# LOGIC GATES

In electronics, a logic gate is an idealized or physical device implementing a Boolean function; that is, it performs a logical operation on one or more binary inputs and produces a single binary output.

Logic gates are primarily implemented using diodes or transistors acting as electronic switches, but can also be constructed using vacuum tubes, electromagnetic relays (relay logic), fluidic logic, pneumatic logic, optics, molecules, or even mechanical elements. With amplification, logic gates can be cascaded in the same way that Boolean functions can be composed, allowing the construction of a physical model of all of Boolean logic, and therefore, all of the algorithms and mathematics that can be described with Boolean logic.

Boolean functions may be practically implemented by using electronic gates. The following points are important to understand.

- Electronic gates require a power supply.
- Gate INPUTS are driven by voltages having two nominal values, e.g. 0V and 5V representing logic 0 and logic 1 respectively.
- The OUTPUT of a gate provides two nominal values of voltage only, e.g. 0V and 5V representing logic 0 and logic 1 respectively. In general, there is only one output to a logic gate except in some special cases.
- There is always a time delay between an input being applied and the output responding.

## Truth Table-

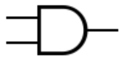
A truth table is a mathematical table which sets out the functional values of logical expressions on each of their functional arguments, that is, for each combination of values taken by their logical variables.

## Types of Logic Gates-

There are seven basic logic gates: **AND, OR, NOT, NAND, NOR, XOR and XNOR**. Among which NAND and NOR gates are called universal functions since with either one them, AND and OR functions and NOT can be generated.

### 1. <u>AND Gate</u>

The AND gate is an electronic circuit that gives a true output (1) only if all its inputs are true. A dot (.) is used to show the AND operation i.e. A.B. This dot is sometimes omitted i.e. AB

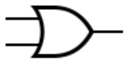
*Logic Gate Symbol-* 

*Truth Table-*

| **Input** | | **Output** |
|---|---|---|
| $A$ | $B$ | $A.B$ |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### 2. <u>OR Gate</u>

The OR gate is an electronic circuit that gives a true output (1) if one or more of its inputs are true. A plus (+) is used to show the OR operation.

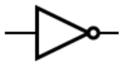
*Logic Gate Symbol-* 

*Truth Table-*

| Input | | Output |
|---|---|---|
| $A$ | $B$ | $A+B$ |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

3. **NOT Gate**

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A', or A with a bar over the top.

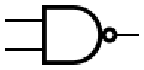
*Logic Gate Symbol-* 

*Truth Table-*

| Input | Output |
|---|---|
| $A$ | $A'$ or $\overline{A}$ |
| 0 | 1 |
| 1 | 0 |

4. **NAND Gate**

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are true if any of the inputs are false. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.

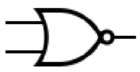
*Logic Gate Symbol-* 

*Truth Table-*

| Input | | Output |
|---|---|---|
| $A$ | $B$ | $\overline{A.B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

5. **NOR Gate**

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are false if any of the inputs are true. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

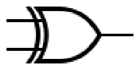
*Logic Gate Symbol-* 

*Truth Table-*

| Input | | Output |
|---|---|---|
| $A$ | $B$ | $\overline{A+B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

## 6. <u>XOR Gate</u>

The 'Exclusive-OR' gate is a circuit which will give a true output if either, but not both, of its two inputs are true. An encircled plus sign ( $\oplus$ ) is used to show the XOR operation.

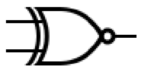
*Logic Gate Symbol-* 

*Truth Table-*

| Input | | Output |
|---|---|---|
| $A$ | $B$ | $A \oplus B$ |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## 7. <u>XNOR Gate</u>

The 'Exclusive-NOR' gate circuit does the opposite to the XOR gate. It will give a false output if either, but not both, of its two inputs are true. The symbol is an XOR gate with a small circle on the output. The small circle represents inversion.

*Logic Gate Symbol-* 

*Truth Table-*

| Input | | Output |
|---|---|---|
| $A$ | $B$ | $\overline{A \oplus B}$ |
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

# ADDER

An adder is a digital logic circuit in electronics that implements addition of numbers. In many computers and other types of processors, adders are used to calculate addresses, similar operations and table indices in the ALU and also in other parts of the processors. These can be built for many numerical representations like excess-3 or binary coded decimal.

## Half Adder

The half adder adds two single binary digits A and B. It has two outputs, sum (S) and carry (C). The carry signal represents an overflow into the next digit of a multi-digit addition. The value of the sum is 2C + S. The simplest half-adder design, pictured below, incorporates an XOR gate for S and an AND gate for C. The Boolean logic for the sum (in this case S) will be **A’B + AB’** whereas for the carry (C) will be **AB**. The half adder adds two input bits and generates a carry and sum, which are the two outputs of a half adder. The input variables of a half adder are called the augend and addend bits. The output variables are the sum and carry.

*Truth Table-*

| **Input** | | **Output** | |
|---|---|---|---|
| $A$ **(Augend)** | $B$ **(Addend)** | $Sum$ **(S)** | $Carry$ **(C)** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

*Simplification-*

| | $\overline{B}$ | $B$ |
|---|---|---|
| $\overline{A}$ | 0 | 1 |
| $A$ | 1 | 0 |

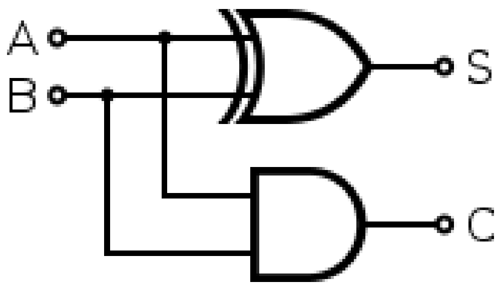
***SUM***

| | $\overline{B}$ | $B$ |
|---|---|---|
| $\overline{A}$ | 0 | 0 |
| $A$ | 0 | 1 |

***CARRY***

It is clear from the above K-Map(s) that we can’t further simplify these equations.

*Logic Diagram-*



## Limitations of Half Adder

The reason these simple binary adders are called Half Adders is that there is no scope for them to add the carry bit from previous bit. This is a major limitation of half adders when used as binary adders especially in real time scenarios which involves addition of multiple bits. To overcome this limitation, full adders are developed.

