# Unit –ii

## **Boolean Algebra**.

**Boolean Algebra** is the mathematics we use to analyse digital gates and circuits. We can use these "Laws of Boolean" to both reduce and simplify a complex Boolean expression in an attempt to reduce the number of logic gates required. *Boolean Algebra* is therefore a system of mathematics based on logic that has its own set of rules or laws which are used to define and reduce Boolean expressions.

The variables used in **Boolean Algebra** only have one of two possible values, a logic "0" and a logic "1" but an expression can have an infinite number of variables all labelled individually to represent inputs to the expression, For example, variables A, B, C etc, giving us a logical expression of A + B = C, but each variable can ONLY be a 0 or a 1

## **Laws of Boolean Algebra**

- <u>Annulment Law</u> – A term AND´ed with a "0" equals 0 or OR´ed with a "1" will equal 1
-
    - A . 0 = 0    A variable AND'ed with 0 is always equal to 0
    - A + 1 = 1    A variable OR'ed with 1 is always equal to 1
-
- <u>Identity Law</u> – A term OR´ed with a "0" or AND´ed with a "1" will always equal that term
-
    - A + 0 = A   A variable OR'ed with 0 is always equal to the variable
    - A . 1 = A    A variable AND'ed with 1 is always equal to the variable
-
- <u>Idempotent Law</u> – An input that is AND´ed or OR´ed with itself is equal to that input
-
    - A + A = A    A variable OR'ed with itself is always equal to the variable
    - A . A = A    A variable AND'ed with itself is always equal to the variable
-
- <u>Complement Law</u> – A term AND´ed with its complement equals "0" and a term OR´ed with its complement equals "1"
-
    - A . A = 0    A variable AND'ed with its complement is always equal to 0
    - A + A = 1    A variable OR'ed with its complement is always equal to 1
-
- <u>Commutative Law</u> – The order of application of two separate terms is not important
-

    - A . B = B . A    The order in which two variables are AND’ed makes no difference
    - A + B = B + A    The order in which two variables are OR’ed makes no difference
-
- Double Negation Law – A term that is inverted twice is equal to the original term
-
    - A = A    A double complement of a variable is always equal to the variable
-
- de Morgan´s Theorem – There are two “de Morgan´s” rules or theorems,
-
- (1) Two separate terms NOR´ed together is the same as the two terms inverted (Complement) and AND´ed for example:  A+B = A . B
-
- (2) Two separate terms NAND´ed together is the same as the two terms inverted (Complement) and OR´ed for example:  A.B = A + B

Other algebraic Laws of Boolean not detailed above include:

- Distributive Law – This law permits the multiplying or factoring out of an expression.
-
    - A(B + C) = A.B + A.C    (OR Distributive Law)
    - A + (B.C) = (A + B).(A + C)    (AND Distributive Law)
-
- Absorptive Law – This law enables a reduction in a complicated expression to a simpler one by absorbing like terms.
-
    - A + (A.B) = A    (OR Absorption Law)
    - A(A + B) = A    (AND Absorption Law)
-
- Associative Law – This law allows the removal of brackets from an expression and regrouping of the variables.
-
    - A + (B + C) = (A + B) + C = A + B + C    (OR Associate Law)
    - A(B.C) = (A.B)C = A . B . C    (AND Associate Law)

**Boolean Algebra Functions**

Using the information above, simple 2-input AND, OR and NOT Gates can be represented by 16 possible functions as shown in the following table.

| | | |
|---|---|---|
| 1. | NULL | 0 |
| 2. | IDENTITY | 1 |
| 3. | Input A | A |
| 4. | Input B | B |
| 5. | NOT A | A |
| 6. | NOT B | B |
| 7. | A AND B (AND) | A . B |
| 8. | A AND NOT B | A . B |
| 9. | NOT A AND B | A . B |
| 10. | NOT AND (NAND) | A . B |
| 11. | A OR B (OR) | A + B |
| 12. | A OR NOT B | A + B |
| 13. | NOT A OR B | A + B |
| 14. | NOT OR (NOR) | A + B |
| 15. | Exclusive-OR | A . B + A . B |
| 16. | Exclusive-NOR | A . B + A . B |

**Laws of Boolean Algebra Example No1**

Using the above laws, simplify the following expression: (A + B)(A + C)

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

A.A + A.C + A.B + B.C – Distributive law

A + A.C + A.B + B.C – Idempotent AND law (A.A = A)

|  |  |  |
|---|---|---|
|  | A(1 + C) + A.B + B.C | – Distributive law |
|  | A.1 + A.B + B.C | – Identity OR law (1 + C = 1) |
|  | A(1 + B) + B.C | – Distributive law |
|  | A.1 + B.C | – Identity OR law (1 + B = 1) |
| Q = | A + (B.C) | – Identity AND law (A.1 = A) |

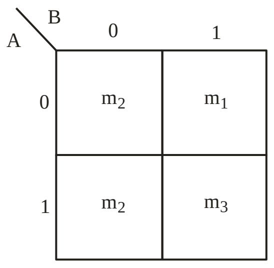
**The K-map method up to five variable**

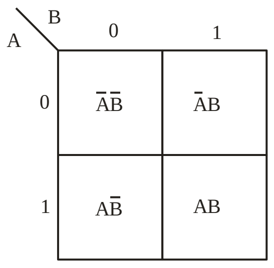
**Karnaugh Map, on the hand is a simple and a straight forward method of minimizing Boolean expressions. Karnaugh Maps are used for many small design problems. Karnaugh Map is also known as *K*-Map. *K*-Map is just another way of presenting the information in a truth table.**

***K*-Map is grid like representation of a truth table. It is made up of squares. It has 0's and 1's entries at different positions. Each position in a grid corresponds to a truth table entry. So, in other words, *K*-Map is just another way of presenting the information in a truth table.**

**<u>Two variable map</u>**

***K*-Map has an advantage that it is designed to present the information in a way that allow easy grouping of terms that can be combined. It is pictorial map method. Fig. shows a two variable map.**





**In fig *A* and *B* are the two inputs having two possible values 0 and 1 individually. Combining the inputs we have four possible combinations *A'B'*, *A'B*, *AB'*, *AB* and can be represented by $m_0$, $m_1$, $m_2$, $m_3$. If we have to represent *A'B* then it is equal to $m_1$ and belong to the square to which $m_1$ belongs. Example below shows the relation between the *K*-Map and truth table for a two-variable problem.**

| ***A*** | ***B*** | ***F*** |
|---|---|---|

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

**Table . Truth table for two variables**

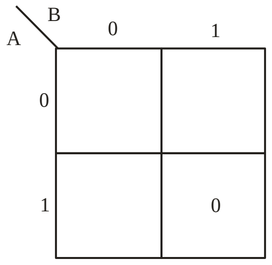


**Fig. *K*-Map for two variables**

**In this the square on the lower right of map has coordinates *A* = 1 and *B* = 0.  This square corresponds to the row in the truth table where *A* = 1 and *B* = 0 and *F* = 0.**

