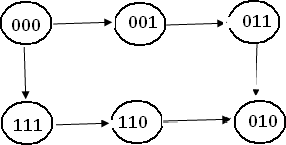
X



### Design of a synchronous modulo-6 gray cod counter:

**Step 1:** the number of flip-flops: we know that the counting sequence for a modulo-6 gray code counter is 000, 001, 011, 010, 110, and 111. It requires n=3FFs (N≤2n, i.e., 6≤23). 3 FFs can have 8 states. So the remaining two states 101 and 100 are invalid. The entries for excitation corresponding to invalid states are don‘t cares.

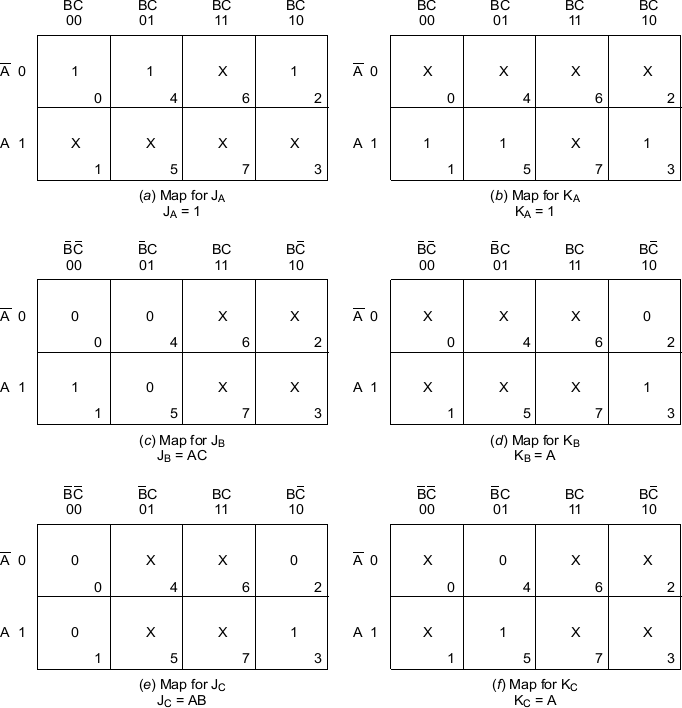
**Step2:** the state diagram: the state diagram of the mod-6 gray code converter is drawn as shown in fig.



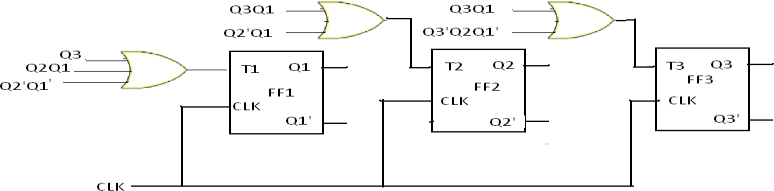
**Step3:** type of flip-flop and the excitation table: T flip-flops are selected and the excitation table of the mod-6 gray code counter using T-flip-flops is written as shown in fig.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **PS** | |  | **NS** | |  | **required excitations** | | |
| **Q3** | **Q2** | **Q1** | **Q3** | **Q2** | **Q1** | **T3** | **T2** | **T1** |
| 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |

**Step4:** The minimal expressions: the K-maps for excitations of FFs T3,T2,and T1 in terms of outputs of FFs Q3,Q2, and Q1, their minimization and the minimal expressions for excitations obtained from them are shown if fig



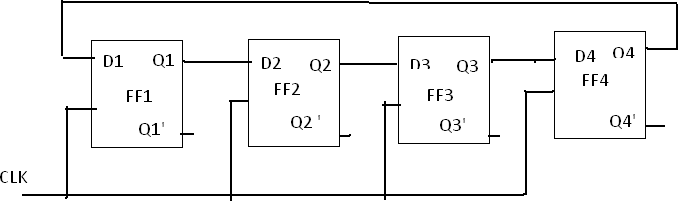
**Step5:** the logic diagram: the logic diagram based on those minimal expressions is drawn as shown in fig.



### Shift register counters:

One of the applications of shift register is that they can be arranged to form several types of counters. The most widely used shift register counter is ring counter as well as the twisted ring counter.

**Ring counter:** this is the simplest shift register counter. The basic ring counter using D flip- flops is shown in fig. the realization of this counter using JK FFs. The Q output of each stage is connected to the D flip-flop connected back to the ring counter.

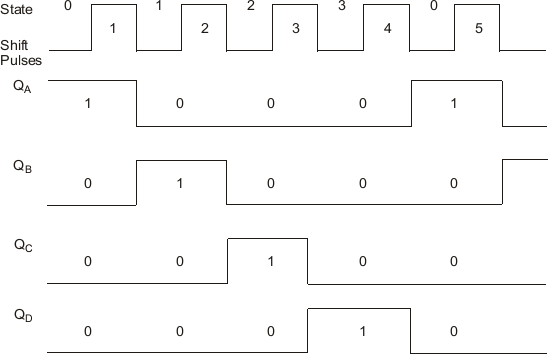


### FIGURE: logic diagram of 4-bit ring counter using D flip-flops

Only a single 1 is in the register and is made to circulate around the register as long as clock pulses are applied. Initially the first FF is present to a 1. So, the initial state is 1000, i.e., Q1=1, Q2=0,Q3=0,Q4=0. After each clock pulse, the contents of the register are shifted to the right by one bit and Q4 is shifted back to Q1. The sequence repeats after four clock pulses. The number

of distinct states in the ring counter, i.e., the mod of the ring counter is equal to number of FFs used in the counter. An n-bit ring counter can count only n bits, where as n-bit ripple counter can count 2n bits. So, the ring counter is uneconomical compared to a ripple counter but has advantage of requiring no decoder, since we can read the count by simply noting which FF is set. Since it is entirely a synchronous operation and requires no gates external FFs, it has the further advantage of being very fast.

### Timing diagram:



**Twisted Ring counter (Johnson counter):**

This counter is obtained from a serial-in, serial-out shift register by providing feedback from the inverted output of the last FF to the D input of the first FF. the Q output of each is connected to the D input of the next stage, but the Q‘ output of the last stage is connected to the D input of the first stage, therefore, the name twisted ring counter. This feedback arrangement produces a unique sequence of states.

The logic diagram of a 4-bit Johnson counter using D FF is shown in fig. the realization of the same using J-K FFs is shown in fig.. The state diagram and the sequence table are shown in figure. The timing diagram of a Johnson counter is shown infigure.

Let initially all the FFs be reset, i.e., the state of the counter be 0000. After each clock pulse, the level of Q1 is shifted to Q2, the level of Q2to Q3, Q3 to Q4 and the level of Q4‘to Q1 and the sequences given in fig.

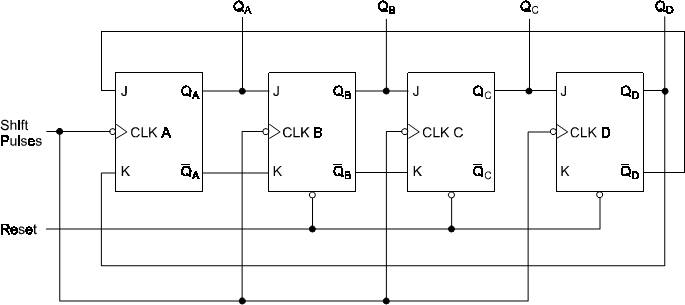


Figure: Johnson counter with JK flip-flops

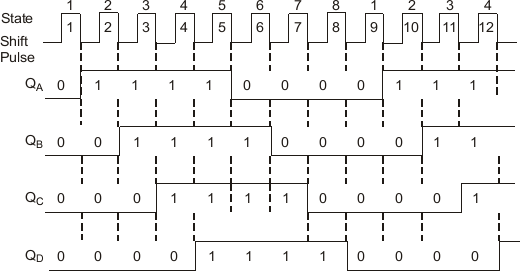


Figure: timing diagram

SEQUENCE GENERATOR

A sequence generator is a sequential circuit which generates a prescribed sequence at its output. The output sequence is in synchronization with the clock input. It is possible to design a sequence generator using counters or using the shift registers.

Sequence Generator using Shift Register: The sequence generator is a circuit which generates a desired sequence bf bits at its output in synchronization with the clock.

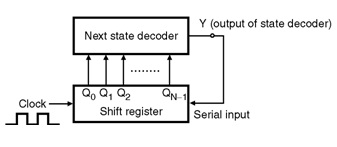
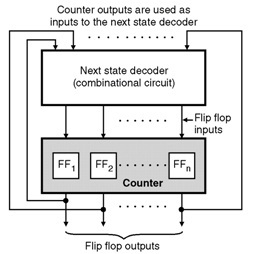


Figure shows the basic structure of a sequence generator.  
 The outputs of an N bit shift register (Q0 through QNâ€“1) are applied as inputs to a combinational circuit called Next state decoder. And the output (Y) of the next state decoder is applied to the serial input of the shift register. The Next state decoder is designed according to the required sequence

Sequence Generator using Counters:

The general block diagram of a sequence generator using counter is shown in Figure below.



The next state decoder is a combinational circuit. The input to it are obtained from the flip-flop outputs and its outputs are applied to the inputs of the flip-flops.

Some of the applications of the sequence generator are as follows :

1. Random bit generator
2. Counters
3. Code generators
4. Period and sequence generator.

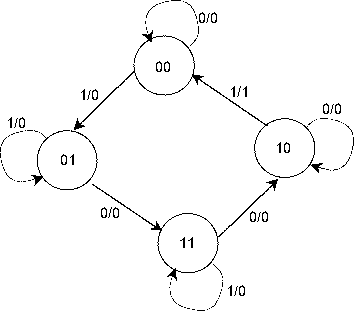
State Tables and State Diagrams

We have examined a general model for sequential circuits. In this model the effect of all previous inputs on the outputs is represented by a state of the circuit. Thus, the output of the circuit at any time depends upon its current state and the input. These also determine the next state of the circuit. The relationship that exists among the inputs, outputs, present states and next states can be specified by either the state table or the state diagram.

State Table

The state table representation of a sequential circuit consists of three sections labelled present state, next state and output. The present state designates the state of flip-flops before the occurrence of a clock pulse. The next state shows the states of flip-flops after the clock pulse, and the output section lists the value of the output variables during the present state.

State Diagram

In addition to graphical symbols, tables or equations, flip-flops can also be represented graphically by a state diagram. In this diagram, a state is represented by a circle, and the transition between states is indicated by directed lines (or arcs) connecting the circles. An example of a state diagram is shown in Figure 3 below.

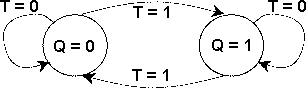
Figure

3. State Diagram

The binary number inside each circle identifies the state the circle represents. The directed lines are labeled with two binary numbers separated by a slash (/). The input value that causes the state transition is labeled first. The number after the slash symbol / gives the value of the output. For example, the directed line from state 00 to 01 is labeled 1/0, meaning that, if the sequential circuit is in a present state and the input is 1, then the next state is 01 and the output is 0. If it is in a present state 00 and the input is 0, it will remain in that state. A directed line connecting a circle with itself indicates that no change of state occurs. The state diagram provides exactly the same information as the state table and is obtained directly from the state table.

State Diagrams of Various Flip-flops

|  |  |
| --- | --- |
| NAME | STATE DIAGRAM |
| SR |  |
| JK |  |
| D |  |



T

**Outcomes:**

i. Able to understand the concept of sequential logic circuit.

ii. Design and Implement sequential circuit using Logic Gate and flip flop.

**UNIT IV**

**MEMORY DEVICES**

# Pre requisition:

# i. Basic Knowledge in digital logic gates.

# ii. Basic Knowledge in combinational and sequential logic.

# iii. Basic Knowledge in minimization techniques.

**MEMORY INTRODUCTION:**

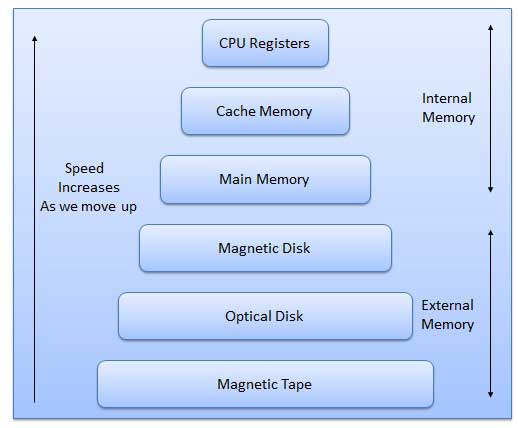
A memory is just like a human brain. It is used to store data and instruction. Computer memory is the storage space in computer where data is to be processed and instructions required for processing are stored.

The memory is divided into large number of small parts. Each part is called a cell. Each location or cell has a unique address which varies from zero to memory size minus one.

For example if computer has 64k words, then this memory unit has 64 \* 1024 = 65536 memory location. The address of these locations varies from 0 to 65535.

Memory is primarily of two types

* **Internal Memory** − cache memory and primary/main memory
* **External Memory** − magnetic disk / optical disk etc.



Characteristics of Memory Hierarchy are following when we go from top to bottom.

* Capacity in terms of storage increases.
* Cost per bit of storage decreases.
* Frequency of access of the memory by the CPU decreases.
* Access time by the CPU increases.

## **RAM**

A RAM constitutes the internal memory of the CPU for storing data, program and program result. It is read/write memory. It is called random access memory (RAM).

Since access time in RAM is independent of the address to the word that is, each storage location inside the memory is as easy to reach as other location & takes the same amount of time. We can reach into the memory at random & extremely fast but can also be quite expensive.