## Full Adder

The difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs, whereas half adder has only two inputs and two outputs. The first two inputs are A and B and the third input is an input carry as $C_{in}$. A full adder adds binary numbers and accounts for values carried in as well as out. A one-bit full-adder adds three one-bit numbers, often written as A, B, and $C_{in}$ where A and B are the operands, and $C_{in}$ is a bit carried in from the previous less-significant stage. Output carry and sum typically represented by the signals $C_{out}$ and S, where the sum equals $2C_{out} + S$.

A full adder can be constructed from two half adders by connecting A and B to the input of one half adder, then taking its sum-output S as one of the inputs to the second half adder and $C_{in}$ as its other input, and finally the carry outputs from the two half-adders are connected to an OR gate. The sum-output from the second half adder is the final sum output (S) of the full adder and the output from the OR gate is the final carry output ($C_{out}$). The Boolean logic for the sum will be **$S = A \oplus B \oplus C_{in}$** whereas for the carry will be **$C_{out} = (A \cdot B) + \{C_{in} \cdot (A \oplus B)\}$**.

*Truth Table-*

| **Input** | | | **Output** | |
|---|---|---|---|---|
| $A$ **(Augend)** | $B$ **(Addend)** | $C_{in}$ **(Add to previous sum)** | $Sum$ **(S)** | $Carry$ **($C_{out}$)** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

*Simplification-*

| | $\overline{B}\overline{C}_{in}$ | $\overline{B}C_{in}$ | $BC_{in}$ | $B\overline{C}_{in}$ |
|---|---|---|---|---|
| $\overline{A}$ | 0 | 1 | 0 | 1 |
| $A$ | 1 | 0 | 1 | 0 |

***SUM***

| | $\overline{B}\overline{C}_{in}$ | $\overline{B}C_{in}$ | $BC_{in}$ | $B\overline{C}_{in}$ |
|---|---|---|---|---|
| $\overline{A}$ | 0 | 0 | 1 | 0 |
| $A$ | 0 | 1 | 1 | 1 |

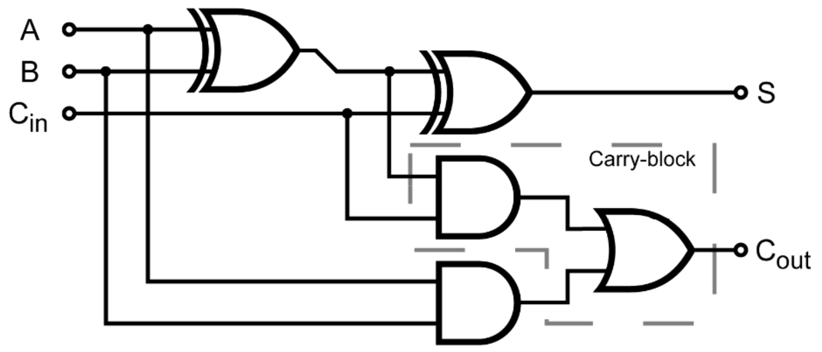
***CARRY***

It is clear from the above K-Map(s) that we can further simplify Carry part but not Sum.

So, **Sum** = $A\overline{B}\overline{C_{in}} + \overline{A}\overline{B}C_{in} + \overline{A}B\overline{C}_{in} + ABC_{in}$ or **Sum = $A \oplus B \oplus C_{in}$**,

**Carry** = $AB + AC_{in} + BC_{in}$ or **Carry = $(A \cdot B) + \{C_{in} \cdot (A \oplus B)\}$**.

*Logic Diagram-*



# SUBTRACTOR

A subtractor is a combinational circuit that performs subtraction of two bits. A subtractor can be designed using the same approach as that of an adder. As with an adder, in the general case of calculations on multi-bit numbers, three bits are involved in performing the subtraction for each bit of the difference: the minuend $\boldsymbol{X}$ subtrahend $\boldsymbol{Y}$ and a borrow in from the previous (less significant) bit order position $\boldsymbol{B}$. The outputs are the difference bit $\boldsymbol{D}$ and borrow bit $\boldsymbol{B_{out}}$.

## Half Subtractor

The half subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, the minuend $\boldsymbol{X}$ and subtrahend $\boldsymbol{Y}$ and two outputs the difference $\boldsymbol{D}$ and borrow out $\boldsymbol{B_{out}}$. The borrow out signal is set when the subtractor needs to borrow from the next digit in a multi-digit subtraction. The Boolean logic for the half subtractor will be **D = X ⊕ Y** whereas for the carry will be $\mathbf{B_{out}} = \overline{X}$ **. Y**.

*Truth Table-*

| **Input** | | **Output** | |
|---|---|---|---|
| $X$ **(Minuend)** | $Y$ **(Subtrahend)** | $Difference$ **(D)** | $Borrow$ **($B_{out}$)** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 0 |

*Simplification-*

| | $\overline{Y}$ | $Y$ |
|---|---|---|
| $\overline{X}$ | 0 | 1 |
| $X$ | 1 | 0 |

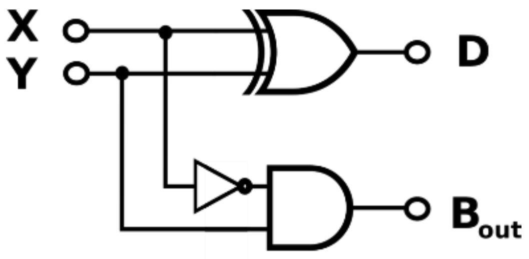
***DIFFERENCE***

| | $\overline{Y}$ | $Y$ |
|---|---|---|
| $\overline{X}$ | 0 | 1 |
| $X$ | 0 | 0 |

***BORROW***

It is clear from the above K-Map(s) that we can't further simplify these equations.

*Logic Diagram-*



## Limitations of Half Subtractor

Half Subtractors do not take into account "Borrow-in" from the previous circuit when we perform the subtraction of multiple number of bits.

This is a major drawback of half subtractors because real time scenarios involve subtracting the multiple number of bits which can not be accomplished using half subtractors.

To overcome this drawback, Full Subtractor comes into play.