**<u>Three Variable Map</u>. In this map we have three input variables and minimum one output. For three variables the possible input combinations are 000, 001, 010, 011, 100, 101, 110, 111. So, we can make the *K*-Map by taking these values in such a way that adjacent squares differ only in one way *i.e.***

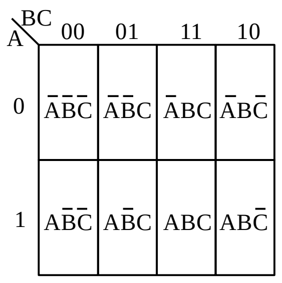


**Fig. *K*-Map for three variables.**

***K*-Maps are used for minimization; this can be achieved by combining those squares in *K*-map which contain 1’s.**

**<u>Four Variable Map</u>. We make this map similarly as three variable map, we have a total of 8 squares *i.e.* $2^3$, so in four variable maps we create $2^4$ = 16 squares.  If the variables are *A, B, C, D* the**

| AB \ CD | 00 | 01 | 11 | 10 |
|---|---|---|---|---|
| 00 | A′B′C′D | ABCD | A′B′CD | A′B′CD′ |
| 01 | A′BC′D′ | ABCD | A′BCD | A′BCD′ |
| 11 | ABC′D′ | ABCD | ABCD | ABCD′ |
| 10 | AB′C′D′ | ABCD | AB′CD | AB′CD′ |

**Fig. . *K*-Map for four variables**

**For minimization or simplification of *K*-Map we follow the given rules**

**(i) Groups may either contain only 1's or only 0's.**

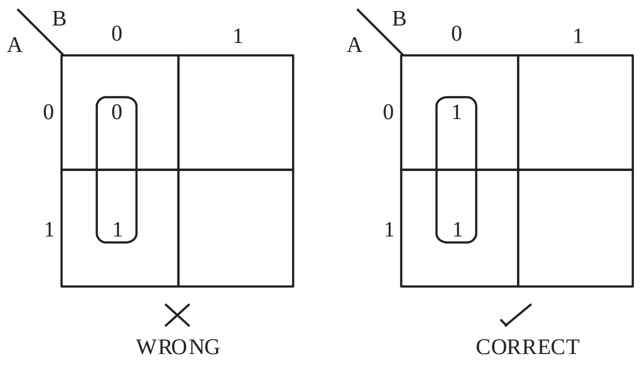


**Fig. Grouping of 1's.**

**(ii) Groups may be vertical or horizontal but not diagonal.**

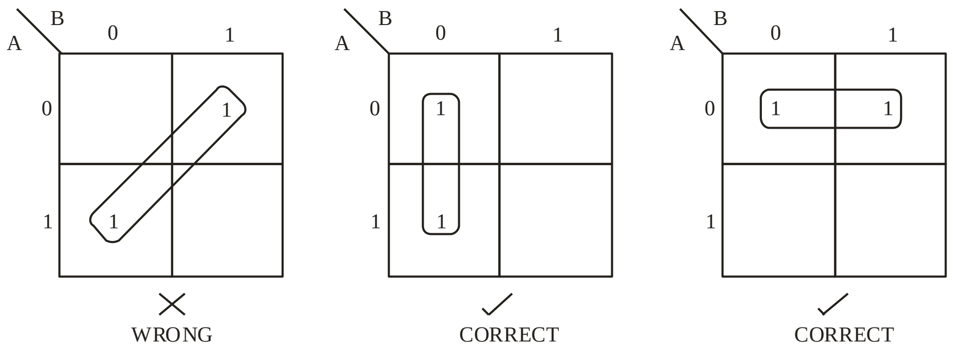
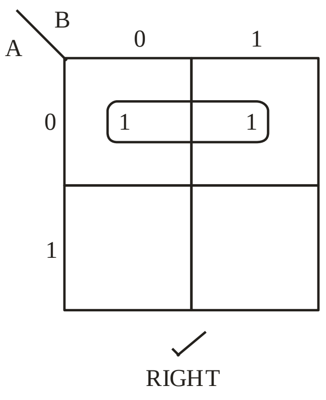


**Fig. Vertical and horizontal grouping of 1's**

**(iii) Map may contain 1, 2, 4, 8 or in general $2^n$ cells *i.e.* if $n = 1$, a group will contain two 1's as $2^1 = 2$ where *n* is the number of input variable and a group may contain as many number of cells which satisfy equation $2^n$.**



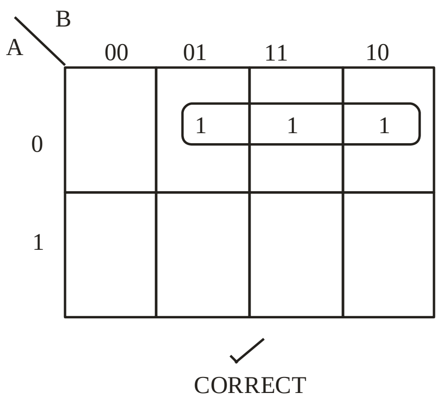
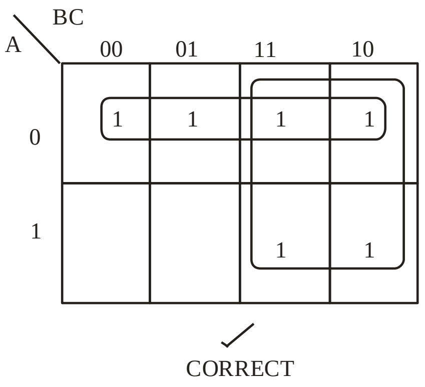