RAM is volatile, i.e. data stored in it is lost when we switch off the computer or if there is a power failure. Hence, a backup uninterruptible power system (UPS) is often used with computers. RAM is small, both in terms of its physical size and in the amount of data it can hold.

RAM is of two types

* Static RAM (SRAM)
* Dynamic RAM (DRAM)

### Static RAM (SRAM)

The word **static** indicates that the memory retains its contents as long as power remains applied. However, data is lost when the power gets down due to volatile nature. SRAM chips use a matrix of 6-transistors and no capacitors. Transistors do not require power to prevent leakage, so SRAM need not have to be refreshed on a regular basis.

Because of the extra space in the matrix, SRAM uses more chips than DRAM for the same amount of storage space, thus making the manufacturing costs higher.

Static RAM is used as cache memory needs to be very fast and small.

### Dynamic RAM (DRAM)

DRAM, unlike SRAM, must be continually refreshed in order for it to maintain the data. This is done by placing the memory on a refresh circuit that rewrites the data several hundred times per second. DRAM is used for most system memory because it is cheap and small. All DRAMs are made up of memory cells. These cells are composed of one capacitor and one transistor.

## **ROM**

ROM stands for Read Only Memory. The memory from which we can only read but cannot write on it. This type of memory is non-volatile. The information is stored permanently in such memories during manufacture.

A ROM, stores such instruction as are required to start computer when electricity is first turned on, this operation is referred to as bootstrap. ROM chip are not only used in the computer but also in other electronic items like washing machine and microwave oven.

Following are the various types of ROM −

### MROM (Masked ROM)

The very first ROMs were hard-wired devices that contained a pre-programmed set of data or instructions. These kind of ROMs are known as masked ROMs. It is inexpensive ROM.

### PROM (Programmable Read Only Memory)

PROM is read-only memory that can be modified only once by a user. The user buys a blank PROM and enters the desired contents using a PROM programmer. Inside the PROM chip there are small fuses which are burnt open during programming. It can be programmed only once and is not erasable.

### EPROM (Erasable and Programmable Read Only Memory)

The EPROM can be erased by exposing it to ultra-violet light for a duration of upto 40 minutes. Usually, an EPROM eraser achieves this function. During programming an electrical charge is trapped in an insulated gate region. The charge is retained for more than ten years because the charge has no leakage path. For erasing this charge, ultra-violet light is passed through a quartz crystal window (lid). This exposure to ultra-violet light dissipates the charge. During normal use the quartz lid is sealed with a sticker.

### EEPROM (Electrically Erasable and Programmable Read Only Memory)

The EEPROM is programmed and erased electrically. It can be erased and reprogrammed about ten thousand times. Both erasing and programming take about 4 to 10 ms (millisecond). In EEPROM, any location can be selectively erased and programmed. EEPROMs can be erased one byte at a time, rather than erasing the entire chip. Hence, the process of re-programming is flexible but slow.

## **Serial Access Memory**

Sequential access means the system must search the storage device from the beginning of the memory address until it finds the required piece of data. Memory device which supports such access is called a Sequential Access Memory or Serial Access Memory. Magnetic tape is an example of serial access memory.

## **Direct Access Memory**

Direct access memory or Random Access Memory, refers to conditions in which a system can go directly to the information that the user wants. Memory device which supports such access is called a Direct Access Memory. Magnetic disks, optical disks are examples of direct access memory.

## **Cache Memory**

Cache memory is a very high speed semiconductor memory which can speed up CPU. It acts as a buffer between the CPU and main memory. It is used to hold those parts of data and program which are most frequently used by CPU. The parts of data and programs, are transferred from disk to cache memory by operating system, from where CPU can access them.

### Advantages

* Cache memory is faster than main memory.
* It consumes less access time as compared to main memory.
* It stores the program that can be executed within a short period of time.
* It stores data for temporary use.

### Disadvantages

* Cache memory has limited capacity.
* It is very expensive.

Virtual memory is a technique that allows the execution of processes which are not completely available in memory. The main visible advantage of this scheme is that programs can be larger than physical memory. Virtual memory is the separation of user logical memory from physical memory.

This separation allows an extremely large virtual memory to be provided for programmers when only a smaller physical memory is available. Following are the situations, when entire program is not required to be loaded fully in main memory.

* User written error handling routines are used only when an error occurred in the data or computation.
* Certain options and features of a program may be used rarely.
* Many tables are assigned a fixed amount of address space even though only a small amount of the table is actually used.
* The ability to execute a program that is only partially in memory would counter many benefits.
* Less number of I/O would be needed to load or swap each user program into memory.
* A program would no longer be constrained by the amount of physical memory that is available.
* Each user program could take less physical memory, more programs could be run the same time, with a corresponding increase in CPU utilization and throughput.

## **Auxiliary Memory**

Auxiliary memory is much larger in size than main memory but is slower. It normally stores system programs, instruction and data files. It is also known as secondary memory. It can also be used as an overflow/virtual memory in case the main memory capacity has been exceeded. Secondary memories cannot be accessed directly by a processor. First the data/information of auxiliary memory is transferred to the main memory and then that information can be accessed by the CPU. Characteristics of Auxiliary Memory are following −

* **Non-volatile memory** − Data is not lost when power is cut off.
* **Reusable** − The data stays in the secondary storage on permanent basis until it is not overwritten or deleted by the user.
* **Reliable** − Data in secondary storage is safe because of high physical stability of secondary storage device.
* **Convenience** − With the help of a computer software, authorised people can locate and access the data quickly.
* **Capacity** − Secondary storage can store large volumes of data in sets of multiple disks.
* **Cost** − It is much lesser expensive to store data on a tape or disk than primary memory.

MEMORY CLASSIFICATION:

There are two types of memories that are used in digital systems:

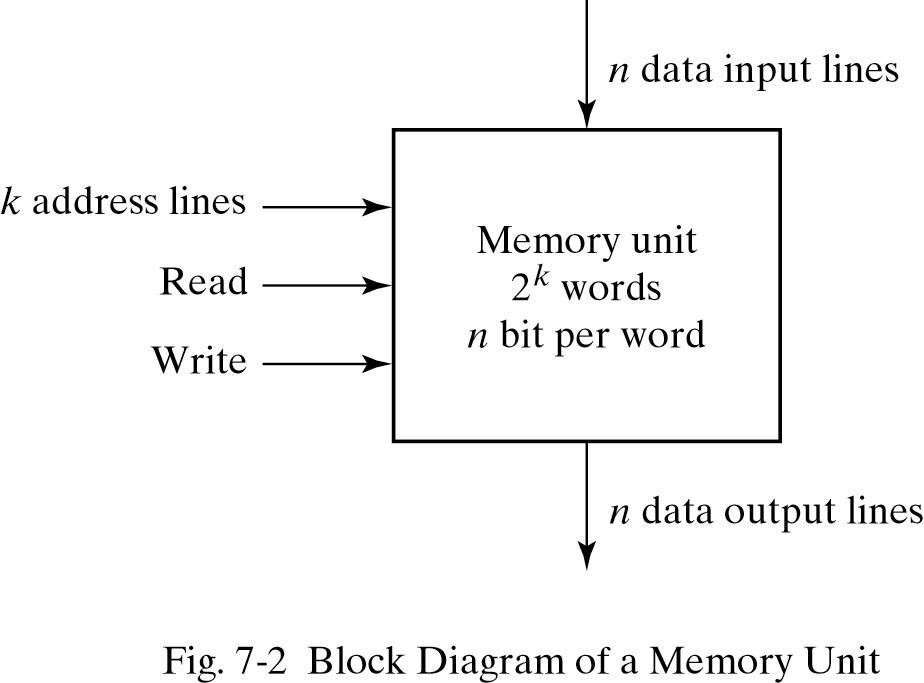
* + Random-access memory (RAM): perform both the write and read operations.
  + Read-only memory (ROM): perform only the read operation.

The read-only memory is a programmable logic device. Other such units are the programmable logic array (PLA), the programmable array logic(PAL), and the field-programmable gate array(FPGA).

A memory unit stores binary information in groups of bits called words.

* + - byte = 8 bits
      * word = 2 bytes

The communication between a memory and its environment is achieved through data input and output lines, address selection lines, and control lines that specify the direction of transfer.



## Random-Access Memory:

In random-access memory, the word locations may be thought of as being separated in space, with each word occupying one particular location.

In sequential-access memory, the information stored in some medium is not immediately accessible, but is available only certain intervals of time. A magnetic disk or tape unit is of this type.

In a random-access memory, the access time is always the same regardless of the particular locationof the word.

In a sequential-access memory, the time it takes to access a word depends on the position of the word with respect to the reading head position; therefore, the access time is variable.



# Static RAM

SRAM consists essentially of internal latches that store the binary information. The stored information remains valid as long as power is applied to the unit.

SRAM is easier to use and has shorter read and write cycles.

Low density, low capacity, high cost, high speed, high power consumption.

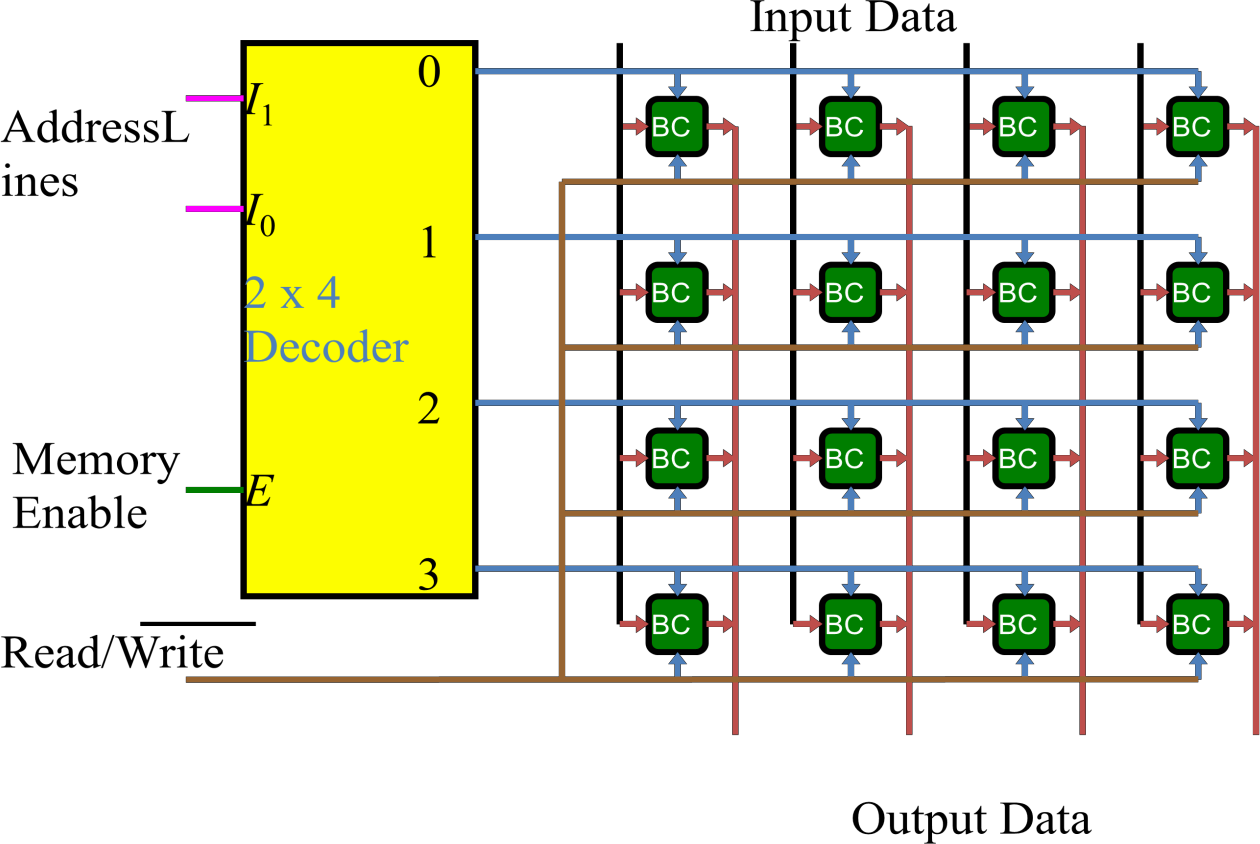
# Dynamic RAM

DRAM stores the binary information in the form of electric charges on capacitors. The capacitors are provided inside the chip by MOS transistors.

The capacitors tends to discharge with time and must be periodically recharged by refreshing the dynamic memory.

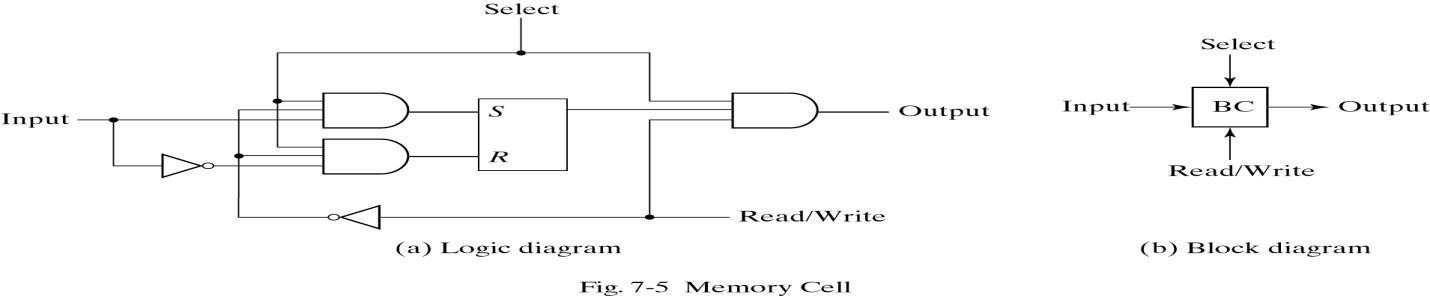
DRAM offers reduced power consumption and larger storage capacity in a single memory chip. High density, high capacity, low cost, low speed, low power consumption.