## Full Subtractor

When there is a situation where the minuend and subtrahend number contains more significant bit, then the borrow bit which is obtained from the subtraction of 2-bit binary digits is subtracted from the next higher order pair of bits. In such situation, the subtraction involves the operation of 3 bits. Such situation of subtraction can't handle by a simple half subtractor. So, combining two half subtractor we can form another combinational circuit which can perform this type of operation. This circuit is known as the full subtractor.

So we can define full subtractor as a combinational circuit which takes three inputs and produces two outputs difference and borrow. Below is the truth table of the full subtractor, we have used three input variables X, Y and $B_{in}$ which refers to the term minuend, subtrahend and borrow bit respectively. The two outputs difference and borrow are named as D and $B_{out}$ respectively.

*Truth Table-*

| Input | | | Output | |
|---|---|---|---|---|
| $X$ (Minuend) | $Y$ (Subtrahend) | $B_{in}$ (Previous Borrow) | $Difference$ (D) | $Borrow$ ($B_{out}$) |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

*Simplification-*

| | $\overline{Y}\overline{B}_{in}$ | $\overline{Y}B_{in}$ | $YB_{in}$ | $Y\overline{B}_{in}$ |
|---|---|---|---|---|
| $\overline{X}$ | 0 | 1 | 0 | 1 |
| $X$ | 1 | 0 | 1 | 0 |

***DIFFERENCE***

| | $\overline{Y}\overline{B}_{in}$ | $\overline{Y}B_{in}$ | $YB_{in}$ | $Y\overline{B}_{in}$ |
|---|---|---|---|---|
| $\overline{X}$ | 0 | 1 | 1 | 1 |
| $X$ | 0 | 0 | 1 | 0 |

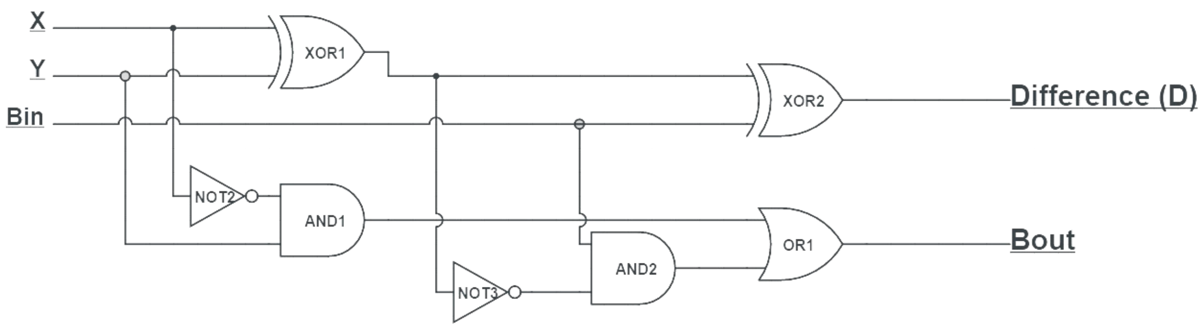
***BORROW***

It is clear from the above K-Map(s) that we can further simplify Borrow part but not Difference.

So, **Difference =** $X\overline{Y}\overline{B_{in}} + \overline{X}\overline{Y}B_{in} + \overline{X}Y\overline{B}_{in} + XYB_{in}$ or **Difference =** X ⊕ Y ⊕ $B_{in}$,

**Borrow =** $YB_{in} + \overline{X}B_{in} + \overline{X}Y$ or **Borrow =** $B_{in}(\overline{X} + Y) + \overline{X}Y$.

*Logic Diagram-*



# DECODER

The name "Decoder" means to translate or decode coded information from one format into another, so a binary decoder transforms "**n**" binary input signals into an equivalent code using "**$2^n$**" outputs.

Binary Decoders are another type of digital logic device that has inputs of 2-bit, 3-bit or 4-bit codes depending upon the number of data input lines, so a decoder that has a set of two or more bits will be defined as having an n-bit code, and therefore it will be possible to represent $2^n$ possible values. Thus, a decoder generally decodes a binary value into a non-binary one by setting exactly one of its n outputs to logic "1".

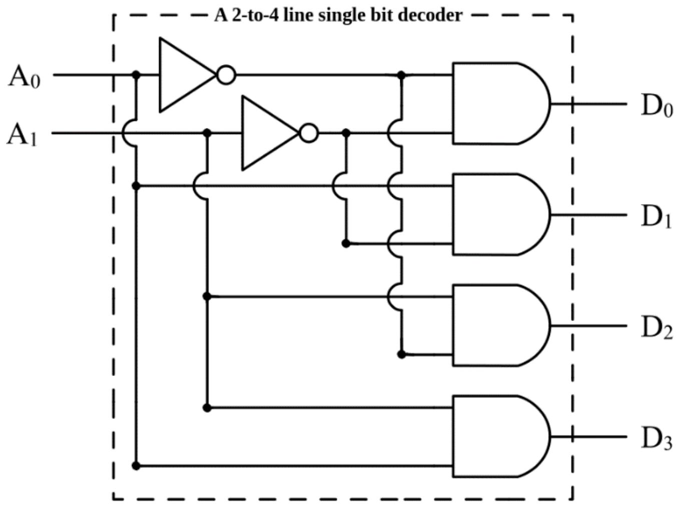
If a binary decoder receives n inputs (usually grouped as a single Binary or Boolean number) it activates one and only one of its $2^n$ outputs based on that input with all other outputs deactivated.

An **example** of a 2-to-4 line decoder along with its truth table is given below:

*Truth Table-*

| **Input** | | **Output** | | | | **Minterms** |
|---|---|---|---|---|---|---|
| **$A_1$** | **$A_0$** | **$D_0$** | **$D_1$** | **$D_2$** | **$D_3$** | |
| 0 | 0 | 1 | 0 | 0 | 0 | $\boldsymbol{D_0} = \bar{A}_1 . \bar{A}_0$ |
| 0 | 1 | 0 | 1 | 0 | 0 | $\boldsymbol{D_1} = \bar{A}_1 . A_0$ |
| 1 | 0 | 0 | 0 | 1 | 0 | $\boldsymbol{D_2} = A_1 . \bar{A}_0$ |
| 1 | 1 | 0 | 0 | 0 | 1 | $\boldsymbol{D_3} = A_1 . A_0$ |

*Logic Diagram-*



This simple example above of a 2-to-4 line binary decoder consists of an array of four AND gates. The 2 binary inputs labelled $A_1$ and $A_0$ are decoded into one of 4 outputs, hence the description of 2-to-4 binary decoder. Each output represents one of the minterms of the 2 input variables, (each output = a minterm).