**Fig. Two inputs so total of 4 cells and a group of $2 = 2^1$**

**Three inputs so 8 cells is correct and group of 3 is wrong as 3 does not satisfy $2^n$**

**(iv) Each group should be as large as possible.**



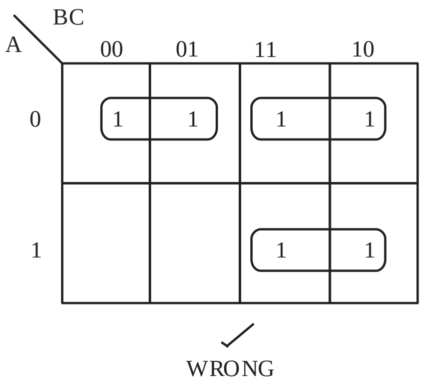


**Fig. Four 1's grouped together.**

**(v) Each cell containing *a* 1 must be in at least one group.**

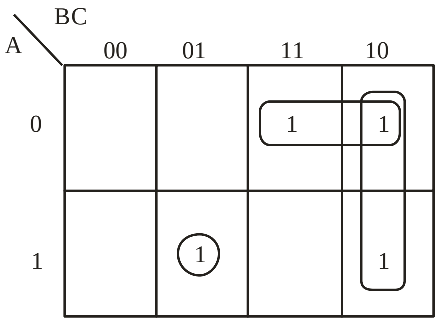


**Fig. Left out 1 grouped alone**

**Each 1 should be in at least one group, if that is not possible the left out 1 is grouped alone.**

**(i) Groups may overlap.**

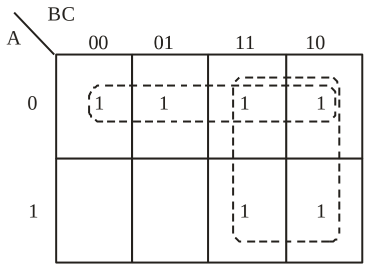


**Fig. Groups overlapping**

**(ii) Groups may wrap around the table. The leftmost, cell in a row may be grouped with the rightmost cell and the top cell in a column may be grouped with the bottom cell.**

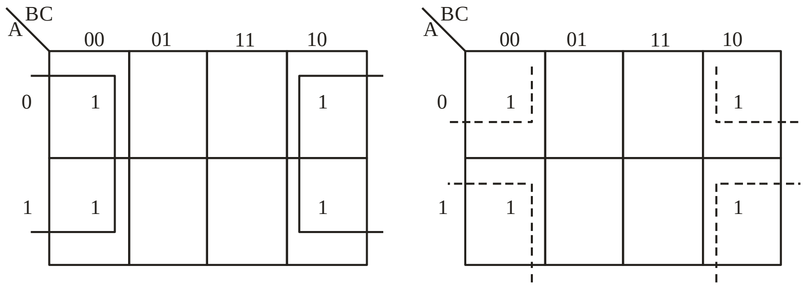


**Fig. Groups wrapping around the table**

**(iii) There should be as few groups as possible, as long as this not contradict any of the previous rules.**

**Example. Minimize the following**

**(i)** $Z = \bar{A}\bar{B} + A\bar{B} + \bar{A}B$

**(ii)** $Z = \bar{A}\bar{B}\bar{C} + \bar{A}B + AB\bar{C} + AC$

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

**Solution: (i)** $Z = \bar{A}\bar{B} + A\bar{B} + \bar{A}B$

**We make the 2 variable *K*-Map**

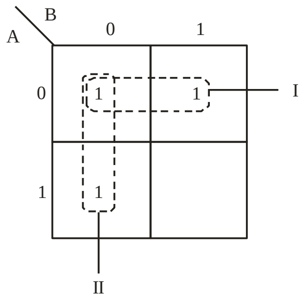


**Fig Two variable K-Map for equation** $Z = \bar{A}\bar{B} + A\bar{B} + \bar{A}B$

**In this example, according to rule (iii) 1's are grouped horizontal as well as vertically and we generate two groups *i.e.* group I and group II. In group I, two 1's are grouped which corresponds to *A* = 0 & *B* = 0 and *A* = 0 *B* = 1 or we can say that they corresponds to *A* = 0 independent of the value of *B*. The expression of the output will contain a term *A*'.**

**Similarly, group II will contribute *B'* in the output. So, simplifying we get**

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

**(ii)** $Z = \bar{A}\bar{B}\bar{C} + \bar{A}B + AB\bar{C} + AC$

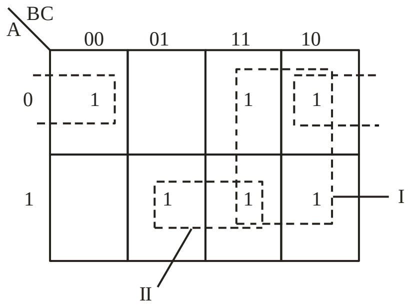


**Fig Three variable K-Map for equation** $Z = \bar{A}\bar{B}\bar{C} + \bar{A}B + AB\bar{C} + AC$

**In this example using rule IV maximum number of 1's *i.e.* four 1's in this case are grouped giving group I. In this example, there are a total of three groups I, II, III.**

**Group I reduces to *B***

**Group II reduces to *AC***

**Group III reduces to *AC'***

**Simplifying we get**

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

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

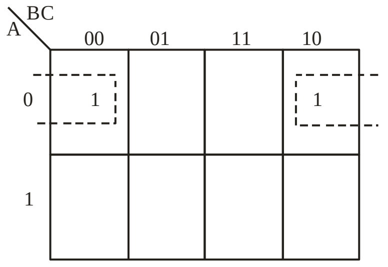


**Fig Three variable *K*-Map for equation** $\boldsymbol{Z = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C}}$

$Z = \bar{A}\bar{C}$

**Five and Six Variable Map. Map for 5 variables will contain $2^5$ = 32 squares and for six variables will contain 64 squares. Handling such large number of squares and combining adjacent squares becomes tedious.**

**A five variable map is having five input variables say *A, B, C, D, E* then we can represent the function using two four variable maps each distinguished by the value as shown in fig below**

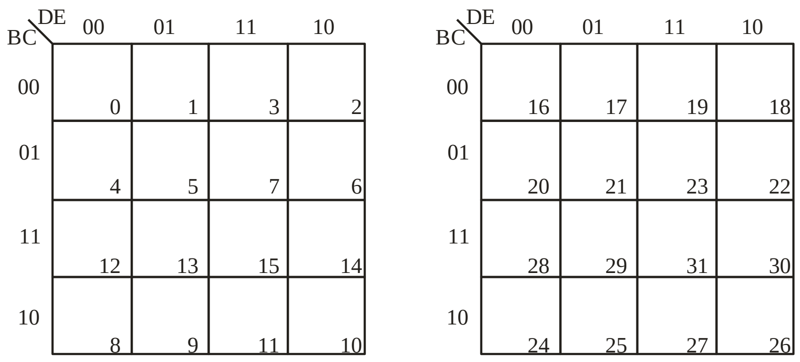


**Fig variable *K*-Map**

**The two maps have variable *B, C, D, E* and are distinguished by *A* = 0 and *A* = 1. For map having *A* = 0 the squares represents minterms 0 through 15 and the map for *A* = 1 represents from 16 to 31**

**Don’t Care Conditions.**

**Don't care conditions can be defined as a condition where the output levels are not specified for the inputs *i.e.* we don't consider the outputs for these inputs. In such situations designer has a flexibility and it is left to him whether to assume *a* 0 or *a* 1 as outputs for each of these combinations. Don't care conditions are represented as *X* mark on the *K*-Map. The *X* mark in a cell may be assumed to be *a* 1 or *a* 0 depending upon which one leads to a simpler expression.**