## Memory decoding



The equivalent logic of a binary cell that stores one bit of information is shown below.

* Read/Write = 0, select = 1, input data to S-R latch
* Read/Write = 1, select = 1, output data from S-R latch



**PROGRAMMABLE LOGIC DEVICES:**

**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 Array logic,
* FPGA : Field Programmable Gate Arrays,
* CPLDs : Complex Programmable Logic Devices.

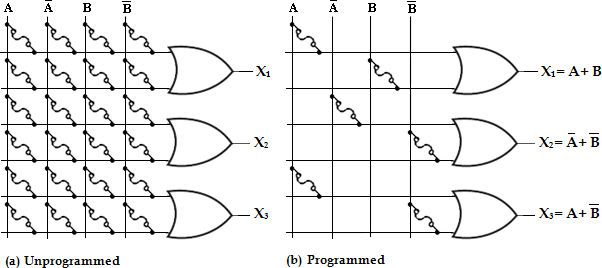
### Programmable Arrays:

All PLDs consists of programmable arrays. A programmable array is essentially 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.

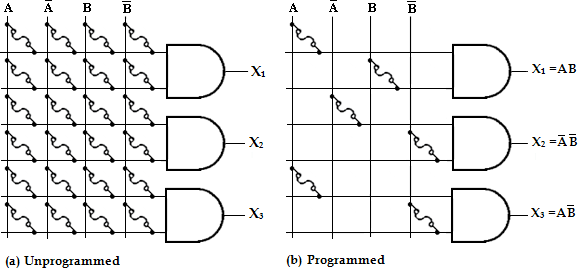
Another method of programming a PLD is the antifuse, which is the opposite of the fuse. Instead of a fusible link being broken or opened to program a variable, a normally open contact is shorted by ―melting‖ the antifuse material to form a connection.



**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 OR array, the AND array with fusible links or with anti fuses is one-time programmable.

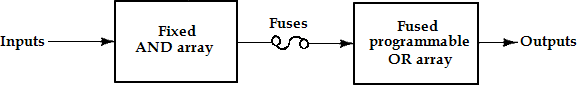


**An example of a basic programmable AND array**

# 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.

### Programmable Read-Only Memory (PROM):

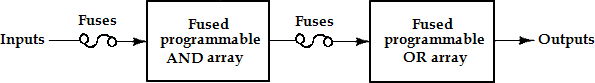
A PROM consists of a set of fixed (non-programmable) AND array constructed as a decoder and a programmable OR array. The

programmable OR gates implement the Boolean functions in sum of minterms.

* 1. **Programmable read- only memory (PROM)**

### Programmable Logic Array (PLA):

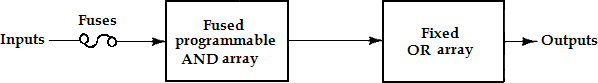
A PLA consists of a programmable AND array and a programmable OR array. The product terms in the AND array may be shared by any OR gate to provide the required sum of product implementation. The PLA is developed to overcome some of the limitations of the PROM. The PLA is also called an FPLA (Field Programmable Logic Array) because the user in the field, not the manufacturer, programs it.



**Programmable Logic Array (PLA)**

### Programmable Array Logic (PAL):

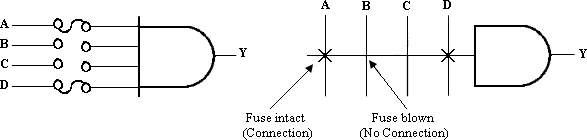
The basic PAL consists of a programmable AND array and a fixed OR array. The AND gates are programmed to provide the product terms for the Boolean functions, which are logically summed in each OR gate.

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.

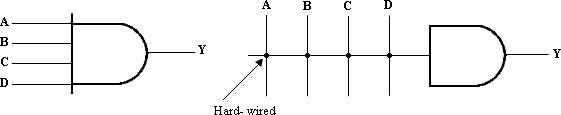
**Programmable Array Logic (PAL)**

### 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**.

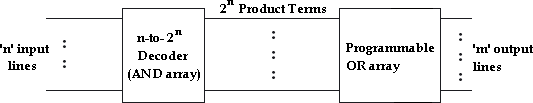






**PROGRAMMABLE ROM:**

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-2n 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 2n lines at the decoder output. The 2n outputs are connected to each of the ‗m‘ gates in the OR array via programmable fusible links.



**2n x m PROM**

**Implementation of Combinational Logic Circuit using PROM**

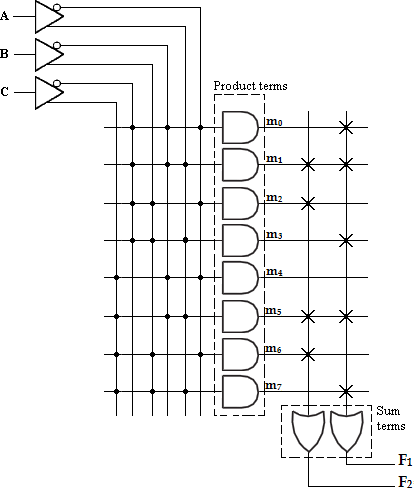
1. **Using PROM realize the following expression**

### F1 (A, B, C) = ∑m (0, 1, 3, 5, 7)

**F2 (A, B, C) = ∑m (1, 2, 5, 6)**

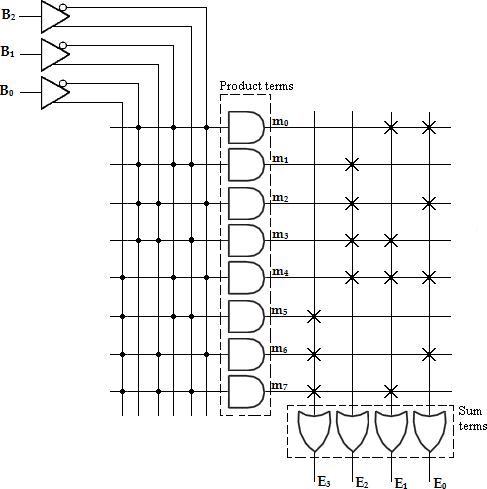
**Step1:** Truth table for the given function

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F1** | **F2** |
| 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 |

1. 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

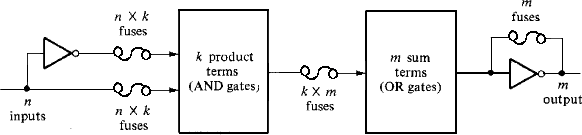
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **B2** | **B1** | **B0** | **E3** | **E2** | **E1** | **E0** |
| 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 | 0 |



**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.

# Implementation of Combinational Logic Circuit using PLA

### Implement the combinational circuit with a PLA having 3 inputs, 4 product terms and 2 outputs for the functions.

**F1 (A, B, C) = ∑m (0, 1, 2, 4)**

**F2 (A, B, C) = ∑m (0, 5, 6, 7)**

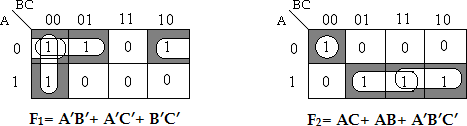
**Solution:**

**Step 1:** Truth table for the given functions

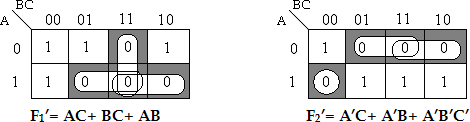
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F1** | **F2** |
| 0 | 0 | 0 | 1 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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



With this simplification, total number of product term is 6. But we require only 4 product terms. Therefore find out F1‘ and F2‘.



Now select, F1‘ and F2, the product terms are AC, AB, BC and A‘B‘C‘

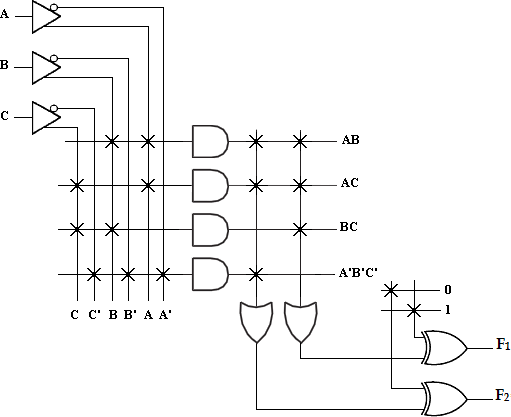
**Step 3:** PLA Program table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Product**  **term** | **Inputs** | | | **Outputs** | |
| **A** | **B** | **C** | **F1 (C)** | **F2 (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

STUDENTSFOCUS.COM



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 indicated by 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 to an 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)**.

### Implement the combinational circuit with a PLA having 3 inputs, 4 product terms and 2 outputs for the functions.

**F1 (A, B, C) = ∑m (3, 5, 6, 7)**

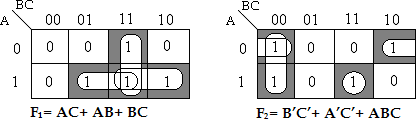
**F2 (A, B, C) = ∑m (0, 2, 4, 7)**

**Solution:**

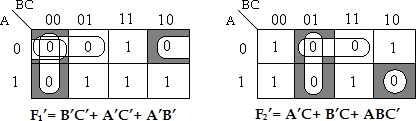
**Step 1:** Truth table for the given functions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F1** | **F2** |
| 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



With this simplification, total number of product term is 6. But we require only 4 product terms. Therefore find out F1‘ and F2‘.

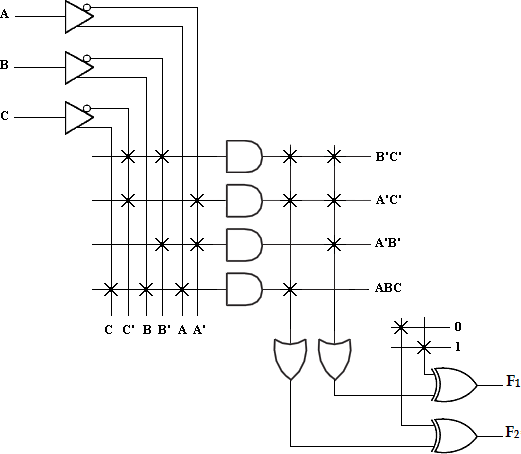


Now select, F1‘ and F2, the product terms are **B’C’, A’C’, A’B’ and ABC**. **Step 3:** PLA Program table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Product**  **term** | **Inputs** | | | **Outputs** | |
| **A** | **B** | **C** | **F1 (C)** | **F2 (T)** |
| B‘C‘  A‘C‘  A‘B‘ ABC | 1 | - | 0 | 0 | 1 | 1 |
| 2 | 0 | - | 0 | 1 | 1 |
| 3 | 0 | 0 | - | 1 | - |
| 4 | 1 | 1 | 1 | - | 1 |

**Step 4:** PLA Diagram

STUDENTSFOCUS.COM



1. Implement the following functions using PLA.

### F1 (A, B, C) = ∑m (1, 2, 4, 6)

**F2 (A, B, C) = ∑m (0, 1, 6, 7)**

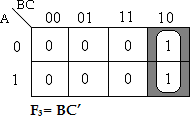
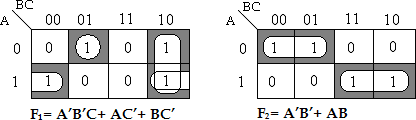
**F3 (A, B, C) = ∑m (2, 6)**

**Solution:**

**Step 1:** Truth table for the given functions

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F1** | **F2** | **F3** |
| 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

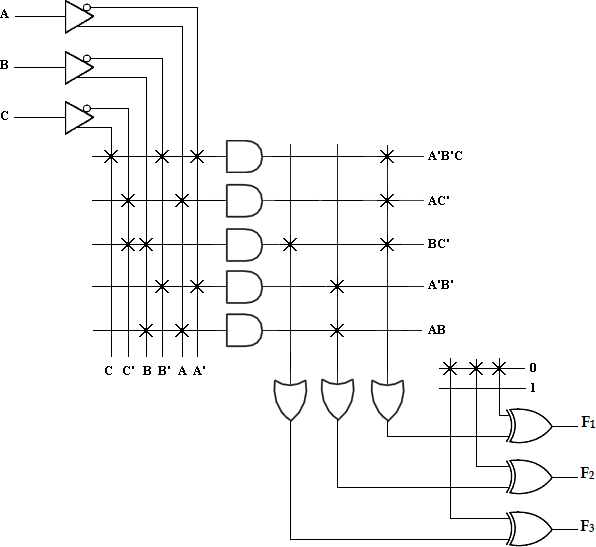


**Step 3:** PLA Program table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Product**  **term** | **Inputs** | | | **Outputs** | | |
| **A** | **B** | **C** | **F1 (T)** | **F2 (T)** | **F3 (T)** |
| A‘B‘C AC‘  BC‘ A‘B‘ AB | 1 | 0 | 0 | 1 | 1 | - | - |
| 2 | 1 | - | 0 | 1 | - | - |
| 3 | - | 1 | 0 | 1 | - | 1 |
| 4 | 0 | 0 | - | - | 1 | - |
| 5 | 1 | 1 | - | - | 1 | - |