The binary inputs $A_1$ and $A_0$ determine which output line from $D_0$ to $D_3$ is "HIGH or TRUE" at logic level "1" while the remaining outputs are held "LOW or FALSE" at logic "0" so only one output can be active at any one time. Therefore, whichever output line is "HIGH" identifies the binary code present at the input, in other words it "de-codes" the binary input.

# ENCODER

A binary encoder has 2n input lines and n output lines, hence it encodes the information from $2^n$ inputs into an n-bit code. Depending on the number of input lines, digital or binary encoders produce the output codes in the form of 2 or 3 or 4 bit codes.

These are mainly used to reduce the number of bits needed to represent given information. In digital systems, encoders are used for transmitting the information. Thus the transmission link uses fewer lines to transmit the encoded information.

An **example** of a 4-to-2 line decoder along with its truth table is given as:

The truth table consists of four rows, since, it is assumed that only one input is true at a Moment, having value of 1 then the corresponding binary code associated with that enabled input is displayed at the outputs.

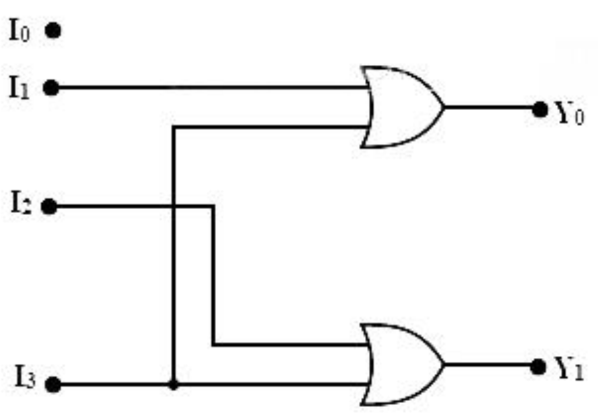
*Truth Table-*

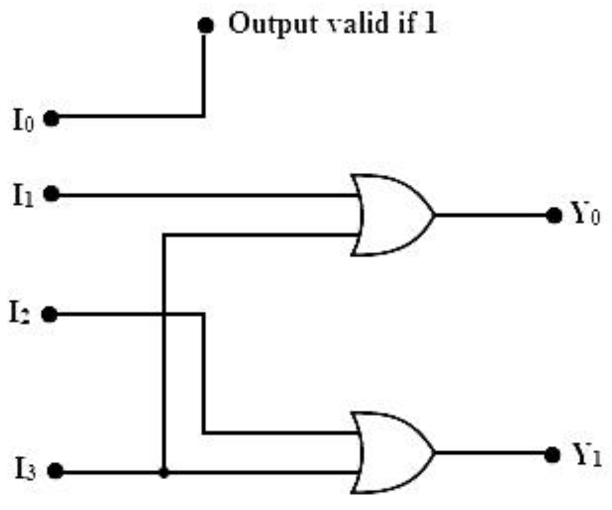
| **Input** | | | | **Output** | |
|---|---|---|---|---|---|
| **$I_3$** | **$I_2$** | **$I_1$** | **$I_0$** | **$Y_1$** | **$Y_0$** |
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

It is to be observed from the table is the output $Y_0$ is 1 when either input $I_1$ or $I_3$ is true, also the output Y1 is set to 1 when either input $I_2$ or $I_3$ is true or We can say $\mathbf{Y_0} = I_1 + I_3$ & $\mathbf{Y_1} = I_2 + I_3$.

The output from 4-to-2 encoder is generated by the logic circuit implemented by a set of OR gates as shown below. In the figure below, the output of the encoder is same if the input activated is the $I_o$ input ($I_o = 1$) or if no input is activated i.e., all the inputs are zero. This causes ambiguity in the encoding output. To avoid this ambiguity, a valid encode output can be added as an additional output, so this output assumes a value 1 when $I_o$ is equal to 1.

*Logic Diagram-*





This simple example above of a 4-to-2 line binary encoder consists of an array of four OR gates. The 4 inputs labelled $I_0$ to $I_3$ are encoded to one of 2 Binary outputs, hence the description of 4-to-2 binary encoder.

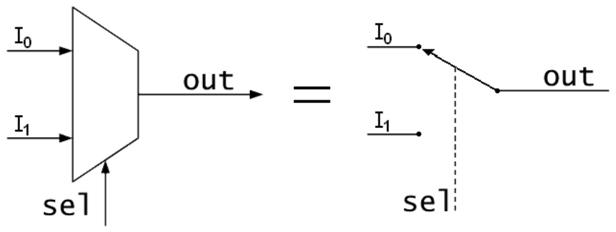
# MULTIPLEXER

Multiplexing is the generic term used to describe the operation of sending one or more analogue or digital signals over a common transmission line at different times or speeds and as such, the device we use to do just that is called a Multiplexer.

A **Multiplexer** (or **MUX**) is a device that selects between several analog or digital input signals and forwards it to a single output line.

A multiplexer of $2^n$ inputs has n select lines, which are used to select which input line to send to the output. Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. A multiplexer is also called a data selector. Multiplexers can also be used to implement Boolean functions of multiple variables.

The **schematic symbol** for a multiplexer is an isosceles trapezoid with the longer parallel side containing the input pins and the short parallel side containing the output pins. The schematic below shows a 2-to-1 multiplexer on the left and an equivalent switch on the right. The `sel` wire connects the desired input to the output.



The selector wires are of digital value. In the case of a 2-to-1 multiplexer, a logic value of 0 would connect $I_0$ to the output while a logic value of 1 would connect $I_1$ to the output.