**Example. Simplify the Boolean function :**

$$F(A, B, C) = \Sigma(5, 6, 7) + d(3, 4)$$

**where *d*(3, 4) represents don't care conditions.**

**Solution:**

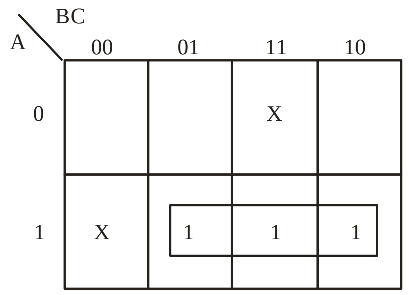
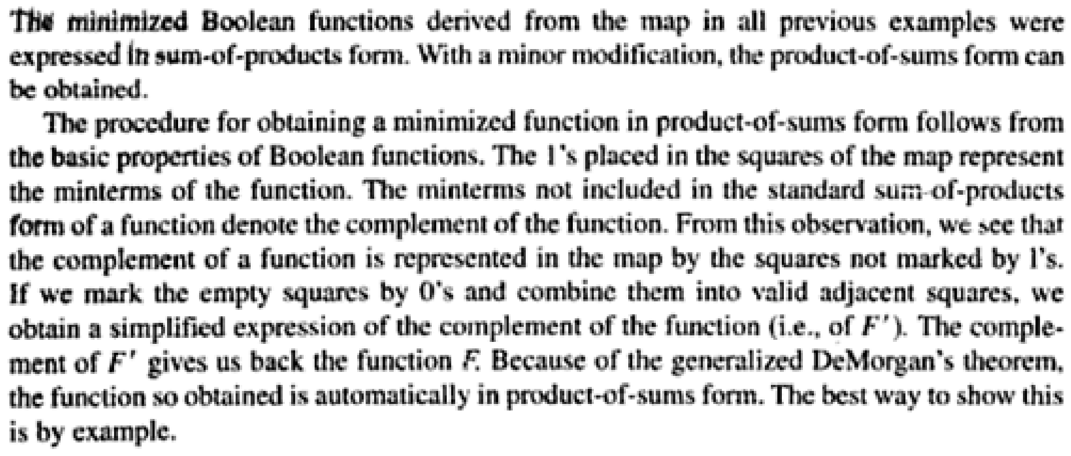


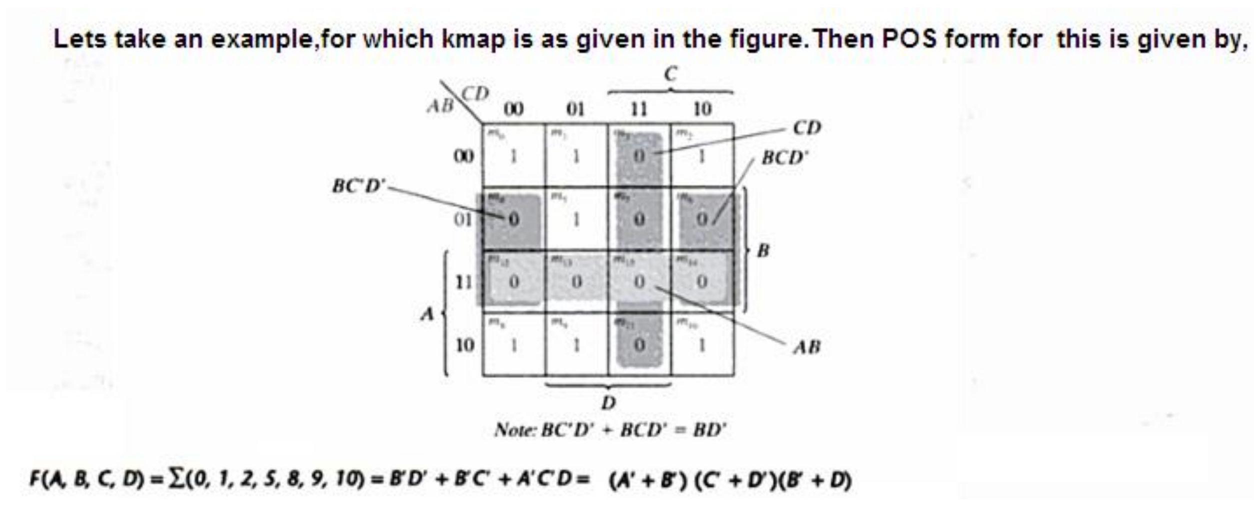
**Fig.Three variable K-Map for function $F(A, B, C) = \Sigma(5, 6, 7) + d(3, 4)$**

**POS simplification**



The minimized Boolean functions derived from the map in all previous examples were expressed in sum-of-products form. With a minor modification, the product-of-sums form can be obtained.

The procedure for obtaining a minimized function in product-of-sums form follows from the basic properties of Boolean functions. The 1's placed in the squares of the map represent the minterms of the function. The minterms not included in the standard sum-of-products form of a function denote the complement of the function. From this observation, we see that the complement of a function is represented in the map by the squares not marked by 1's. If we mark the empty squares by 0's and combine them into valid adjacent squares, we obtain a simplified expression of the complement of the function (i.e., of $F'$). The complement of $F'$ gives us back the function $F$. Because of the generalized DeMorgan's theorem, the function so obtained is automatically in product-of-sums form. The best way to show this is by example.


Lets take an example,for which kmap is as given in the figure.Then POS form for  this is given by,

$$F(A, B, C, D) = \Sigma(0, 1, 2, 5, 8, 9, 10) = B'D' + B'C' + A'C'D = (A' + B')(C' + D')(B' + D)$$

## Flip Flops

The memory elements in a sequential circuit are called flip-flops, which are capable of storing 1-bit of information. A flip-flop is a synchronous version of the latch. "Flip-flop" is the common name given to two-state devices, which offer basic memory for sequential logic operations. Flip-flops are heavily used for digital data storage and transfer and are commonly used in banks called "registers" for the storage of binary numerical data. A flip-flop circuit has two outputs, one for the normal value and one for the complement value of the stored bit. Binary information can enter a flip-flop in a variety of ways and gives rise to different types of flip-flops. A flip-flop is usually controlled by one or two control signals and/or a gate or clock signal. The main difference between latches and flip-flops is the method used to change their states. Latches are level sensitive, or level-triggered. This means that the outputs are dependent on the voltage level applied, not on any signal transition. Flip-flops are edge-triggered, that is that they depend on the transition of a signal. This may either be a LOW-to-HIGH (rising edge) or a HIGH-to-LOW (falling edge) transition.

### S- R Flip-Flop:

S-R (Set-Reset) flip flop can be constructed from NAND gates as shown in fig..