**Step 4:** PLA Diagram

STUDENTSFOCUS.COM



### A combinational circuit is designed by the function

### F1 (A, B, C) = ∑m (3, 5, 7)

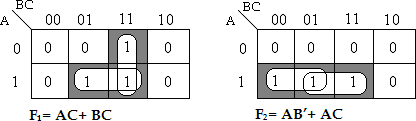
**F2 (A, B, C) = ∑m (4, 5, 7)**

**Solution:**

**Step 1:** Truth table for the given functions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F1** | **F2** |
| 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 |

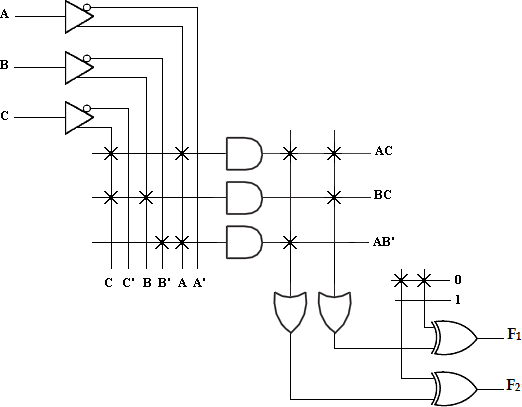
**Step 2:** K-map Simplification



**Step 3:** PLA Program table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Product**  **term** | **Inputs** | | | **Outputs** | |
| **A** | **B** | **C** | **F1 (C)** | **F2 (T)** |
| AC BC  AB‘ | 1 | 1 | - | 1 | 1 | 1 |
| 2 | - | 1 | 1 | 1 | - |
| 3 | 1 | 0 | - | - | 1 |

**Step 4:** PLA Diagram



### A combinational circuit is defined by the functions,

### F1 (A, B, C) = ∑m (1, 3, 5)

**F2 (A, B, C) = ∑m (5, 6, 7)**

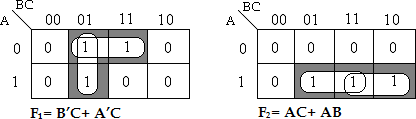
**Implement the circuit with a PLA having 3 inputs, 3 product terms and 2 outputs.**

**Solution:**

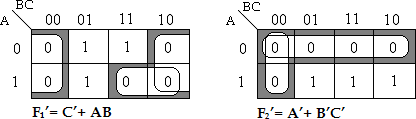
**Step 1:** Truth table for the given functions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F1** | **F2** |
| 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



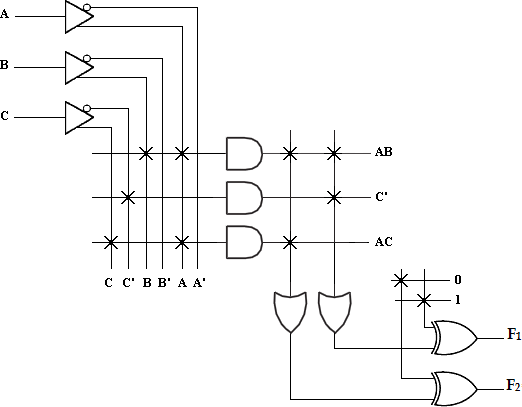
With this simplification, total number of product term is 5. But we require only 3 product terms. Therefore find out F1‘ and F2‘.



Now select, F1‘ and F2, the product terms are **AC, AB and C’**. **Step 3:** PLA Program table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Product**  **term** | **Inputs** | | | **Outputs** | |
| **A** | **B** | **C** | **F1 (C)** | **F2 (T)** |
| AB C‘ AC | 1 | 1 | 1 | - | 1 | 1 |
| 2 | - | - | 0 | 1 | - |
| 3 | 1 | - | 1 | - | 1 |

**Step 4:** PLA Diagram



### A combinational circuit is defined by the functions,

### F1 (A, B, C) = ∑m (0, 1, 3, 4)

**F2 (A, B, C) = ∑m (1, 2, 3, 4, 5)**

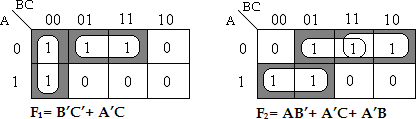
**Implement the circuit with a PLA having 3 inputs, 4 product terms and 2 outputs.**

**Solution:**

**Step 1:** Truth table for the given functions

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A** | **B** | **C** | **F1** | **F2** |
| 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

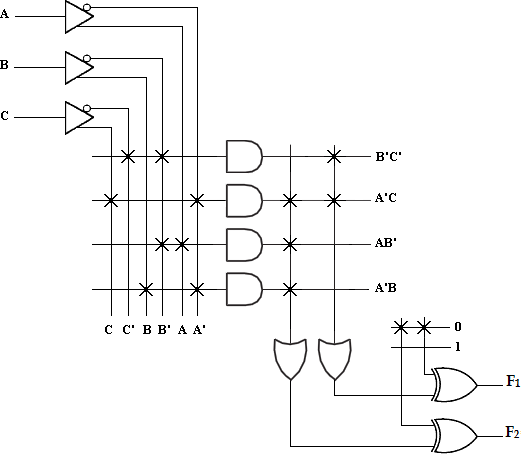


The product terms are **B’C’, A’C, AB’ and A’B**. **Step 3:** PLA Program table

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Product term** | **Inputs** | | | **Outputs** | |
| **A** | **B** | **C** | **F1 (T)** | **F2 (T)** |
| B‘C‘ A‘C  AB‘  A‘B | 1 | - | 0 | 0 | 1 | - |
| 2 | 0 | - | 1 | 1 | 1 |
| 3 | 1 | 0 | - | - | 1 |
| 4 | 0 | 1 | - | - | 1 |

**Step 4:** PLA Diagram

STUDENTSFOCUS.COM



### 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)**

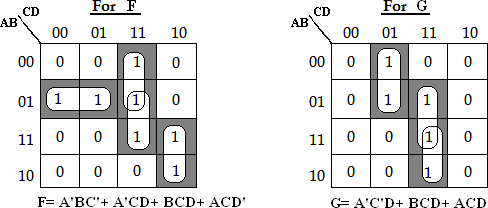
**Implement the circuit with a PLA having 4 inputs, 6 product terms and 2 outputs.**

**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



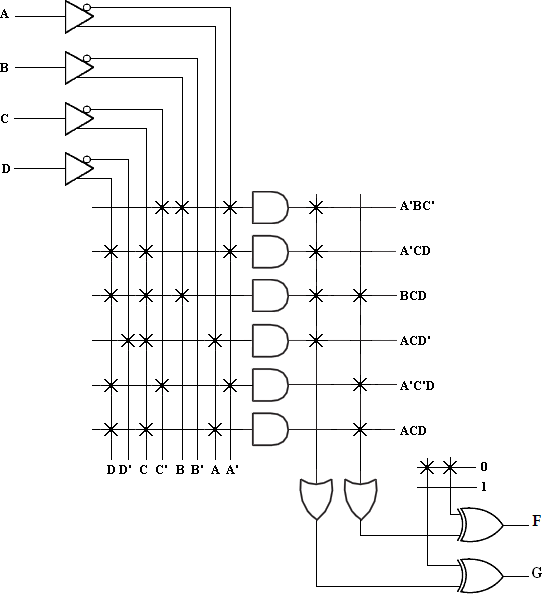
The product terms are A‘BC‘, A‘CD, BCD, ACD‘, A‘C‘D, ACD

**Step 3:** PLA Program table

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Product term** | **Inputs** | | | | **Outputs** | |
| **A** | **B** | **C** | **D** | **F (T)** | **G (T)** |
| A‘BC‘ A‘CD BCD ACD‘ A‘C‘D ACD | 1 | 0 | 1 | 0 | - | 1 | - |
| 2 | 0 | - | 1 | 1 | 1 | - |
| 3 | - | 1 | 1 | 1 | 1 | 1 |
| 4 | 1 | - | 1 | 0 | 1 | - |
| 5 | 0 | - | 0 | 1 | - | 1 |
| 6 | 1 | - | 1 | 1 | - | 1 |

**Step 4:** PLA Diagram

STUDENTSFOCUS.COM



### Design a BCD to Excess-3 code converter and implement using suitable PLA.

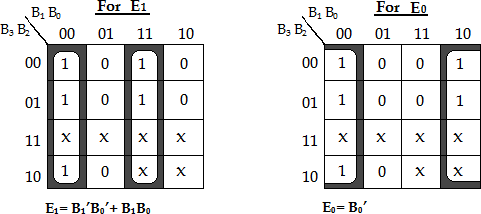
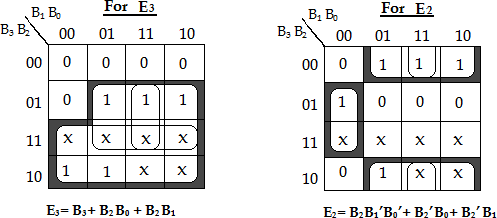
**Solution:**

**Step 1:** Truth table of BCD to Excess-3 converter is shown below,

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Decimal** | **BCD code** | | | | **Excess-3 code** | | | |
| **B3** | **B2** | **B1** | **B0** | **E3** | **E2** | **E1** | **E0** |
| 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

STUDENTSFOCUS.COM

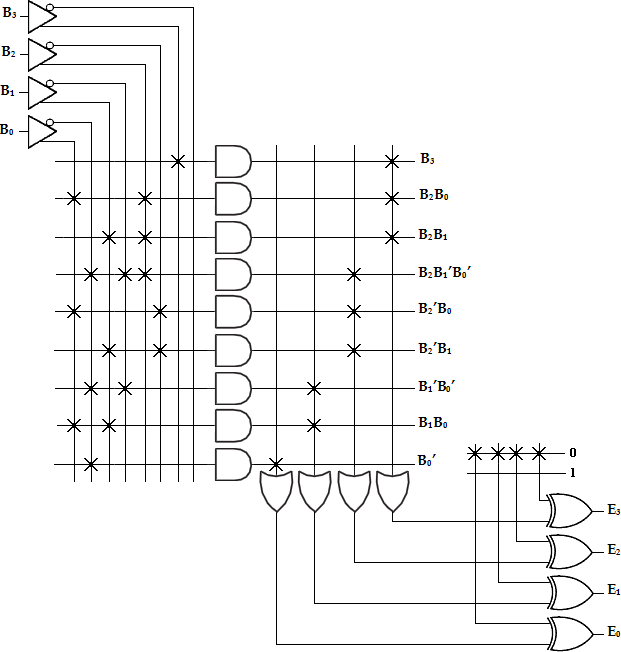


The product terms are **B3, B2B0, B2B1, B2B1’B0’, B2’B0, B2’B1, B1’B0’, B1B0, B0’**

**Step 3:** PLA Program table

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Product**  **terms** | **Inputs** | | | | **Outputs** | | | |
| **B3** | **B2** | **B1** | **B0** | **E3 (T)** | **E2 (T)** | **E1 (T)** | **E0 (T)** |
| B3 | 1 | 1 | - | - | - | 1 | - | - | - |
| B2B0 | 2 | - | 1 | - | 1 | 1 | - | - | - |
| B2B1 |
| 3 | - | 1 | 1 | - | 1 | - | - | - |
| B2B1‘B0‘ |
| 4 | - | 1 | 0 | 0 | - | 1 | - | - |
| B2‘B0 |
| 5 | - | 0 | - | 1 | - | 1 | - | - |
| B2‘B1 |
| 6 | - | 0 | 1 | - | - | 1 | - | - |
| B1‘B0‘ |
| 7 | - | - | 0 | 0 | - | - | 1 | - |
| B1B0 |
| 8 | - | - | 1 | 1 | - | - | 1 | - |
| B0‘ |
| 9 | - | - | - | 0 | - | - | - | 1 |

**Step 4:** PLA Diagram



### Comparison between PROM, PLA, and PAL:

STUDENTSFOCUS.COM

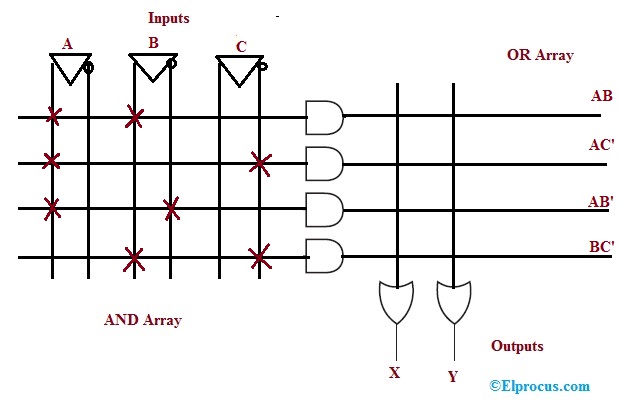
|  |  |  |  |
| --- | --- | --- | --- |
| **S.No** | **PROM** | **PLA** | **PAL** |
| 1 | AND array is fixed  and OR array is programmable | Both AND and OR  arrays are programmable | OR array is fixed and  AND array is programmable |
| 2 | Cheaper and simpler  to use | Costliest and complex | Cheaper 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 PAL |

**Implement the following Boolean expression with the help of programmable array logic (PAL)**

**X =AB + AC’**  
**Y= AB’ + BC’**

The above given two Boolean functions are in the form of **SOP (sum of products)**. The product terms present in the Boolean expressions are X & Y, and one product term that is AC’ is common in every equation. So, the total required logic gates for generating the above two equations is AND gates-4 OR programmable gates-2. The equivalent PAL logic diagram is shown below

* 

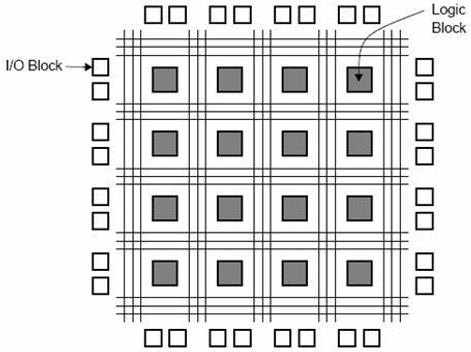
The AND gates which are programmable have the right of entry for normal as well as complemented variable inputs. In the above logic diagram, the available inputs for each AND gate are A, A’, B, B’, C, C’. So, in order to generate a single product term with every AND gate, the program is required.  
All the product terms are obtainable at the inputs of an each OR gate. Here, the programmable connections on the logic gate can be denoted with the symbol ‘X’. Here, the OR gate inputs are fixed. Thus, the required product terms are associated with each OR gate inputs. As a result, these gates will generate particular Boolean equations. The **‘.’** The symbol represents permanent connections.