A **2-to-1** multiplexer has a boolean equation where **A** and **B** are the two inputs, **X** is the selector input, and **Y** is the output:

$$\boldsymbol{Y = (A\,.\,\overline{X}) + (B\,.\,X)}$$

which can be expressed as Truth Table:

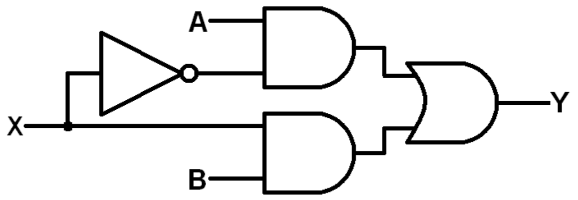
| Selection Line Value (X) | Input A | Input B | Output (Y) |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

or Simply:

| X | Y |
|---|---|
| 0 | A |
| 1 | B |

These tables show that when **X=0** then **Y=A** but when **X=1** then **Y=B**.

*Logic Diagram for a 2 to 1 MUX-*



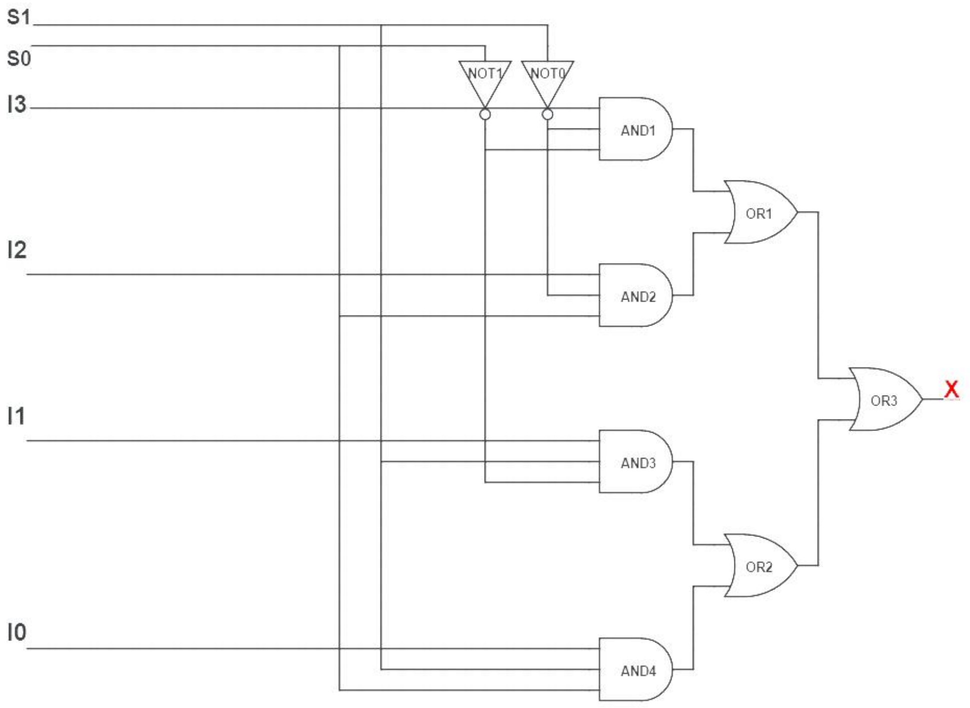
A **4-to-1** multiplexer has a boolean equation where **I₀, I₁, I₂, I₃** are the four inputs, **S₀** & **S₁** is the selector input, and **X** is the output:

$$X = (I_0 . \overline{S_0} . \overline{S_1}) + (I_1 . S_0 . \overline{S_1}) + (I_2 . \overline{S_0} . S_1) + (I_3 . S_0 . S_1)$$

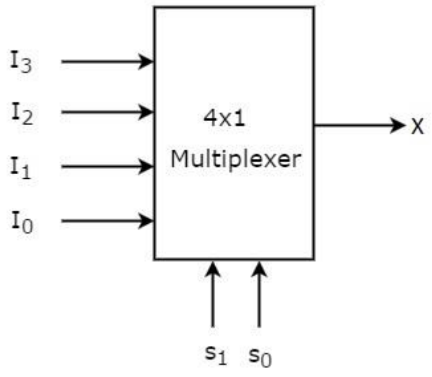
*Truth Table-*

| **Selection Line $S_1$** | **Selection Line $S_0$** | **Output X** |
|---|---|---|
| 0 | 0 | $I_0$ |
| 0 | 1 | $I_1$ |
| 1 | 0 | $I_2$ |
| 1 | 1 | $I_3$ |

*Logic Diagram for a 4 to 1 MUX-*



*Block Diagram-*

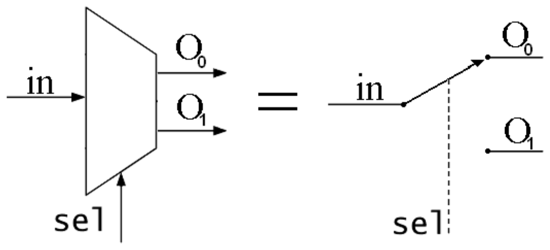


# DEMULTIPLEXER

A **Demultiplexer** (or **DMUX**) is a device that takes a single input line and routes it to one of several digital output lines. A demultiplexer of $2^n$ outputs has n select lines, which are used to select which output line to send the input. A demultiplexer is also called a data distributor.

Demultiplexers can be used to implement general purpose logic. By setting the input to true, the demux behaves as a decoder. The Demultiplexer is the reverse of Digital Multiplexer.

The **schematic symbol** for a demultiplexer is an isosceles trapezoid with the longer parallel side containing the input pins and the short parallel side containing the output pins. The schematic below shows a 1-to-2 demultiplexer on the left and an equivalent switch on the right. The `sel` wire connects the desired input to the output.



The selector wires are of digital value. In the case of a 1-to-2 demultiplexer, a logic value of 0 would connect Input to the output $O_0$ while a logic value of 1 would connect Input to the output $O_2$.