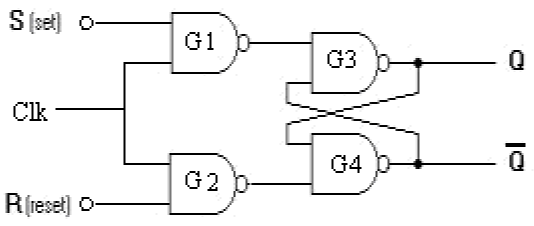


Fig. :  SR Flip-Flop constructed from NAND gates

In the circuit shown in fig., there are three inputs S(Set), R (Reset) and Clk (Clock). In this circuit, Clk input acts as a enable signal for the other two inputs i.e. S and R inputs. The circuit operates normally when Clk input is 1 and when it is 0 the circuit remains in its previous state. The information from S and R input is allowed to reach the output only when the Clk input is 1. When Clk= 1, we can analyze the working of SR flip flop shown in fig. as follows:

**Case(i):** When S = 1 and R = 0, then the inputs at G1 will be 11 and at G2 will be l0 and as a result the output of G1 is 0 and G2 is 1. These outputs then become the input to G3 and G4 respectively. We know that if any one input to NAND gate is 0, the output will be 1. So, the output of G3 will be 1 i.e. $Q$ = 1. Now the input at G4 becomes 11 and its output goes to 0 i.e. $\bar{Q}$ =0. This state when Q = 1 $\bar{Q}$ = 0 is called as SET State.

**Case(ii):** Now when the input changes from S = 1 and R = 0 to S=0 and R=1, then the inputs at G1 will be 01 and at G2 will be l1 and as a result the output of G1 is 1 and G2 is 0. These outputs then become the input to G3 and G4 respectively. We know that if any one input to NAND gate is 0, the output will be 1. So, the output of G3 will be 1 i.e. $Q$ = 0. Now the input at G4 becomes 11 and its output goes to 0 i.e. $\bar{Q}$ =1. This state when Q = 0 $\bar{Q}$ = 1 is called as RESET State.

**Case(iii):** When from any of the two states explained above, the input changes to S = 0 and R = 0, then we see that the output remains the same as was in previous state i.e. there is no change in the output.

When inputs change to S = R = 1, then an indeterminate condition occurs. In this condition both the outputs Q and $\bar{Q}$ are 0, which is not possible.

Table shows the truth table or characteristic table of SR flip flop.

| *S* | *R* | *Q* | $\bar{Q}$ |
|---|---|---|---|
| 1 | 0 | 1 | 0 (SET) |
| 0 | 0 | 1 | 0(No Change) |
| 0 | 1 | 0 | 1 (RESET) |
| 0 | 0 | 0 | 1(NoChange) |
| 1 | 1 | Not Allowed | |

*Table: Truth Table for SR flip flop*

The standard truth table is given in table.

| *S* | *R* | *Q(t+1)* |
|---|---|---|
| 0 | 0 | Q(t) |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | Not Allowed |

*Table : Truth Table for SR flip flop*

In table , output Q(t+1) represents next state and Q(t) represents present sate. When inputs are 00 then the output in the next sate will be same as present state i.e. Q(t+1) = Q(t) and so on.

**Preset and Clear:**

Initially when the device is switched ON then the state of the device is undefined and so is the states of the memory element *i.e.* flip flop. So, to avoid this condition we assign the initial state to the flip flop using two asynchronous inputs Preset (*Pr*) and Clear (*Cr*). These inputs can be used along with the Clock but are not in synchronization with it. When both *Pr* and *Cr* inputs are High *i.e.* *Pr* = *Cr* = 1 then the flip-flop operates normally. When *Pr* = 1 and *Cr* = 0 then the flip-flop is RESET. When *Pr* = 0 and *Cr* = 1 then the flip-flop is SET. The condition *Pr* = *Cr* = 0 must be avoided as in this condition the state of the flip-flop will be uncertain.

| *Pr* | *Cr* | Operation of flip flop |
|---|---|---|
| 1 | 1 | Normal operation |
| 1 | 0 | Reset ($Q = 0$) |
| 0 | 1 | Set ($Q = 1$) |

*Table*

**SR Flip-Flop with Preset & Clear :**

Block Diagram of Clocked *SR* flip-flop is shown in fig. It consists of Clock (*Clk*), Preset (*Pr*) and Clear (*Cr*) inputs along with the Set (*S*) and Reset (*R*) inputs.

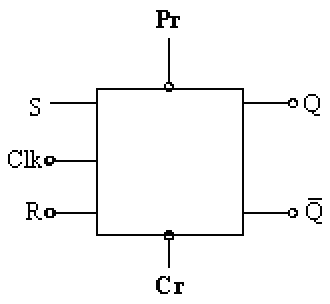


**Fig. : Block diagram of Clocked *SR* flip-flop.**

Clock signal is used for synchronization in a circuit. The circuit operates when the clock signal is HIGH *i.e.* Clk = 1. Preset and Clear inputs applied to the flip flop with a bubble which signifies that these signals are active LOW *i.e.* they are activated when their values are 0. A clocked *SR* flip-flop with *Clk*, *Clr* and *Pr* inputs is shown in fig. below

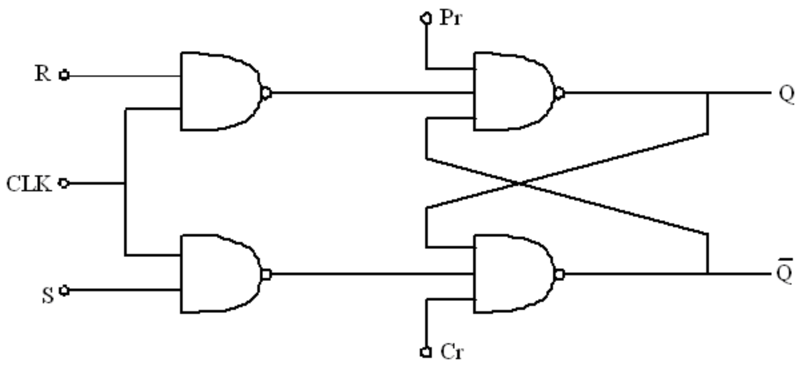


**Fig. : Clocked SR flip-flop.**

*SR* flip-flop is activated *i.e.* the flip-flop to the inputs only when the clock is high (*Clk* =1). The operation of *SR* flip-flop is similar to *SR* latch when *Clk* = 1 and *Pr* = *Cr* = 1. When *Pr* = 0 and *Clr* = 1 then the flip-flop is SET *i.e. Q* = 1, and when *Pr* = 1 and *Cr* = 0 then the circuit is RESET *i.e. Q* = 0. *Pr* and *Cr* signals are not in synchronization with the Clock pulse. They can operate both in the presence as well as in the absence of the Clock pulse. Truth table for *SR* flip-flop is given in Table given below.