## **FPGA – Introduction**

The full form of **FPGA** is “**Field Programmable Gate Array**”. It contains ten thousand to more than a million logic gates with programmable interconnection. Programmable interconnections are available for users or designers to perform given functions easily. A typical model FPGA chip is shown in the given figure. There are I/O blocks, which are designed and numbered according to function. For each module of logic level composition, there are **CLB’s (Configurable Logic Blocks)**.

CLB performs the logic operation given to the module. The inter connection between CLB and I/O blocks are made with the help of horizontal routing channels, vertical routing channels and PSM (Programmable Multiplexers).

The number of CLB it contains only decides the complexity of FPGA. The functionality of CLB’s and PSM are designed by VHDL or any other hardware descriptive language. After programming, CLB and PSM are placed on chip and connected with each other with routing channels.



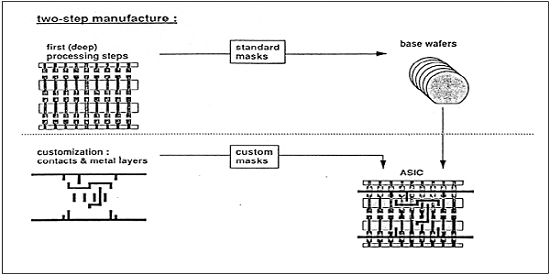
### Advantages

* It requires very small time; starting from design process to functional chip.
* No physical manufacturing steps are involved in it.
* The only disadvantage is, it is costly than other styles.

## **Gate Array Design**

The **gate array (GA)** ranks second after the FPGA, in terms of fast prototyping capability. While user programming is important to the design implementation of the FPGA chip, metal mask design and processing is used for GA. Gate array implementation requires a two-step manufacturing process.

The first phase results in an array of uncommitted transistors on each GA chip. These uncommitted chips can be stored for later customization, which is completed by defining the metal interconnects between the transistors of the array. The patterning of metallic interconnects is done at the end of the chip fabrication process, so that the turn-around time can still be short, a few days to a few weeks. The figure given below shows the basic processing steps for gate array implementation.



Typical gate array platforms use dedicated areas called channels, for inter-cell routing between rows or columns of MOS transistors. They simplify the interconnections. Interconnection patterns that perform basic logic gates are stored in a library, which can then be used to customize rows of uncommitted transistors according to the netlist.

In most of the modern GAs, multiple metal layers are used for channel routing. With the use of multiple interconnected layers, the routing can be achieved over the active cell areas; so that the routing channels can be removed as in Sea-of-Gates (SOG) chips. Here, the entire chip surface is covered with uncommitted nMOS and pMOS transistors. The neighboring transistors can be customized using a metal mask to form basic logic gates.

For inter cell routing, some of the uncommitted transistors must be sacrificed. This design style results in more flexibility for interconnections and usually in a higher density. GA chip utilization factor is measured by the used chip area divided by the total chip area. It is higher than that of the FPGA and so is the chip speed.

## **Standard Cell Based Design**

A standard cell based design requires development of a full custom mask set. The standard cell is also known as the polycell. In this approach, all of the commonly used logic cells are developed, characterized and stored in a standard cell library.

A library may contain a few hundred cells including inverters, NAND gates, NOR gates, complex AOI, OAI gates, D-latches and Flip-flops. Each gate type can be implemented in several versions to provide adequate driving capability for different fan-outs. The inverter gate can have standard size, double size, and quadruple size so that the chip designer can select the proper size to obtain high circuit speed and layout density.

Each cell is characterized according to several different characterization categories, such as,

* Delay time versus load capacitance
* Circuit simulation model
* Timing simulation model
* Fault simulation model
* Cell data for place-and-route
* Mask data

For automated placement of the cells and routing, each cell layout is designed with a fixed height, so that a number of cells can be bounded side-by-side to form rows. The power and ground rails run parallel to the upper and lower boundaries of the cell. So that, neighboring cells share a common power bus and a common ground bus. The figure shown below is a floorplan for standard-cell based design.



## **Full Custom Design**

In a full-custom design, the entire mask design is made new, without the use of any library. The development cost of this design style is rising. Thus, the concept of design reuse is becoming famous to reduce design cycle time and development cost.

The hardest full custom design can be the design of a memory cell, be it static or dynamic. For logic chip design, a good negotiation can be obtained using a combination of different design styles on the same chip, i.e. standard cells, data-path cells, and **programmable logic arrays (PLAs)**.

Practically, the designer does the full custom layout, i.e. the geometry, orientation, and placement of every transistor. The design productivity is usually very low; typically a few tens of transistors per day, per designer. In digital CMOS VLSI, full-custom design is hardly used due to the high labor cost. These design styles include the design of high-volume products such as memory chips, high-performance microprocessors and FPGA.

**Outcomes:**

i. Able to understand the concept of memory unit and PLD.

ii. Design and Implement different types of PLD.

# UNIT V

# SYNCHRONOUS AND ASYNCHRONOUS

# SEQUENTIAL CIRCUITS

# Pre requisition:

# i. Basic Knowledge in sequential logic.

# ii. Basic Knowledge in PLD.

# iii. Basic Knowledge in minimization techniques.

**Algorithmic State Machine**

Every digital system can be partitioned into two parts. Those are data path digital circuits and control circuits. Data path circuits perform the functions such as storing of binary information data and transfer of data from one system to the other system. Whereas, control circuits determine the flow of operations of digital circuits.

It is difficult to describe the behavior of large state machines using state diagrams. To overcome this difficulty, Algorithmic State Machine ASM charts can be used. **ASM charts** are similar to flow charts. They are used to represent the flow of tasks to be performed by data path circuits and control circuits.

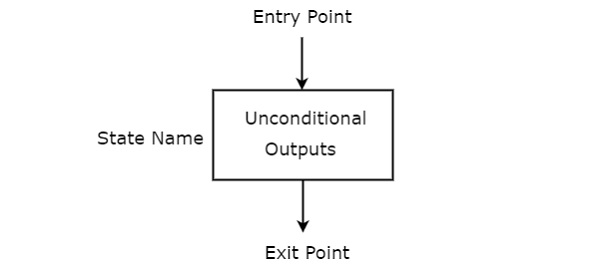
## **Basic Components of ASM charts**

Following are the three basic components of ASM charts.

* State box
* Decision box
* Conditional output box

### State box

State box is represented in rectangular shape. Each state box represents one state of the sequential circuit. The **symbol** of state box is shown in the following figure.

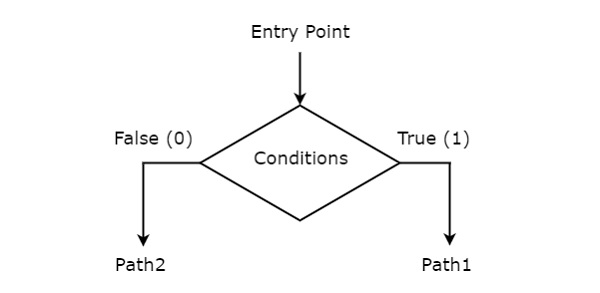


It is having one entry point and one exit point. Name of the state is placed to the left of state box. The unconditional outputs corresponding to that state can be placed inside state box. **Moore** state machine outputs can also be placed inside state box.

### Decision box

Decision box is represented in diamond shape. The **symbol** of decision box is shown in the following

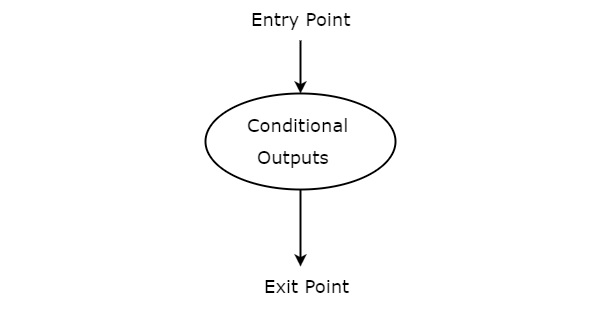
Figure.



It is having one entry point and two exit paths. The inputs or Boolean expressions can be placed inside the decision box, which are to be checked whether they are true or false. If the condition is true, then it will prefer path1. Otherwise, it will prefer path2.

### Conditional output box

Conditional output box is represented in oval shape. The **symbol** of conditional output box is shown in the following figure.



It is also having one entry point and one exit point similar to state box. The conditional outputs can be placed inside state box. In general, Mealy state machine outputs are represented inside conditional output box. So, based on the requirement, we can use the above components properly for drawing ASM charts.

**Synchronous sequential circuit**

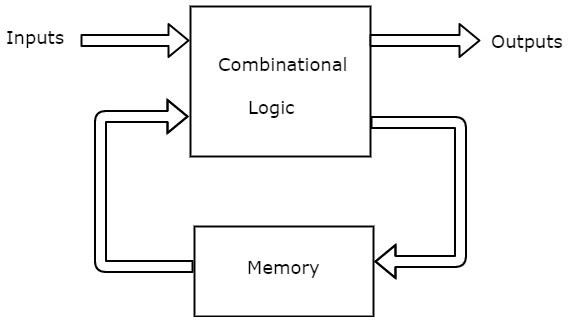
A synchronous sequential circuit is also called as **Finite State Machine** (FSM), if it has finite number of states. There are two types of FSMs.

* + Mealy State Machine
  + Moore State Machine

Now, let us discuss about these two state machines one by one.

## Mealy State Machine

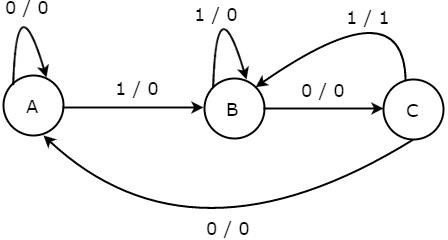
A Finite State Machine is said to be Mealy state machine, if outputs depend on both present inputs & present states. The **block diagram** of Mealy state machine is shown in the following figure.



As shown in figure, there are two parts present in Mealy state machine. Those are combinational logic and memory. Memory is useful to provide some or part of previous outputs **(present states)** as inputs of combinational logic.

So, based on the present inputs and present states, the Mealy state machine produces outputs. Therefore, the outputs will be valid only at positive (or negative) transition of the clock signal.

The **state diagram** of Mealy state machine is shown in the following figure.

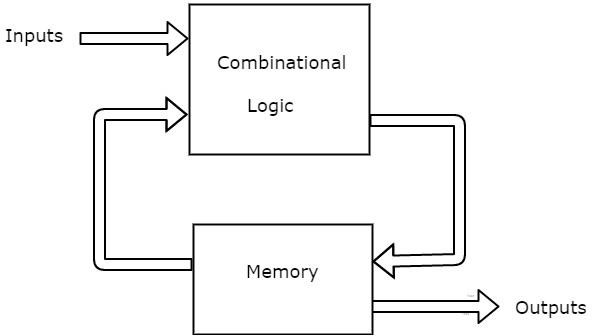


In the above figure, there are three states, namely A, B & C. These states are labeled inside the circles & each circle corresponds to one state. Transitions between these states are represented with directed lines. Here, 0 / 0, 1 / 0 & 1 / 1 denotes **input / output**. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Mealy state machine is less than or equal to the number of states required in Moore state machine. There is an equivalent Moore state machine for each Mealy state machine.

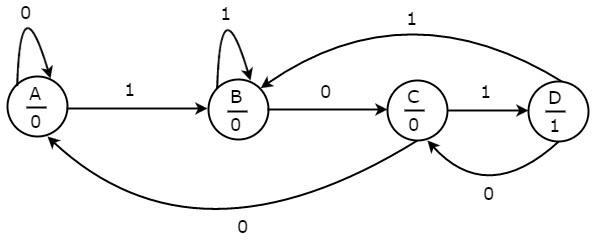
## Moore State Machine

A Finite State Machine is said to be Moore state machine, if outputs depend only on present states. The **block diagram** of Moore state machine is shown in the following figure.



As shown in figure, there are two parts present in Moore state machine. Those are combinational logic and memory. In this case, the present inputs and present states determine the next states. So, based on next states, Moore state machine produces the outputs. Therefore, the outputs will be valid only after transition of the state.

The **state diagram** of Moore state machine is shown in the following figure.



In the above figure, there are four states, namely A, B, C & D. These states and the respective outputs are labelled inside the circles. Here, only the input value is labeled on each transition. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Moore state machine is more than or equal to the number of states required in Mealy state machine. There is an equivalent Mealy state machine for each Moore state machine. So, based on the requirement we can use one of them.