A **1-to-2** demultiplexer has a boolean equation where **D** is the input, **S** is the selector input, and **$Y_0$** & **$Y_1$** are the outputs:

$$Y_0 = (D.\overline{S})$$

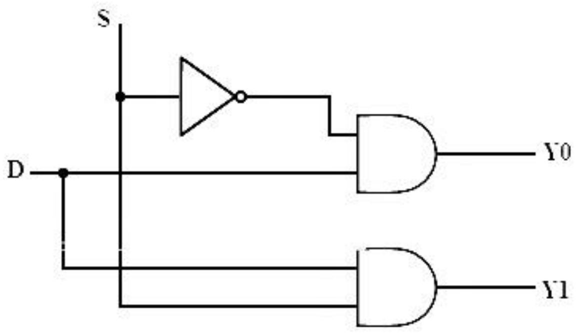
$$Y_1 = (D.S)$$

*Truth Table-*

| Selection Line Value (S) | Input (D) | Output |
|---|---|---|
| 0 | 1 | $Y_0$ |
| 1 | 1 | $Y_1$ |

Above table shows that when **S=0** then output is **$Y_0$** but when **S=1** then output is **$Y_1$**.

*Logic Diagram for a 1 to 2 DEMUX-*



A **1-to-4** demultiplexer has a boolean equation where **D** is the input, **$S_0$** & **$S_1$** are the selector inputs, and **$Y_0$**, **$Y_1$, $Y_2$** & **$Y_3$** are the outputs:

$$Y_0 = ( D . \overline{S_0} . \overline{S_1} )$$

$$Y_1 = ( D . S_0 . \overline{S_1} )$$

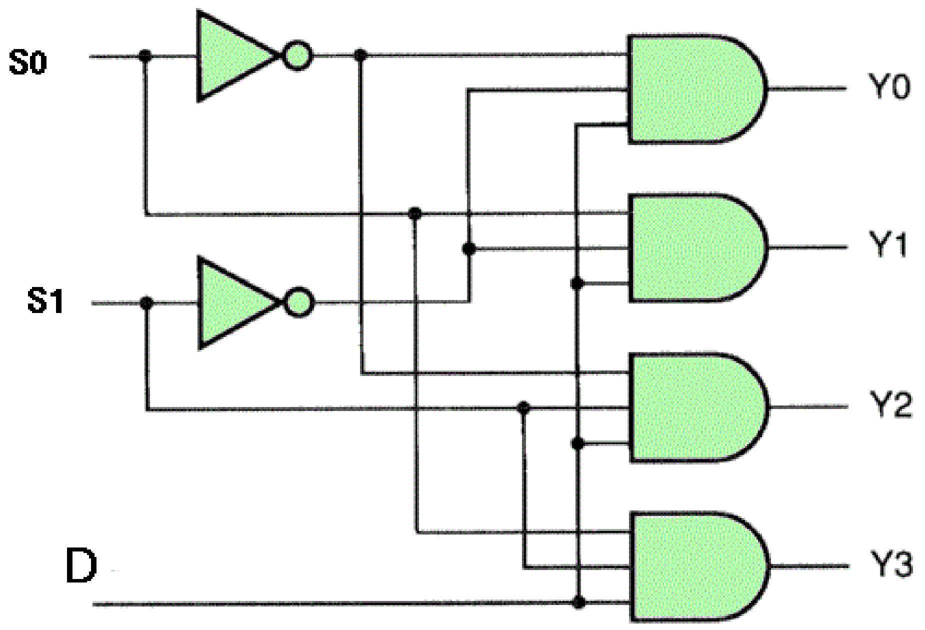
$$Y_2 = ( D . \overline{S_0} . S_1 )$$

$$Y_3 = ( D . S_0 . S_1 )$$

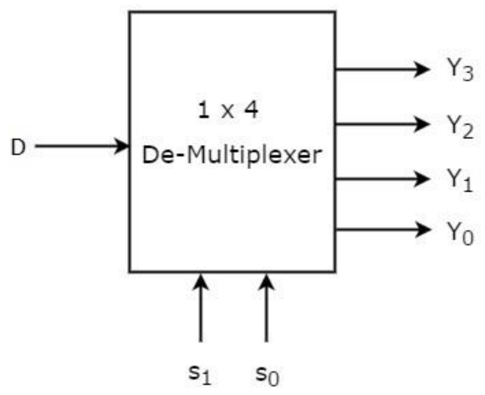
*Truth Table-*

| $S_1$ | $S_0$ | Input (D) | Output |
|---|---|---|---|
| 0 | 0 | 1 | $Y_0$ |
| 0 | 1 | 1 | $Y_1$ |
| 1 | 0 | 1 | $Y_2$ |
| 1 | 1 | 1 | $Y_3$ |

*Logic Diagram for a 1 to 4 DEMUX-*



*Block Diagram-*



# FLIP-FLOP

A Digital computer needs devices which can store information. A **Flip-Flop** is a Binary storage Device. It can store binary bit either 0 or 1. It has two stable states *HIGH* & *LOW* i.e. 1 & 0. It has the property to remain in one state indefinitely until it is directed by an input signal to switch over to the other state. It is also called **Bistable Multivibrator**.

Flip-flops are used as data storage elements. Such data storage can be used for storage of state, and such a circuit is described as sequential logic in electronics. When used in a finite-state machine, the output and next state depend not only on its current input, but also on its current state (and hence, previous inputs). It can also be used for counting of pulses, and for synchronizing variably-timed input signals to some reference timing signal. The term "Flip-flop" relates to the actual operation of the device, as it can be "flipped" into one logic Set state or "flopped" back into the opposing logic Reset state.

Flip-flops can be either level-triggered (asynchronous, transparent or opaque) or edge-triggered (synchronous, or clocked).