*Table : Truth Table for SR flip flop*

| ***Clk*** | ***Pr*** | ***Cr*** | ***Output*** |
|---|---|---|---|
| 1 | 1 | 1 | NormalOperation |
| X | 1 | 0 | 1(Preset) |
| X | 0 | 1 | 0(Clear) |

In table, the don't care condition (X) of Clk input represents that Pr and Cr inputs can override the Clk input i.e. the circuit can be set or reset even in the presence of the Clk input.

## ***JK* Flip-Flop**

The *JK* flip flop is the most versatile flip-flop, and the most commonly used flip-flop when discrete devices are used to implement arbitrary state machines. The *RS* flip-flop has a limitation that it can only be used in applications where it can be guaranteed that both *R* and *S* cannot be logic 1 at the same time as this can condition is indeterminate. The *JK* flip-flop augments the behavior of the *SR* flip-flop by interpreting the *S* = *R* = 1 condition. A *JK* flip-flop is a refinement of the *SR* flip-flop in that the indeterminate state of the *SR* type is defined in the *JK* type. Block diagram of *JK* flip-flop is shown in fig..

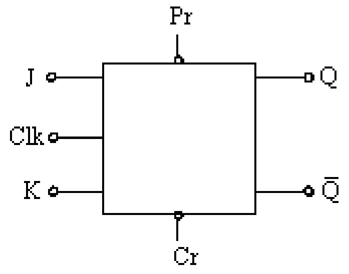


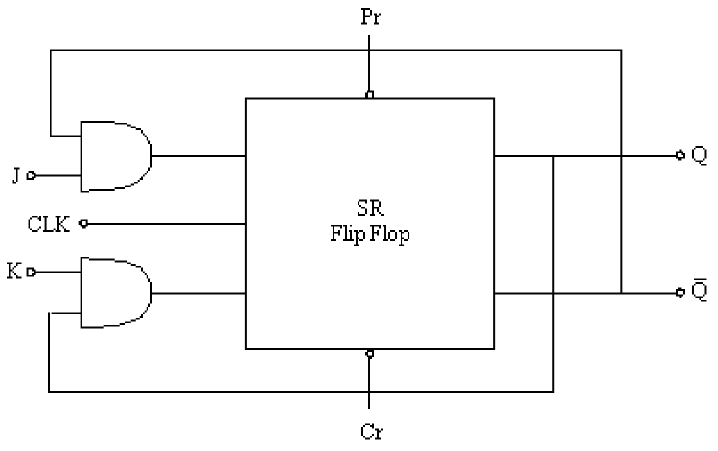
**Fig. Block Diagram of *J-K* Flip Flop.**

We can obtain a *JK* flip flop from *SR* flip-flop by using following equations;

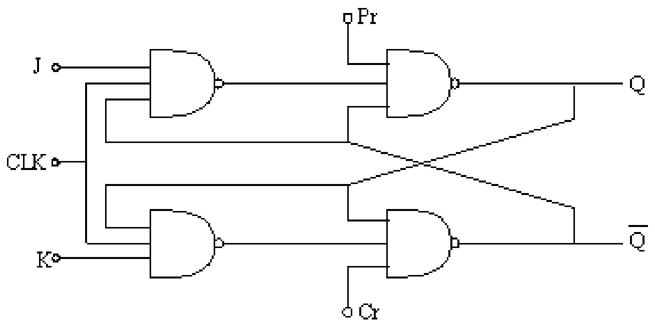
$$S = J\bar{Q}$$

$$R = KQ$$

A *JK* flip flop obtained from such operation is shown in fig.



(a)



(b)

**Fig.: A *J-K* Flip Flop obtained from *S-R* Flip Flop.**

Fig.(a) shows the basic block diagram of the conversion of *SR* flip flop to *JK* flip-flop. In this block diagram the inputs of *SR* flip flop are ANDed with the outputs according to the equation defined. Fig (b) shows the logic circuit of the JK flip-flop using SR flip flop along with the Pr and Cr inputs. The operation of *JK* flip flop is summarized as follows :

**Case** (i) **When *Pr* = 1 and *Cr* = 1**

When both *Pr* and *Cr* are HIGH and *Clk* = 1 then the circuit operates normally according to the inputs. So, we consider the following cases:

(a) **When both the inputs are LOW *i.e. J* = *K* = 0** then the output of the flip flop is same as it was in the previous state *i.e.* $Q(t + 1) = Q(t)$.

(b) **When both the inputs are HIGH i.e. J=K=1** then the flip-flop switches to complement of previous state, *i.e.,* $Q(t + 1) = \overline{Q(t)}$ .

(c) **When *J* = 0 and *K* = 1** then the flip flop is RESET *i.e.* $Q(t + 1) = 0$.

(d) **When *J* = 1 and *K* = 0** then the flip flop is SET *i.e.* $Q(t + 1) = 1$.

**Case** (ii) **When *Pr* = 0 and *Clr* = 1** then the flip flop is SET *i.e.* $Q(t + 1) = 1$ whatever be the values of the inputs $J$ and $K$.

**Case** (iii) **Similarly, When *Pr* = 1 and *Clr* = 0** then the flip flop is cleared *i.e.* $Q(t+1) = 0$

The condition $Pr = Cr = 0$ must be avoided as in this condition the state of the flip-flop will be uncertain. Table shows the truth table of *JK* flip-flop.

| ***Clk*** | ***Pr*** | ***Clr*** | ***J*(*t*)** | ***K*(*t*)** | ***Q*(*t* + 1)** |
|---|---|---|---|---|---|
| 1 | 0 | 1 | × | × | 1 |
| 1 | 1 | 0 | × | × | 0 |
| 1 | 1 | 1 | 0 | 0 | $Q(t)$ |
| 1 | 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | $\overline{Q(t)}$ |

*Table: Truth Table for JK Flip flop*

**Race Around Condition**

In *JK* flip-flop we have observed that our circuit operates normally when the *Clk* signal is HIGH. In this theory we have assumed that the inputs remain the same during the clock pulse. But this is not true, the input can change multiple times from 0 to 1 and 1 to 0 when the *Clk* signal is high, due to feedback.

Let us take condition $J = K = 1$ and $Q = 0$ in account. The input clock pulse is applied is of width $t_p$ as shown in fig 4.3.5.

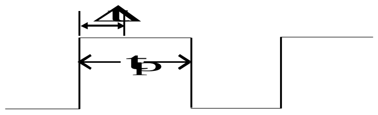


**Fig. : Clock Pulse of width $t_p$.**

The propagation delay of NAND gates is assumed to be $\Delta t$. So after $\Delta$t time interval the output changes from $Q = 0$ to $Q = 1$. Similarly after $\Delta t$ time period the output will again change from 1 to 0. If $t_p = 3 \times \Delta$t, then in a single clock pulse the output will change thrice. So, at the end of the clock pulse the output would be uncertain. This condition of uncertainty in the output is known as Race Around Condition.