## Comparison between the Moore machine and mealy machine:

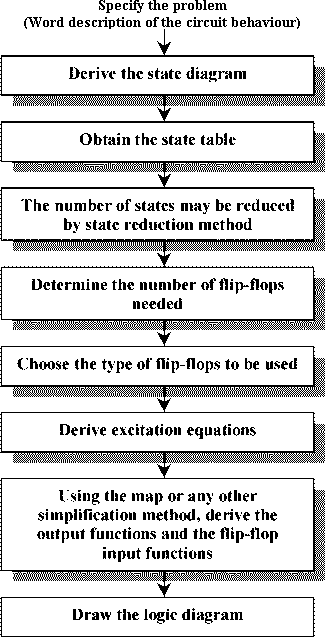
|  |  |
| --- | --- |
| **Moore machine** | **mealy machine** |
| 1. its output is a function of present | 1. its output is a function of present state |
| state only Z(t)= g{S(t)} | as well as present input Z(t)=g{S(t),X(t)} |
| 2. input changes do not affect the | 2. input changes may affect the output of |
| output | the circuit |
| 3. it requires more number of states | 3. it requires less number of states for |
| for implementing same function | implementing same function |

## Design of Synchronous Sequential Circuits:

The design of a synchronous sequential circuit starts from a set of specifications and culminates in a logic diagram or a list of Boolean functions from which a logic diagram can be obtained. In contrast to a combinational logic, which is fully specified by a truth table, a sequential circuit requires a state table for its specification. The first step in the design of sequential circuits is to obtain a state table or an equivalence representation, such as a state diagram.

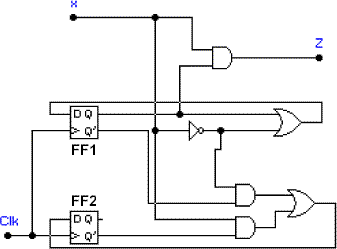
A synchronous sequential circuit is made up of flip-flops and combinational gates. The design of the circuit consists of choosing the flip-flops and then finding the combinational structure which, together with the flip-flops, produces a circuit that fulfils the required specifications. The number of flip-flops is determined from the number of states needed in the circuit.

The recommended steps for the design of sequential circuits are set out below.



Problem:

Consider a sequential circuit shown below Figure . It has one input x, one output Z and two state variables Q1Q2 (thus having four possible present states 00, 01, 10, 11).



The behaviour of the circuit is determined by the following Boolean expressions:

Z = x\*Q1 D1 = x' + Q1

D2 = x\*Q2' + x'\*Q1'

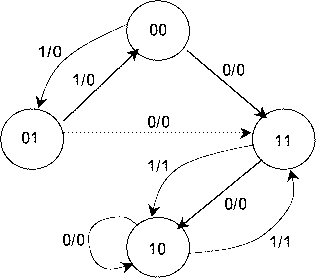
These equations can be used to form the state table. Suppose the present state (i.e. Q1Q2) = 00 and input x = 0. Under these conditions, we get Z = 0, D1 = 1, and D2 = 1. Thus the next state of the circuit D1D2 = 11, and this will be the present state after the clock pulse has been applied. The output of the circuit corresponding to the present state Q1Q2 = 00 and x = 1 is Z

= 0. This data is entered into the state table as shown in Table 2.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Present State  Q1Q2 | Next | State | Output  x = 0 x = 1 | |
| x = 0 | x = 1 |
| 0 0 | 1 1 | 0 1 | 0 | 0 |
| 0 1 | 1 1 | 0 0 | 0 | 0 |
| 1 0 | 1 0 | 1 1 | 0 | 1 |
| 1 1 | 1 0 | 1 0 | 0 | 1 |

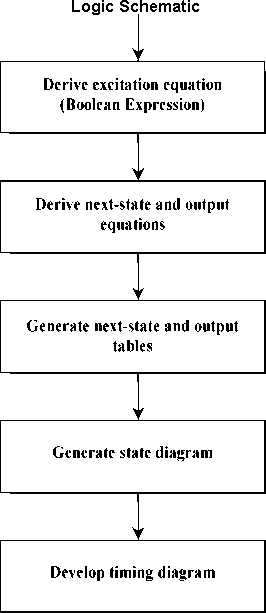
Table 2. State table for the sequential circuit in Figure 4.

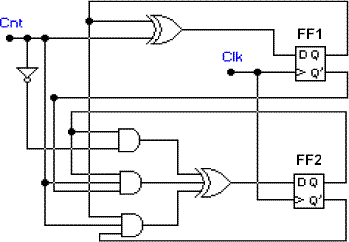
The state diagram for the sequential circuit in Figure 4 is shown in Figure 5.



Analysis of synchronous Sequential Circuits

The behaviour of a sequential circuit is determined from the inputs, the outputs and the states of its flip-flops. Both the output and the next state are a function of the inputs and the present state.



Derive the state table and state diagram for the sequential circuit shown in below Figure.

SOLUTION:

STEP 1: First we derive the Boolean expressions for the inputs of each flip-flops in the schematic, in terms of external input Cnt and the flip-flop outputs Q1 and Q0. Since there are two D flip-flops in this example, we derive two expressions for D1 and D0:

D0 = Cnt Q0 = Cnt'\*Q0 + Cnt\*Q0'

D1 = Cnt'\*Q1 + Cnt\*Q1'\*Q0 + Cnt\*Q1\*Q0'

These Boolean expressions are called excitation equations since they represent the inputs to the flip-flops of the sequential circuit in the next clock cycle.

STEP 2: Derive the next-state equations by converting these excitation equations into flip- flop characteristic equations. In the case of D flip-flops, Q(next) = D. Therefore the next state equal the excitation equations.

Q0(next) = D0 = Cnt'\*Q0 + Cnt\*Q0'

Q1(next) = D1 = Cnt'\*Q1 + Cnt\*Q1'\*Q0 + Cnt\*Q1\*Q0'

STEP 3: Now convert these next-state equations into tabular form called the next-state table.

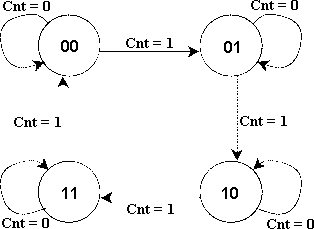
|  |  |
| --- | --- |
| Present State | Next State |
| Q1Q0 | Cnt = 0 Cnt = 1 |
| 0 0 | 0 0 0 1 |
| 0 1 | 0 1 1 0 |
| 1 0 | 1 0 1 1 |
| 1 1 | 1 1 0 0 |

Each row is corresponding to a state of the sequential circuit and each column represents one set of input values. Since we have two flip-flops, the number of possible states is four - that is, Q1Q0 can be equal to 00, 01, 10, or 11. These are present states as shown in the table.

For the next state part of the table, each entry defines the value of the sequential circuit in the next clock cycle after the rising edge of the Clk. Since this value depends on the present state and the value of the input signals, the next state table will contain one column for each assignment of binary values to the input signals. In this example, since there is only one input signal, Cnt, the next-state table shown has only two columns, corresponding to Cnt = 0 and Cnt = 1.

Note that each entry in the next-state table indicates the values of the flip-flops in the next state if their value in the present state is in the row header and the input values in the column header.

Each of these next-state values has been computed from the next-state equations in STEP 2.

STEP 4: The state diagram is generated directly from the next-state table, shown in below Figure

Figure

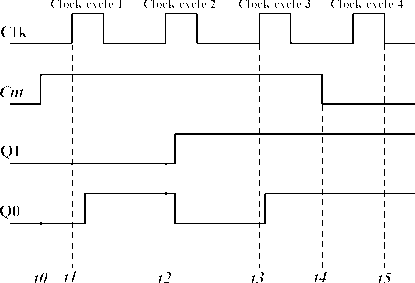
State diagram

Each arc is labelled with the values of the input signals that cause the transition from the present state (the source of the arc) to the next state (the destination of the arc).

In general, the number of states in a next-state table or a state diagram will equal 2m , where m is the number of flip-flops. Similarly, the number of arcs will equal 2m x 2k , where k is the number of binary input signals. Therefore, in the state diagram, there must be four states and eight transitions. Following these transition arcs, we can see that as long as Cnt = 1, the sequential circuit goes through the states in the following sequence: 0, 1, 2, 3, 0, 1, 2, On

the other hand, when Cnt = 0, the circuit stays in its present state until Cnt changes to 1, at which the counting continues.

Since this sequence is characteristic of modulo-4 counting, we can conclude that the sequential circuit in Figure 7 is a modulo-4 counter with one control signal, Cnt, which enables counting when Cnt = 1 and disables it when Cnt = 0.

Below, we show a timing diagram, representing four clock cycles, which enables us to observe the behaviour of the counter in greater detail.

In this timing diagram we have assumed that Cnt is asserted in clock cycle 0 at t0 and is disasserted in clock cycle 3 at time t4. We have also assumed that the counter is in state Q1Q0 = 00 in the clock cycle 0. Note that on the clock's rising edge, at t1, the counter will go to state Q1Q0 = 01 with a slight propagation delay; in cycle 2, after t2, to Q1Q0 = 10; and in cycle 3, after t3 to Q1Q0 = 11. Since Cnt becomes 0 at t4, we know that the counter will stay in state Q1Q0 = 11 in the next clock cycle. To see the timing behaviour of the circuit click on this image .

In above Example we demonstrated the analysis of a sequential circuit that has no outputs by developing a next-state table and state diagram which describes only the states and the transitions from one state to the next. In the next example we complicate our analysis by adding output signals, which means that we have to upgrade the next-state table and the state diagram to identify the value of output signals in each state.

### ASYNCHRONOUS SEQUENTIAL CIRCUIT

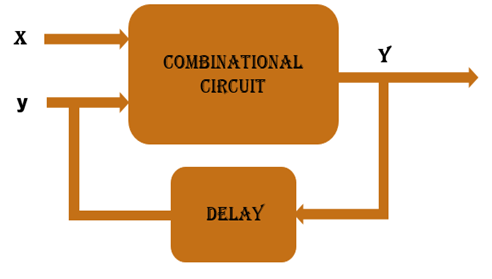
1.**Agenda**

* Introduction
* Block Diagram
* Modes of asynchronous sequential circuit
* Design Flow
* Example Explanation
* Applications
* Advantages
* Disadvantages

2. **Introduction**

* Asynchronous sequential circuit works without Clock.
* Asynchronous sequential circuit works based on memory concept
* Feedback is present as delay line, and Delay of feedback is not predictable so application of asynchronous sequential circuit is limited.
* Changes in inputs cause changes in output (State changes)
* Asynchronous sequential circuit design is more complicated than synchronous sequential circuit design
* The memory of the asynchronous sequential circuit may include flip-flops or time-delay devices.

3. **Block diagram**



* Here changes in inputs cause changes in output (State changes)
* Here in Block Diagram, Clock is not used but output is directly connected to inputs through some calculated delay (Feedback)
* when Inputs changes then output changes and like this stability is achieved
* When stability is achieved that state is called stable state. Rests of states are unstable states.

4. **Modes of asynchronous sequential circuit**

* Fundamental Mode
  + Only One input can be change at a time after stable state
  + This mode is widely used for design.
* Pulse mode: - More than one input can be change at a time after stable state.

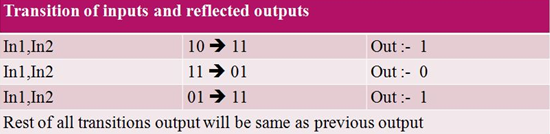
5. **Design Flow**

* Nomenclature
  + (a),0 :- Here a is a stable state and 0 is output
  + a  :-  Here a is unstable state and output remain same as previous
* Write word statement from problem statement
* Derive primitive flow table
* Design merger diagram
* Adjacent diagram
* Derive synthesis equation

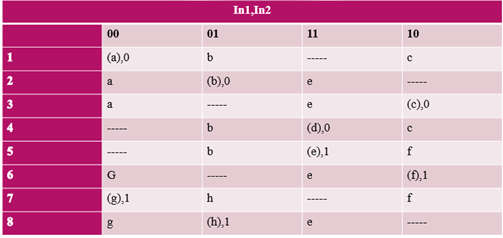
For more details refer example…..

6. **Example**

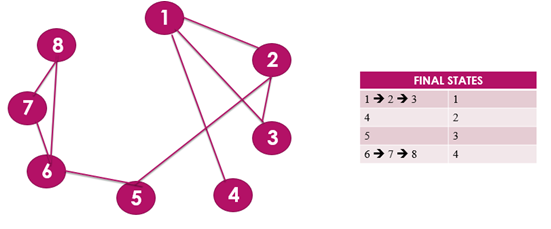
* Problem Statement
  + Apply fundamental Mode
  + **2 Bit inputs “In1” and “In2”**
  + **1 bit output “out”**
  + **“out=1” when “In1” changes from “0” to “1” and “In2=1”**
  + **“out=1” when “In2” changes from “0” to “1” and “In1=1”**
  + **“out=0” when “In1” changes from “1” to “0” and “In2=1”**
* **​**Problem statement in In/Out form
  + Convert problem statement into input and output form it will helpful for designing steps.
  + Complex design includes more signals in that case this step is very useful for designing primitive flow table.