## Flip – Flop types –

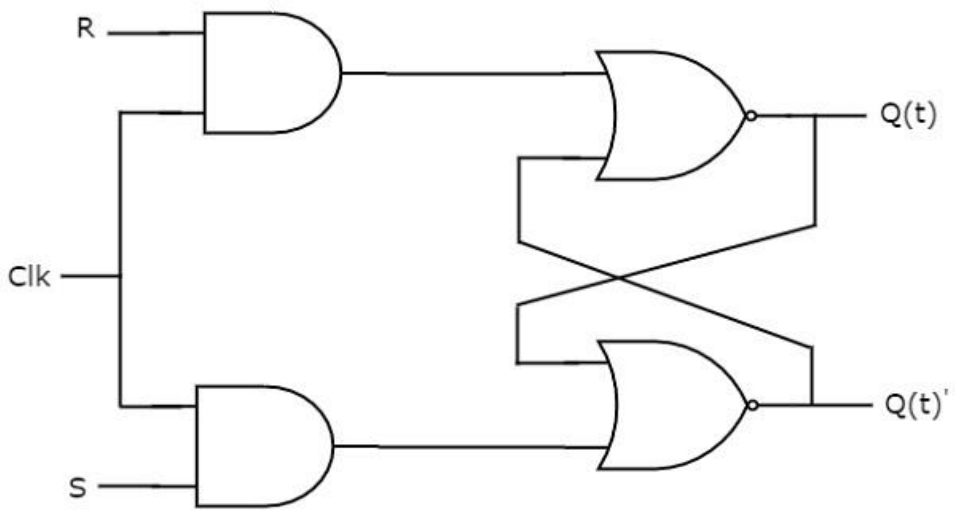
Flip-flops can be divided into common types: the **SR** ("set-reset"), **D** ("data" or "delay"), **T** ("toggle"), and **JK**. The behaviour of a particular type can be described by what is termed the characteristic equation, which derives the "next" (i.e., after the next clock pulse) output, **$Q_{next}$/ Q(t)' / $\overline{Q}$** in terms of the input signal(s) and/or the current output, **Q**.

### 1. SR Flip-Flop

The SR flip-flop, can be considered as one of the most basic sequential logic circuit possible. This simple flip-flop is basically a one-bit memory bistable device that has two inputs, one which will "SET" the device (meaning the output = "1"), and is labelled **S** and one which will "RESET" the device (meaning the output = "0"), labelled **R**.

A basic NAND gate SR flip-flop circuit provides feedback from both of its outputs back to its opposing inputs and is commonly used in memory circuits to store a single data bit. Then the SR flip-flop actually has three inputs, Set, Reset and its current output Q relating to it's current state or history.

In the clocked R-S Flip-Flop the Appropriate levels applied to their inputs are blocked till the receipt of a pulse from another source called Clock. The Flip-Flop changes state only when clock pulse is applied depending upon the inputs. The basic circuit is shown below-



The following table shows the **state table** of SR flip-flop-

| S | R | Q (t+1) |
|---|---|---|
| 0 | 0 | Q(t) |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | - |

Here, **Q(t)** & **Q(t)’** are present state & next state respectively. So, SR flip-flop can be used for one of these three functions such as Hold, Reset & Set based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of SR flip-flop-

| Present Inputs | | Present State | Next State |
|---|---|---|---|
| **S** | **R** | **Q(t)** | **Q (t+1)** |
| 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 | X |
| 1 | 1 | 1 | X |

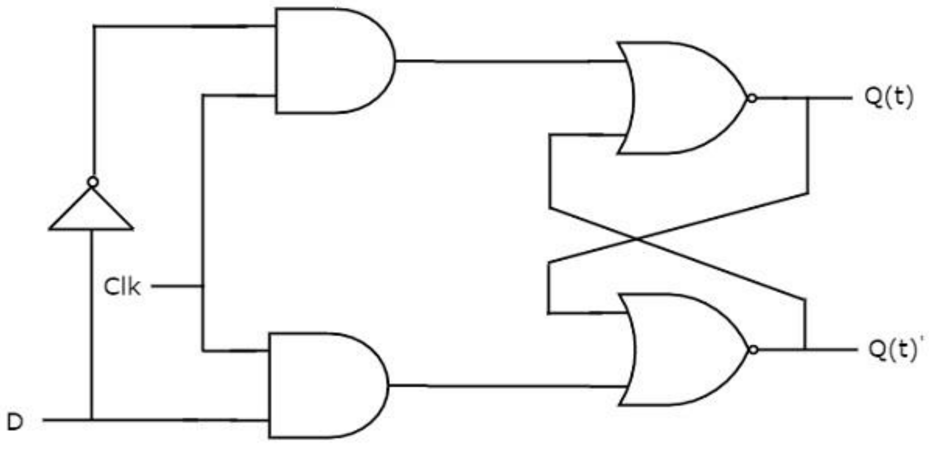
By using three variable K-Map, we can get the simplified expression for next state Q(t+1). The three variable K-Map for next state, Q(t+1) is shown in the below figure-

| | $\overline{RQ}(t)$ | $\overline{R}Q(t)$ | $RQ(t)$ | $R\overline{Q(t)}$ |
|---|---|---|---|---|
| $\overline{S}$ | 0 | 1 | 0 | 0 |
| $S$ | 1 | 1 | x | x |

The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the simplified expression for next state is $\boldsymbol{Q(t+1) = S + R'Q(t)}$.

## 2. D Flip- Flop

A **D** type (Data or Delay) Flip-Flop has a Single input in Addition to the Clock input & It operates with only positive clock transitions or negative clock transitions. That means, the output of D flip-flop is insensitive to the changes in the input, D except for active transition of the clock signal. The circuit diagram of D flip-flop is as shown in figure below-



The following table shows the **state table** of D flip-flop-

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

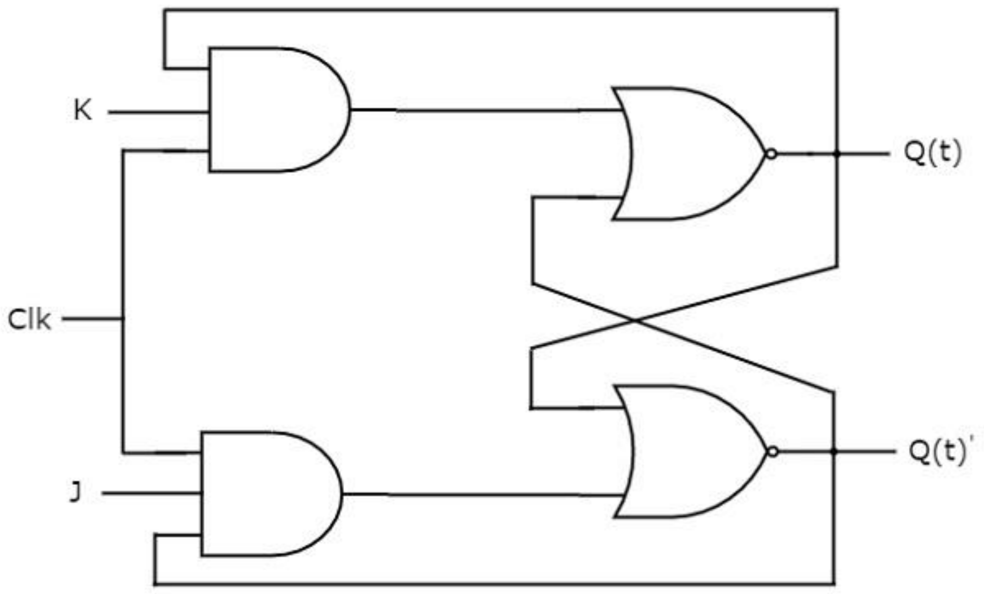
Therefore, D flip-flop always Hold the information, which is available on data input, D of earlier positive transition of clock signal. From the above state table, we can directly write the next state equation as

**Q(t+1) = D**

Next state of D flip-flop is always equal to data input, D for every positive transition of the clock signal. Hence, D flip-flops can be used in registers, shift registers and some of the counters.