Race around condition can be removed when $t_p < \Delta t$. This condition is quite difficult to achieve as the propagation delay itself is very small. There are two more ways of eliminating race around condition. They are :

(i) Master Slave Configuration (ii) Edge Triggering

## Master Slave JK Flip Flop:

A master-slave flip-flop is constructed from two separate JK flip-flops. One circuit serves as a master and the other as a slave. The block-diagram of Master Slave *JK* flip-flop is shown in Fig. 4.3.6(a).

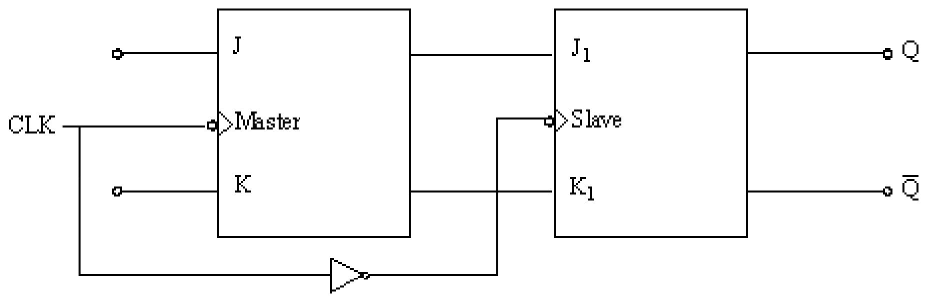


**Fig. (a) : Block Diagram of Master Slave JK flip flop**

The first flip-flop *i.e.* master is enabled on the positive edge of the clock pulse *Clk* and the second flip-flop *i.e.* slave is enabled on the negative edge of *Clk*. When *Clk* = 1 then the first flip-flop is enabled and produces the output, whereas at this time for the second flip flop *Clk* = 0 through the inverter and so is disabled. Again when *Clk* = 0, the first flip flop is disabled and the output is the same as before, whereas now second flip flop is enabled and accepts the output of the master flip flop as input and accordingly produces the output. So, it is clear that the operation of first flip-flop depends on the primary inputs and of second flip-flop depends on first flip-flop *i.e.* the second flip-flop simply follows the first flip-flop. Because of this reason the first flip-flop is called as master and the second is called slave. The logic circuit of Master Slave *JK* flip-flop is shown in fig.(b).

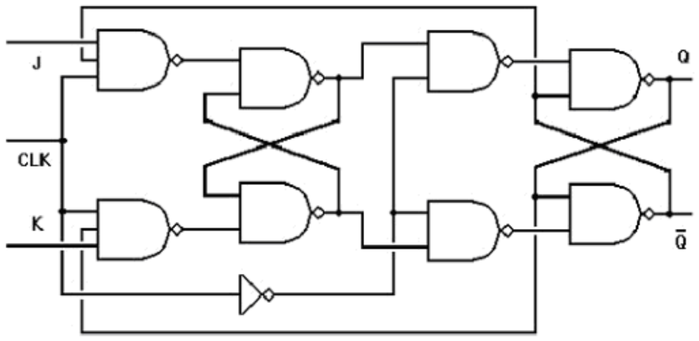


**Fig. (b) : Master Slave JK Flip Flop.**

The working of master slave *JK* flip flop overcomes the problem of race around condition that occurs with a *JK* Flip-when *J* = *K* = 1. As any input to the master-slave flip-flop at J and K is first seen by the master FF part of the circuit while *CLK* is High (=1). This behavior effectively "locks" the input into the master *FF*. An important feature here is that the complement of the *CLK* pulse is fed to the slave *FF*. Therefore the outputs from the master *FF* are only "seen" by the slave *FF* when *CLK* is Low (= 0). Therefore on the High-to-Low *CLK* transition the outputs of the master are fed through the slave *FF*. This means that the at most one change of state can occur when *J* = *K* = 1 and so oscillation between the states *Q* = 0 and *Q* = 1 at successive *CLK* pulses does not occur.

The timing relationship is shown in Fig and is assumed that the flip-flop is in the clear state prior to the occurrence of the clock pulse. The output state of the master-slave flip-flop occurs on the negative transition of the clock pulse.

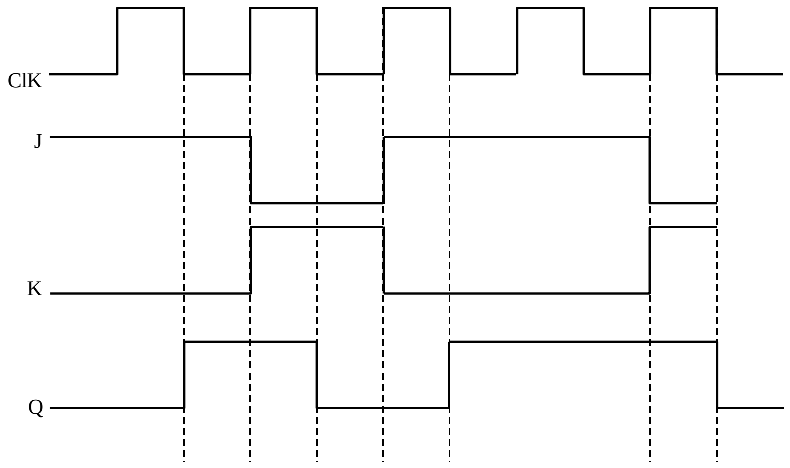
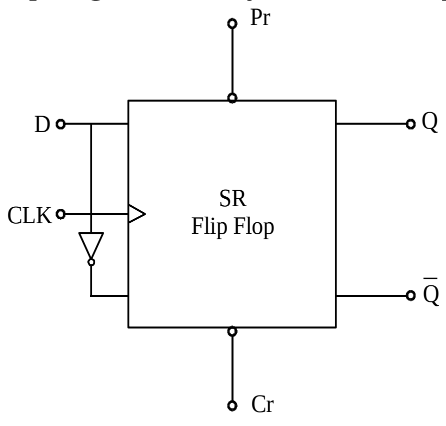
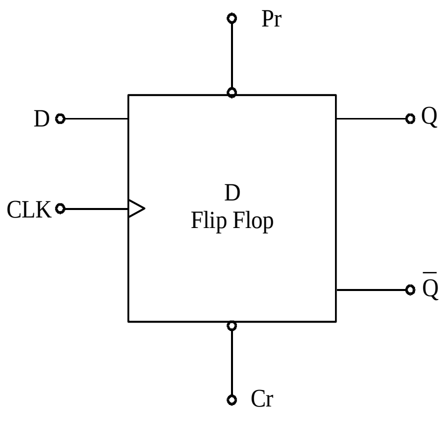
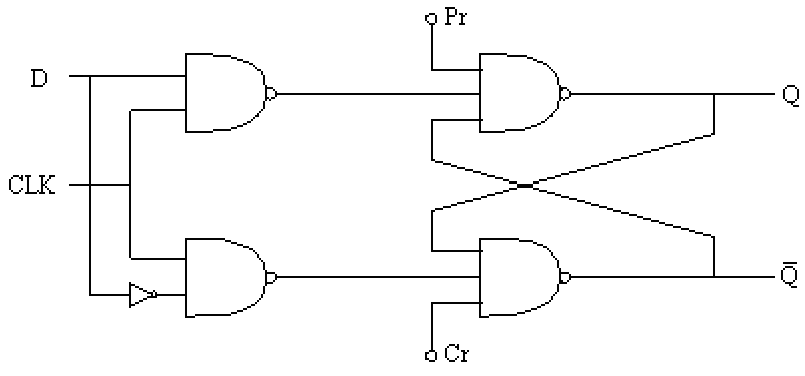