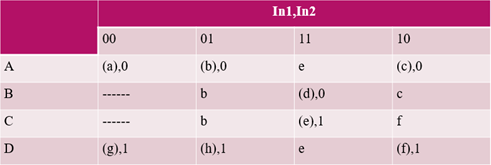
* Primitive flow table
  + Primitive Flow table contains stable and unstable states of asynchronous sequential circuit
  + All possible combination and its output is listed in the primitive flow table
  + First start with stable state "a". So for "00" column state will be (a), 0.
  + Fundamental mode is applied here so "00" to "11" transition is not possible so column of "11" will be blank.
  + Now transition from "00" to "01" is possible and output will be "0" and state will be unstable because in one row only one stable state is possible.
  + Now transition from "00" to "10" is possible and output will be "0" and state will be unstable. Similarly for every transition same rule is applied so primitive table will be like this.



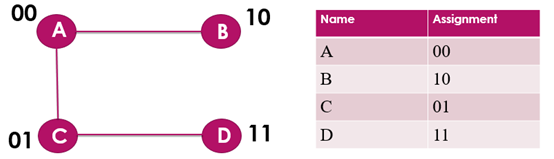
* Merger diagram
  + Merger diagram will merge different row by following some rules.
  + Stable state and unstable state will be merge and result in to stable state but same state must be there.
  + Unstable state and blank(“-“) can be merge into unstable state
  + Stable state and blank (“-“) can be merge into stable state.
  + Two rows can be merge when all column are follow merge rules.
  + In diagram bounded nodes can be merge into one node here bounded nodes are (1,2,3) and (6,7,8)
  + For (1,2,3) one node is finalize and it is (1)
  + For (6,7,8) one node is finalize and it is (4)
  + 4 and 5 is remaining same as primitive table.



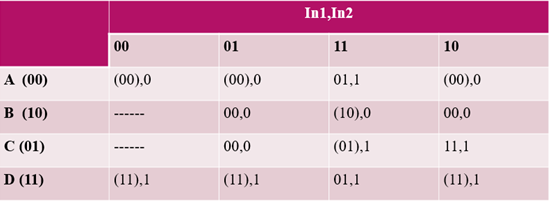
* Merger state table



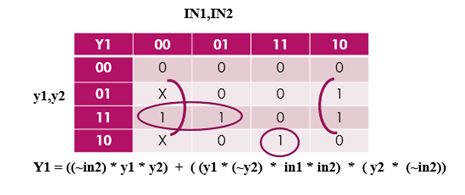
* Adjacent diagram
  + Adjacent diagram is useful to assign value to the variables (Here A, B, C and D).
  + If small circuit is there then this step can be skipped and in this case random value will be assigned to the variables.
  + There are some rules to assign value to the variables.
  + Here in the row of "A" and "B" column of "01" "b" stable and unstable state is there so in adjacent diagram "A" and "B" is connected together similarly "A" and "C" is connected together and similarly "C" and "D" is connected together.
  + Now to assign value to the variable connected variable should be in Grey coding means difference in bits should not be greater than 1. So, Here "A" and "B" is connected so A=00 B=10 so difference of bit is 1 similarly C=01 and D=11.

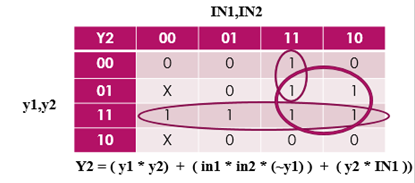


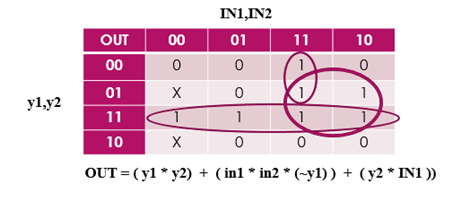
* Revision of merger diagram
  + Merger diagram is revised by replacing variables to its values will be like this



* Synthesis equation for inputs and outputs
  + first input synthesis equation is designed by getting First bit from revised merger diagram
  + second input synthesis equation is designed by getting second bit from revised merger diagram and similarly
  + Output synthesis equation is designed by getting output bit from revised merger diagram.







7. **Application**

* Asynchronous sequential circuits are useful when circuit system must respond quickly without waiting for clock.
* Asynchronous sequential circuit is important for small circuit which behaves independently and contain few components

8. **Advantages**

* Robust handling of metastability and higher performance compare to synchronous sequential circuit.
* Faster than Synchronous sequential circuit
* Lower power consumption
* Clock driver can be removed in this case because Clock is not used here so power consumption of clock drivers and controller can be avoided.
* few assumptions are needed in manufacture process
* system speed adapts changes in environment and voltage levels
* Designing of power distribution network is easy here because here leakage current will be less compare to synchronous sequential circuit

9. **Disadvantages**

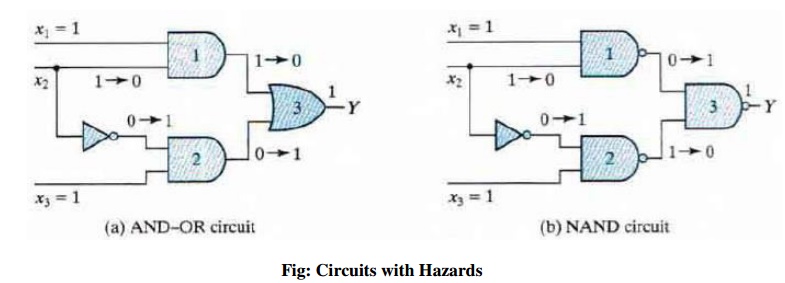
* When some encoding is performed then asynchronous circuit requires more area then synchronous circuit and because of the same power consumption may increase
* Area of circuit is increased. No. of Transistors may be double here because of addition of completion detection circuit and design for test circuit
* synchronous sequential circuits are easier to test and debug compare to asynchronous sequential circuit
* Performance of asynchronous sequential circuits may be reducing in architecture which includes complex data paths and feedbacks.
* Race conditions are generated internally and cannot be handling by outside.

**HAZARDS:**

**Hazards**are unwanted switching transients that may appear at the output of a circuit because differentpaths exhibit different propagation delays. Hazards occur in combinational circuits, where they may cause a temporary false output value. When they occur in asynchronous sequential circuits hazards may result in a transition to a wrong stable state.

**Hazards In Combinational Circuits**

A hazard is a condition in which a change in a single variable produces a momentary change in output when no change in output should occur.



Assume that all three inputs are initially equal to 1. This causes the output of gate 1 10 be 1, that of gate 2 to be 0 and that of the circuit to be 1. Now consider a change in x 2 from 1 to 0. Then the output of gate 1 changes to 0 and that of gate 2 changes to 1, leaving the output at 1. However, 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 gale 2 changes to 1.

The two circuits shown in Fig implement the Boolean function in sum-of-products form:

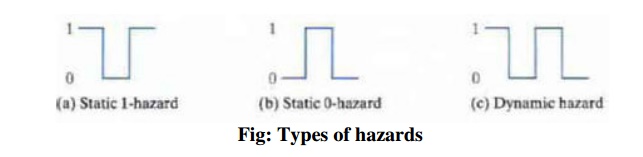
http://img.brainkart.com/extra/qUOxMCn.jpg

This type of implementation may cause the output to go to 0 when it should remain a 1. If however, the circuit is implemented instead in product-of-sums form namely,

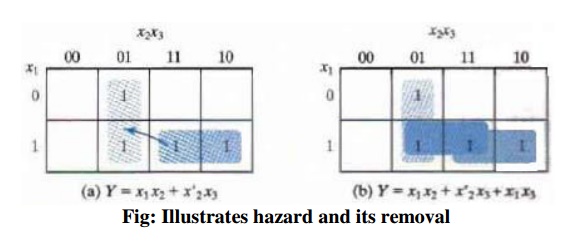
http://img.brainkart.com/extra/DXjwklJ.jpg

then the output may momentarily go to 1 when it should remain 0. The first case is referred to as **static** **1-hazard**and the second case as**static 0-hazard**.

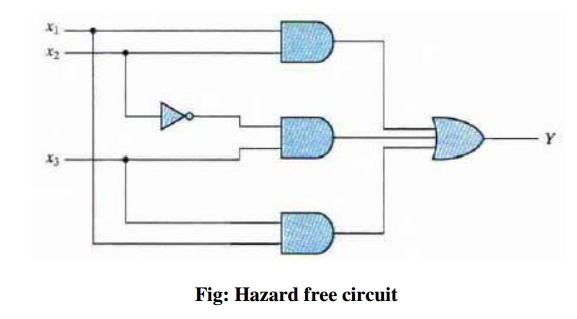
A third type of hazard, known as **dynamic hazard**, causes the output to change three or more times when it should change from 1 to 0 or from 0 to 1.



The change in x2 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 minterm.

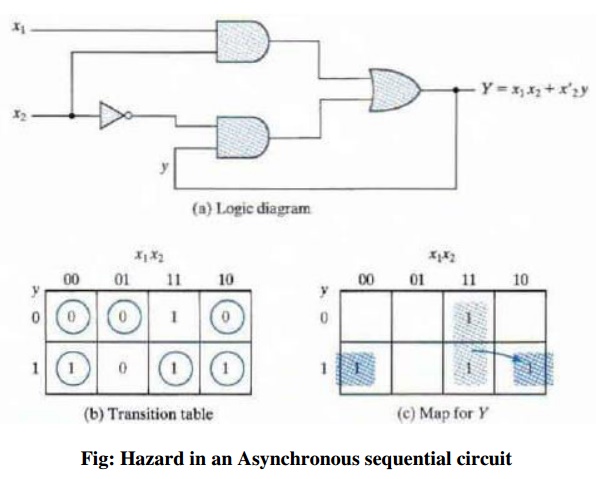


Minterm 111 is covered by the product term implemented in gate 1 and minterm 101 is covered by the product term implemented in gate 2. The remedy for eliminating a hazard is to enclose the two minterms with another product term that overlaps both groupings. The hazard-free circuit obtained by such a configuration is shown in figure below. The extra gate in the circuit generates the product term x1x3. In general, hazard s in combinational circuits can be removed by cove ring 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.



**Hazards in Sequential Circuits**

In normal combinational-circuit design associated with synchronous sequential circuits, hazards are of no concern, since momentary erroneous signals are not generally troublesome. However, if a momentary incorrect signal is fed back in an asynchronous sequential circuit, it may cause the circuit to go to the wrong stable state.



If the circuit is in total stable state yx1x2 =111 and input x2 changes from I to 0, the next total stable state should be 110. However, because of the hazard, output Y may go to 0 momentarily. If this false signal feeds back into gate 2 before the output of the inverter goes to 1, the output of gate 2 will remain at 0 and the circuit will switch to the incorrect total stable state 010. This malfunction can be eliminated by adding an extra gate.

**Essential Hazards**

Another type of hazard that may occur in asynchronous sequential circuits is called an **essential hazard***.* This type of 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 cannot be corrected by adding redundant gates as in static hazards. The problem that they impose can be corrected by adjusting the amount of delay 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 compare d with delays of other signals that originate from the input terminals.

**VERILOG HDL:**

Verilog is a HARDWARE DESCRIPTION LANGUAGE (HDL). It is a language used for describing a digital system like a network switch or a microprocessor or a memory or a flip−flop. It means, by using a HDL we can describe any digital hardware at any level. Designs, which are described in HDL are independent of technology, very easy for designing and debugging, and are normally more useful than schematics, particularly for large circuits.

Verilog supports a design at many levels of abstraction. The major three are −

* Behavioral level
* Register-transfer level
* Gate level

## **Behavioral level**

This level describes a system by concurrent algorithms (Behavioural). Every algorithm is sequential, which means it consists of a set of instructions that are executed one by one. Functions, tasks and blocks are the main elements. There is no regard to the structural realization of the design.

## **Register−Transfer Level**

Designs using the Register−Transfer Level specify the characteristics of a circuit using operations and the transfer of data between the registers. Modern definition of an RTL code is "Any code that is synthesizable is called RTL code".

## **Gate Level**

Within the logical level, the characteristics of a system are described by logical links and their timing properties. All signals are discrete signals. They can only have definite logical values (`0', `1', `X', `Z`). The usable operations are predefined logic primitives (basic gates). Gate level modelling may not be a right idea for logic design. Gate level code is generated using tools like synthesis tools and his netlist is used for gate level simulation and for backend.

## **Lexical Tokens**

Verilog language source text files are a stream of lexical tokens. A token consists of one or more characters, and each single character is in exactly one token.

The basic lexical tokens used by the Verilog HDL are similar to those in C Programming Language. Verilog is case sensitive. All the key words are in lower case.

### White Space

White spaces can contain characters for spaces, tabs, new-lines and form feeds. These characters are ignored except when they serve to separate tokens.

White space characters are Blank space, Tabs, Carriage returns, New line, and Form feeds.

### Comments

There are two forms to represent the comments

* 1) Single line comments begin with the token // and end with carriage return.

Ex.: //this is single line syntax

* 2) Multiline comments begins with the token /\* and end with token \*/

Ex.: /\* this is multiline Syntax\*/

### Numbers

You can specify a number in binary, octal, decimal or hexadecimal format. Negative numbers are represented in 2’s compliment numbers. Verilog allows integers, real numbers and signed & unsigned numbers.

The syntax is given by − <size> <radix> <value>

Size or unsized number can be defined in <Size> and <radix> defines whether it is binary, octal, hexadecimal or decimal.

### Identifiers

Identifier is the name used to define the object, such as a function, module or register. Identifiers should begin with an alphabetical characters or underscore characters. Ex. A\_Z, a\_z,\_

Identifiers are a combination of alphabetic, numeric, underscore and $ characters. They can be up to 1024 characters long.

### Operators

Operators are special characters used to put conditions or to operate the variables. There are one, two and sometimes three characters used to perform operations on variables.

Ex. >, +, ~, &! =.