## 3. JK Flip-Flop

JK flip-flop is the modified version of SR flip-flop. It operates with only positive clock transitions or negative clock transitions. The circuit diagram of JK flip-flop is shown in the below figure-



This circuit has two inputs J & K and two outputs Q(t) & Q(t)'. The operation of JK flip-flop is similar to SR flip-flop. Here, we considered the inputs of SR flip-flop as **S = JQ(t)**' and **R = KQ(t)** in order to utilize the modified SR flip-flop for 4 combinations of inputs.

The following table shows the **state table** of JK flip-flop-

| J | K | Q(t+1) |
|---|---|---|
| 0 | 0 | Q(t) |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | Q(t)' |

Here, Q(t) & Q(t)' are present state & next state respectively. So, JK flip-flop can be used for one of these four functions such as Hold, Reset, Set & Complement of present state based on the input conditions, when positive transition of clock signal is applied.

The following table shows the **characteristic table** of JK flip-flop-

| Present Inputs | | Present State | Next State |
|---|---|---|---|
| J | K | Q(t) | Q(t+1) |
| 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 | 0 |

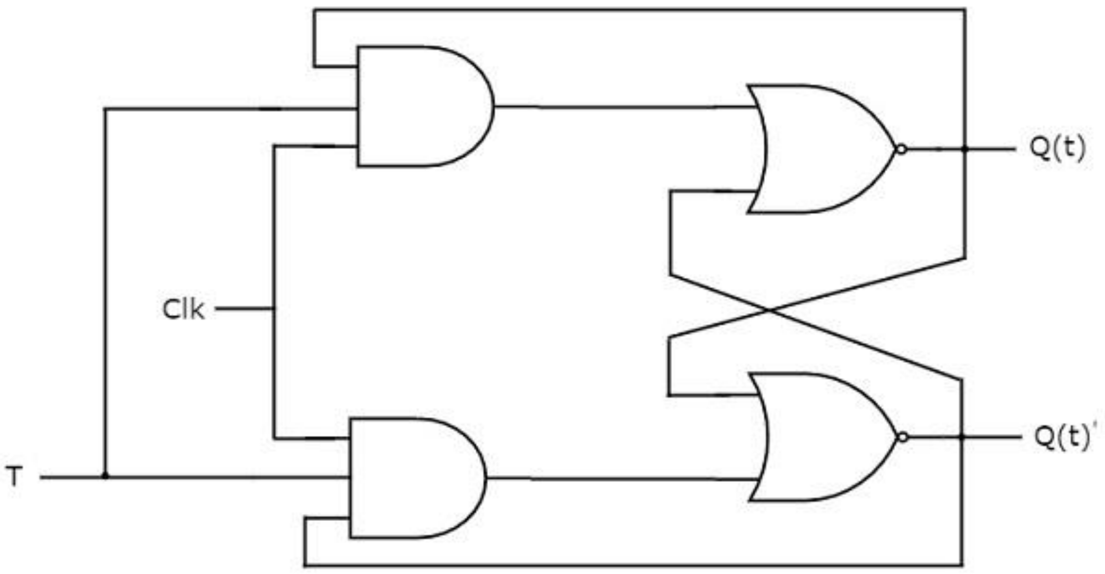
By using three variable K-Map, we can get the simplified expression for next state, Q(t+1). Three variable K-Map for next state, Q(t+1) is shown in the next figure-

| | $\overline{KQ}(t)$ | $\bar{K}Q(t)$ | $KQ(t)$ | $K\overline{Q(t)}$ |
|---|---|---|---|---|
| $\bar{J}$ | 0 | 1 | 0 | 0 |
| $J$ | 1 | 1 | 0 | 1 |

The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the simplified expression for next state is $\boldsymbol{Q(t+1) = JQ(t)' + K'Q(t)}$.

## 4. T Flip-Flop

T flip-flop is the simplified version of JK flip-flop. It is obtained by connecting the same input 'T' to both inputs of JK flip-flop. It operates with only positive clock transitions or negative clock transitions. The circuit diagram of T flip-flop is shown in the below figure-



This circuit has single input T and two outputs Q(t) & Q(t)'. The operation of T flip-flop is same as that of JK flip-flop. Here, we considered the inputs of JK flip-flop as **J = T** and **K = T'** in order to utilize the modified JK flip-flop for 2 combinations of inputs. So, we eliminated the other two combinations of J & K, for which those two values are complement to each other in T flip-flop.

The following table shows the **state table** of T flip-flop-

| **D** | **Q(t+1)** |
|---|---|
| 0 | Q(t) |
| 1 | Q(t)' |

Here, Q(t) & Q(t+1) are present state & next state respectively. So, T flip-flop can be used for one of these two functions such as Hold, & Complement of present state based on the input conditions, when positive transition of clock signal is applied.

The following table shows the **characteristic table** of T flip-flop-

| **Inputs** | **Present State** | **Next State** |
|---|---|---|
| **T** | **Q(t)** | **Q(t+1)** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

From the above characteristic table, we can directly write the next state equation as

$$\boldsymbol{Q(t+1) = T'Q(t) + TQ(t)'}$$

$$\Longrightarrow \boldsymbol{Q(t+1) = T \oplus Q(t)}$$

The output of T flip-flop always toggles for every positive transition of the clock signal, when input T remains at logic High / True. Hence, T flip-flop can be used in counters.