**Fig  Timing relationship in a master slave flip-flop. *D* (Delay) Flip Flop :**

We used the *JK* flip-flop to eliminate the indeterminate state of *SR* flip flop *i.e.* when $S = R = 1$. Another way to eliminate this condition is by making a arrangement such that the two inputs are always complementary to each other. This condition is achieved in *D* flip flop in which the *D* input goes directly to the S input and with a inverter to the *R* input as shown in fig..







The characteristic equation of *D* flip flop is $Q\ (t + 1) = D$

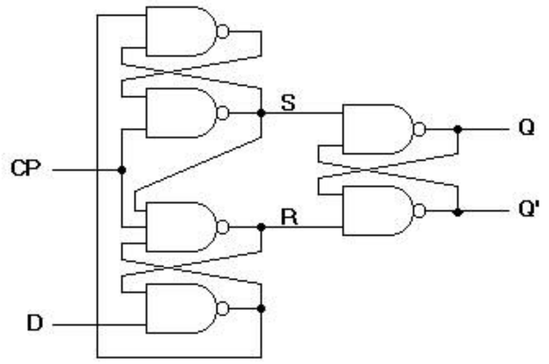
The operations of *D* flip flop are summarized in the truth table shown in table 4.3.3.

| $D$ | $Q(t + 1)$ |
|---|---|
| 0 | 0 |
| 1 | 1 |

**Table: Truth Table of clocked *D* Flip Flop**

It is clear from the truth table shown in table above that the output at the end of the clock pulse *i.e.* $Q(t + 1)$ is equal to the input *D* before the clock pulse. We can summarize this as data in *D* flip flop is transferred to the output after a certain delay and because of this reason this flip flop is known as Delay flip flop.

A positive edge triggered *D*-flip flop is shown in fig.



In fig, there are six NAND gates, which constitute three basic *SR* flip-flops. The two *SR* flip-flops in the beginning produces outputs, which are fed to third flip flop at the end as inputs *S* and *R* and accordingly produces the final output.

***T* (Toggle) Flip Flop:**

*T* flip-flop is a simplified version of the *JK* flip-flop and can be obtained from *JK* flip-flop. When both the inputs of *JK* flip-flop are tied together, then the resulting flip flop is the *T*-flip flop. Block diagram of *T* flip-flop is shown in fig..

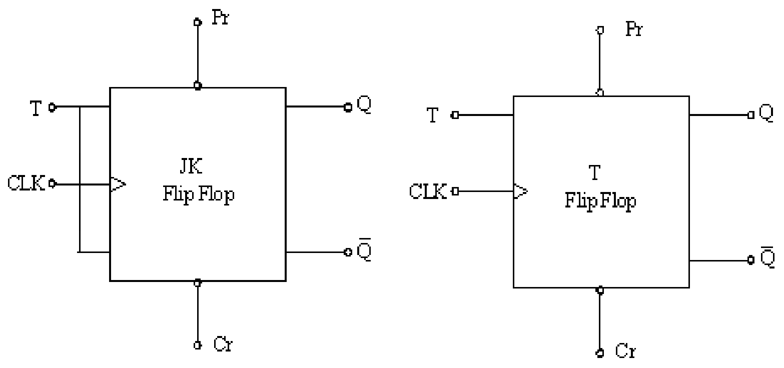


**Fig: Block Diagram of *T* flip flop.**

Fig (a) shows the construction of *T* flip-flop from *JK* flip flop and fig. (b) shows the block diagram of *T* flip flop. If the toggle (*T*) input is high, the *T* (Toggle) flip-flop changes state ("toggles"). If the *T* input is low, it holds the previous value. The logic circuit of *T*-flip flop is shown in fig.

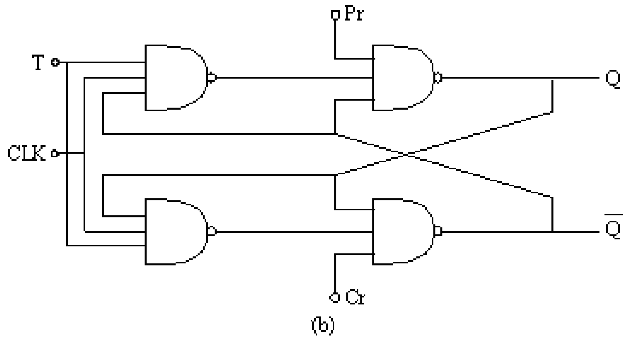


**Fig. : Clocked *T* flip-flop.**

The *T* flip-flop operates normally when the clock input is HIGH *i.e. Clk* = 1. The operation of *T* flip-flop for *Clk* = 1 is summarized as:

When *T* = 0, the flip-flop does a hold. A hold means that the output, *Q* is kept the same as it was before the clock edge *i.e.* $Q(t + 1) = Q(t)$. When *T* = 1, the flip-flop does a toggle, which means the output Q is negated after the clock edge, compared to the value before the clock edge. *i.e.* $Q(t + 1) = \overline{Q(t)}$ . The truth table for *T* flip flop is shown in table.

| $Q(t)$ | $T$ | $Q(t + 1)$ |
|---|---|---|
| 0 | 0 | 0 ($Q(t)$) |
| 0 | 1 | 1 ($\overline{Q(t)}$) |
| 1 | 0 | 1 ($Q(t)$) |
| 1 | 1 | 1 ($\overline{Q(t)}$) |

| T | $Q(t + 1)$ |
|---|---|
| 0<br>1 | $Q(t)$<br>$\overline{Q(t)}$ |

**Truth table for *T* flip flop**

This behavior is described by the characteristic equation :

$$Q(t + 1) = T\,\overline{Q(t)} + \bar{T}\,Q(t)$$