### Verilog Keywords

Words that have special meaning in Verilog are called the Verilog keywords. For example, assign, case, while, wire, reg, and, or, nand, and module. They should not be used as identifiers. Verilog keywords also include compiler directives, and system tasks and functions.

## **Gate Level Modeling**

Verilog has built-in primitives like logic gates, transmission gates and switches. These are rarely used for design work but they are used in post synthesis world for modelling of ASIC/FPGA cells.

Gate level modelling exhibits two properties −

**Drive strength** − The strength of the output gates is defined by drive strength. The output is strongest if there is a direct connection to the source. The strength decreases if the connection is via a conducting transistor and least when connected via a pull-up/down resistive. The drive strength is usually not specified, in which case the strengths defaults to strong1 and strong0.

**Delays** − If delays are not specified, then the gates do not have propagation delays; if two delays are specified, then first one represents the rise delay and the second one, fall delay; if only one delay is specified, then both, rise and fall are equal. Delays can be ignored in synthesis.

### Gate Primitives

The basic logic gates using one output and many inputs are used in Verilog. GATE uses one of the keywords - and, nand, or, nor, xor, xnor for use in Verilog for N number of inputs and 1 output.

Example:

Module gate()

Wire ot0;

Wire ot1;

Wire ot2;

Reg in0,in1,in2,in3;

Not U1(ot0,in0);

Xor U2(ot1,in1,in2,in3);

And U3(ot2, in2,in3,in0)

### Transmission Gate Primitives

Transmission gate primitives include both, buffers and inverters. They have single input and one or more outputs. In the gate instantiation syntax shown below, GATE stands for either the keyword buf or NOT gate.

Example: Not, buf, bufif0, bufif1, notif0, notif1

Not – n outout inverter

Buf – n output buffer

Bufifo – tristate buffer, active low enable

Bufif1 – tristate buffer, active high enable

Notifo – tristate inverter, active low enable

Notif1 – tristate inverter, active high enable

Example:

Module gate()

Wire out0;

Wire out1;

Reg in0,in1;

Not U1(out0,in0);

Buf U2(out0,in0);

## **Data Types**

### Value Set

Verilog consists of, mainly, four basic values. All Verilog data types, which are used in Verilog store these values −

0 (logic zero, or false condition)

1 (logic one, or true condition)

x (unknown logic value)

z (high impedance state)

use of x and z is very limited for synthesis.

### Wire

A wire is used to represent a physical wire in a circuit and it is used for connection of gates or modules. The value of a wire can only be read and not assigned in a function or block. A wire cannot store value but is always driven by a continuous assignment statement or by connecting wire to output of a gate/module. Other specific types of wires are −

**Wand (wired-AND)** − here value of Wand is dependent on logical AND of all the device drivers connected to it.

**Wor (wired-OR)** − here value of a Wor is dependent on logical OR of all the device drivers connected to it.

**Tri (three-state)** − here all drivers connected to a tri must be z, except only one (which determines value of tri).

Example:

Wire [msb:lsb] wire\_variable\_list;

Wirec // simple wire

Wand d;

Assign d = a; // value of d is the logical AND of

Assign d = b; // a and b

Wire [9:0] A; // a cable (vector) of 10 wires.

Wand [msb:lsb] wand\_variable\_list;

Wor [msb:lsb] wor\_variable\_list;

Tri [msb:lsb] tri\_variable\_list;

### Register

A reg (register) is a data object, which is holding the value from one procedural assignment to next one and are used only in different functions and procedural blocks. A reg is a simple Verilog, variable-type register and can’t imply a physical register. In multi-bit registers, the data is stored in the form of unsigned numbers and sign extension is not used.

Example −

reg c; // single 1-bit register variable

reg [5:0] gem; // a 6-bit vector;

reg [6:0] d, e; // two 7-bit variables

### Input, Output, Inout

These keywords are used to declare input, output and bidirectional ports of a task or module. Here input and inout ports, which are of wire type and output port is configured to be of wire, reg, wand, wor or tri type. Always, default is wire type.

**Example**

Module sample(a, c, b, d);

Input c; // An input where wire is used.

Output a, b; // Two outputs where wire is used.

Output [2:0] d; /\* A three-bit output. One must declare type in a separate statement. \*/

reg [1:0] a; // The above ‘a’ port is for declaration in reg.

### Integer

Integers are used in general-purpose variables. They are used mainly in loops-indicies, constants, and parameters. They are of ‘reg’ type data type. They store data as signed numbers whereas explicitly declared reg types store them as an unsigned data. If the integer is not defined at the time of compiling, then the default size would be 32 bits.

If an integer holds a constant, the synthesizer adjusts them to the minimum width needed at the time of compilation.

**Example**

Integer c; // single 32-bit integer

Assign a = 63; // 63 defaults to a 7-bit variable.

### Supply0, Supply1

Supply0 define wires tied to logic 0 (ground) and supply1 define wires tied to logic 1 (power).

**Example**

supply0 logic\_0\_wires;

supply0 gnd1; // equivalent to a wire assigned as 0

supply1 logic\_1\_wires;

supply1 c, s;

### Time

Time is a 64-bit quantity that can be used in conjunction with the $time system task to hold simulation time. Time is not supported for synthesis and hence is used only for simulation purposes.

**Example**

time time\_variable\_list;

time c;

c = $time; //c = current simulation time

### Parameter

A parameter is defining a constant which can be set when you use a module, which allows customization of module during the instantiation process.

Example

Parameter add = 3’b010, sub = 2’b11;

Parameter n = 3;

Parameter [2:0] param2 = 3’b110;

reg [n-1:0] jam; /\* A 3-bit register with length of n or above. \*/

always @(z)

y = {{(add - sub){z}};

if (z)

begin

state = param2[1];

else

state = param2[2];

end

## **Operators**

### Arithmetic Operators

These operators is perform arithmetic operations. The + and −are used as either unary (x) or binary (z−y) operators.

The Operators which are included in arithmetic operation are −

+ (addition), −(subtraction), \* (multiplication), / (division), % (modulus)

**Example** −

parameter v = 5;

reg[3:0] b, d, h, i, count;

h = b + d;

i = d - v;

cnt = (cnt +1)%16; //Can count 0 thru 15.

### Relational Operators

These operators compare two operands and return the result in a single bit, 1 or 0.

Wire and reg variables are positive. Thus (−3’d001) = = 3’d111 and (−3b001)>3b110.

The Operators which are included in relational operation are −

* == (equal to)
* != (not equal to)
* > (greater than)
* >= (greater than or equal to)
* < (less than)
* <= (less than or equal to)

**Example**

if (z = = y) c = 1;

else c = 0; // Compare in 2’s compliment; d>b

reg [3:0] d,b;

if (d[3]= = b[3]) d[2:0] > b[2:0];

else b[3];

Equivalent Statement

e = (z == y);

### Bit-wise Operators

Bit-wise operators which are doing a bit-by-bit comparison between two operands.

The Operators which are included in Bit wise operation are −

* & (bitwise AND)
* | (bitwiseOR)
* ~ (bitwise NOT)
* ^ (bitwise XOR)
* ~^ or ^~(bitwise XNOR)

**Example**

module and2 (d, b, c);

input [1:0] d, b;

output [1:0] c;

assign c = d & b;

end module

### Logical Operators

Logical operators are bit-wise operators and are used only for single-bit operands. They return a single bit value, 0 or 1. They can work on integers or group of bits, expressions and treat all non-zero values as 1. Logical operators are generally, used in conditional statements since they work with expressions.

The operators which are included in Logical operation are −

* ! (logical NOT)
* && (logical AND)
* || (logical OR)

**Example**

wire[7:0] a, b, c; // a, b and c are multibit variables.

reg x;

if ((a == b) && (c)) x = 1; //x = 1 if a equals b, and c is nonzero.

else x = !a; // x =0 if a is anything but zero.

### Reduction Operators

Reduction operators are the unary form of the bitwise operators and operate on all the bits of an operand vector. These also return a single-bit value.

The operators which are included in Reduction operation are −

* & (reduction AND)
* | (reduction OR)
* ~& (reduction NAND)
* ~| (reduction NOR)
* ^ (reduction XOR)
* ~^ or ^~(reduction XNOR)

**Example**

Module chk\_zero (x, z);

Input [2:0] x;

Output z;

Assign z = & x; // Reduction AND

End module

### Shift Operators

Shift operators, which are shifting the first operand by the number of bits specified by second operand in the syntax. Vacant positions are filled with zeros for both directions, left and right shifts (There is no use sign extension).

The Operators which are included in Shift operation are −

* << (shift left)
* >> (shift right)

**Example**

Assign z = c << 3; /\* z = c shifted left 3 bits;

Vacant positions are filled with 0’s \*/

### Concatenation Operator

The concatenation operator combines two or more operands to form a larger vector.

The operator included in Concatenation operation is − { }(concatenation)

**Example**

wire [1:0] a, h; wire [2:0] x; wire [3;0] y, Z;

assign x = {1’b0, a}; // x[2] = 0, x[1] = a[1], x[0] = a[0]

assign b = {a, h}; /\* b[3] = a[1], b[2] = a[0], b[1] = h[1],

b[0] = h[0] \*/

assign {cout, b} = x + Z; // Concatenation of a result

### Replication Operator

The replication operator are making multiple copies of an item.

The operator used in Replication operation is − {n{item}} (n fold replication of an item)

**Example**

Wire [1:0] a, f; wire [4:0] x;

Assign x = {2{1’f0}, a}; // Equivalent to x = {0,0,a }

Assign y = {2{a}, 3{f}}; //Equivalent to y = {a,a,f,f}

For synthesis, Synopsis did not like a zero replication.

For example:-

Parameter l = 5, k = 5;

Assign x = {(l-k){a}}

### Conditional Operator

Conditional operator synthesizes to a multiplexer. It is the same kind as is used in C/C++ and evaluates one of the two expressions based on the condition.

The operator used in Conditional operation is −

(Condition) ? (Result if condition true) −

(result if condition false)

**Example**

Assign x = (g) ? a : b;

Assign x = (inc = = 2) ? x+1 : x-1;

/\* if (inc), x = x+1, else x = x-1 \*/

## **Operands**

### Literals

Literals are constant-valued operands that are used in Verilog expressions. The two commonly used Verilog literals are −

* **String** − A string literal operand is a one-dimensional array of characters, which are enclosed in double quotes (" ").
* **Numeric** − A constant number operand is specified in binary, octal, decimal or hexadecimal Number.

**Example**

n − integer representing number of bits

F − one of four possible base formats −

b for binary, o for octal, d for decimal, h for hexadecimal.

“time is” // string literal

267 // 32-bit decimal number

2’b01 // 2-bit binary

20’hB36F // 20-bit hexadecimal number

‘062 // 32-bit octal number

### Wires, Regs, and Parameters

Wires, regs and parameters are the data types used as operands in Verilog expressions.

### Bit-Selection “x[2]” and Part-Selection “x[4:2]”

Bit-selects and part-selects are used to select one bit and a multiple bits, respectively, from a wire, reg or parameter vector with the use of square brackets “[ ]”. Bit-selects and part-selects are also used as operands in expressions in the same way that their main data objects are used.

**Example**

reg [7:0] x, y;

reg [3:0] z;

reg a;

a = x[7] & y[7]; // bit-selects

z = x[7:4] + y[3:0]; // part-selects

### Function Calls

In the Function calls, the return value of a function is used directly in an expression without the need of first assigning it to a register or wire. It just place the function call as one of the type of operands.it is needful to make sure you are knowing the bit width of the return value of function call.

Example

Assign x = y & z & chk\_yz(z, y); // chk\_yz is a function

. . ./\* Definition of the function \*/

Function chk\_yz; // function definition

Input z,y;

chk\_yz = y^z;

End function

## **Modules**

### Module Declaration

In Verilog, A module is the principal design entity. This indicates the name and port list (arguments). The next few lines which specifies the input/output type (input, output or inout) and width of the each port. The default port width is only 1 bit. The port variables must be declared by wire, wand,. . ., reg. The default port variable is wire. Normally, inputs are wire because their data is latched outside the module. Outputs are of reg type if their signals are stored inside.

**Example**

module sub\_add(add, in1, in2, out);

input add; // defaults to wire

input [7:0] in1, in2; wire in1, in2;

output [7:0] out; reg out;

... statements ...

End module

### Continuous Assignment

The continuous assignment in a Module is used for assigning a value on to a wire, which is the normal assignment used at outside of always or initial blocks. This assignment is done with an explicit assign statement or to assign a value to a wire during its declaration. Continuous assignment are continuously executed at the time of simulation. The order of assign statements does not affect it. If you do any change in any of the right-hand-side inputs signal it will change a left-hand-side output signal.

**Example**

Wire [1:0] x = 2’y01; // assigned on declaration

Assign y = c | d; // using assign statement

Assign d = a & b;

/\* the order of the assign statements does not matter. \*/

### Module Instantiations

Module declarations are templates for creating actual objects. Modules are instantiated inside other modules, and each instantiation is creating a single object from that template. The exception is the top-level module which is its own instantiation. The module’s ports must to be matched to those which are defined in the template. It is specified −

* **By name**, using a dot “.template port name (name of wire connected to port)”. Or
* **By position**, placing the ports in the same place in the port lists of both of the template and the instance.

**Example**

MODULE DEFINITION

Module and4 (x, y, z);

Input [3:0] x, y;

Output [3:0] z;

Assign z = x | y;

End module

**Outcomes:**

i. Able to understand the concept of different types of sequential logic.

ii. Design and analysis different types of sequential logic circuit.

iii. Able to design circuits using VERILOG HDL.