

**Syllabus Content:****3.3.2 Boolean algebra**

- show understanding of Boolean algebra
- show understanding of De Morgan's Laws
- perform Boolean algebra using De Morgan's Laws
- simplify a logic circuit/expression using Boolean algebra

**3.3.1 Logic gates and circuit design**

- produce truth tables for common logic circuits including half adders and full adders
- derive a truth table for a given logic circuit

**3.3.4 Flip-flops**

- show understanding of how to construct a flip-flop (SR and JK)
- describe the role of flip-flops as data storage elements

**3.3.2 Boolean algebra**

We have met gate logic and combination of gates. Another way of representing gate logic is through Boolean algebra, a way of algebraically representing logic gates. You should have already covered the symbols, below is a quick reminder

Bitwise Operator	NOT( $\bar{A}$ )	AND(.)	OR(+)	XOR( $\oplus$ )	NAND( $\overline{A.B}$ )	NOR( $\overline{A + B}$ )
Description	invert input	where exactly two 1s	where one or more 1s	where exactly one 1	where less than two 1s	where exactly two 0s

**Boolean Operations and Expressions**

"Variable", "Complement", and "Literal" are terms used in Boolean Algebra.

A **variable** is a symbol used to represent a logical quantity. Any single variable can have a

"1" or a "0" value

The **complement** is the inverse of a variable and is indicated by a bar over the variable. **The complement of a variable is not considered as a different variable.**

Every occurrence of a variable or its complement is called a **Literal**. It could be its true form or its complement, both of them are called Literals

A **SUM TERM** is the SUM of literals (**A+B+C+D**)

A sum term is equal to **1** if one or all of its inputs are **1**, and is equal to **0** only if all of its inputs are zero.

A **PRODUCT TERM** is the PRODUCT of literals (**A.B.C.D**)

A product term is equal to **1** if all of its inputs are 1. And is equal to **0** if any one (or all) of its inputs are zero.

### Describing Logic Circuits Algebraically:

Any logic circuit, no matter how complex, may be completely described using the Boolean operations previously defined. Because the **OR** gate, **AND** gate, and **NOT** gate are the basic building blocks of digital circuits.

Boolean algebra provides a concise way to express the operation of a logic circuit formed by a combination of logic gates so that the output can be determined for various combinations of input values.

To derive the Boolean expression for a given logic circuit, begin at the left most inputs and work towards the final output, writing the expression for each gate.

### Laws and Rules of Boolean Algebra:

#### Equivalent and Complement of Boolean Expressions

Two given Boolean expressions are said to be equivalent if one of them equals "1" only when the other also equals "1" and same case with "0"

They are said to be **complement** of each other if one expression equals "1" only when the other equals "0" and vice versa.

#### Postulates of Boolean Algebra:

The following are the important postulates of Boolean algebra

<b>1.</b>	$1.1 = 1$	&	$0+0 = 0$
<b>2.</b>	$1.0 = 0.1 = 0$	&	$0+1 = 1+0$
<b>3.</b>	$0.0 = 0$	&	$1+1 = 1$
<b>4.</b>	$1 = 0$	&	$0 = 1$

## Theorems of Boolean Algebra

Boolean theorems can be useful in simplifying a logic expression. That is, in reducing the number of terms in the expression.

It is useful in the sense that the number of gates, are reduced which in turn also reduces heat dissipation from the circuit (saves energy)

When this is done, the reduced expression will produce a circuit that is less complex than the one which the original expression would have produced.

### Commutative Laws

#### Rule 1:

For Addition:

$$\mathbf{X + Y = Y + X}$$

$$(i) A.B = B.A$$

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

For Multiplication:

$$\mathbf{X.Y = Y.X}$$

### Associative Laws:

#### Rule 2:

For Addition:

$$\mathbf{X + (Y + Z) = Y + (Z + X) = Z + (X + Y)}$$

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

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

For Multiplication

$$\mathbf{X.(Y.Z) = Y.(Z.X) = Z.(X.Y)}$$

### Distributive Laws

#### Rule 3:

$$\mathbf{X.(Y + Z) = X.Y + X.Z}$$

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

$$\mathbf{(X.Y) + (X.Z) = X(Y + Z)}$$

## Operations with '0' and '1'

Rule 4:

### OR Laws:

These laws use the OR operation. Therefore they are called as **OR** laws.

$$0 + X = X$$

$$1 + X = 1$$

$$(i) A + 0 = A$$

$$(ii) A + 1 = 1$$

$$(iii) A + A = A$$

$$(iv) A + \bar{A} = 1$$

### AND Laws:

These laws use the AND operation. Therefore they are called as **AND** laws.

$$0 \cdot X = 0$$

$$1 \cdot X = X$$

$$(i) A \cdot 0 = 0$$

$$(ii) A \cdot 1 = A$$

$$(iii) A \cdot A = A$$

$$(iv) A \cdot \bar{A} = 0$$

### Idempotent or Identity Laws:

RULE 5:

$$X \cdot X \cdot X \cdot X \dots X = X$$

RULE 7:

$$X + X + X + X + \dots + X = X$$

### Complementation Law:

RULE 6:

$$X \cdot \bar{X} = 0$$

RULE 8:

$$X + \bar{X} = 1$$

### Involution Law / INVERSION law:

RULE 9:

This law uses the NOT operation. The inversion law states that double inversion of a variable result in the original variable itself.

$$\bar{\bar{X}} = X$$

or

$$\bar{\bar{A}} = A$$

### Absorption Law or Redundancy Law:

**RULE 10:**

- $X + X.Y = X$

**RULE 11:**

- $X + \overline{X}.Y = X + Y$

**RULE 12:**

- $(X+Y).(X+Z) = X+Z.Y$

**Explanation of Rule10, Rule 11, Rule12**

**RULE 10**

$A + AB = A$

L.H.S =  $A + AB$

=  $A(1+B)$

=  $A.1$  {Rule 2}

=  $A$  {Rule 4}

= R.H.S

**RULE 11**

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

$A + \overline{A}B = (A + AB) + \overline{A}B$  {Rule 10}

=  $(AA + AB) + \overline{A}B$  {Rule 7}

=  $AA + AB + A\overline{A} + \overline{A}B$  {Rule 8}

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

=  $1.(A+B)$  {Rule 6}

=  $A+B$  {Rule 4}

**RULE 12**

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

$(A + B)(A + C) = AA + AC + BA + BC$

=  $A + AC + BA + BC$  {Rule 7}

=  $A(1+C) + BA + BC$

=  $A.1 + BA + BC$  {Rule 2}

=  $A + BA + BC$  {Rule 4}

=  $A(1 + B) + BC$

=  $A.1 + BC$  {Rule 2}

=  $A + BC$  {Rule 4}

### DeMorgan's Theorem

De Morgan's theorems provide mathematical verification of the equivalency of the NAND and negative-OR gates & the equivalency of the NOR and negative-AND gates.

In [electrical and computer engineering](#), De Morgan's laws are commonly written as:

**Theorem 1:**

The compliment of the product of 2 variables is equal to the sum of the compliments of individual variables

$$\overline{A \cdot B} \equiv \overline{A} + \overline{B}$$

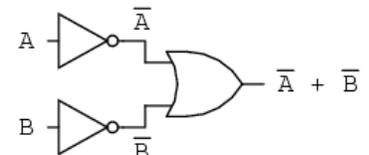
**Theorem 2:**

The compliment of the sum of two variables is equal to the product of the compliment of each variable

$$\overline{A + B} \equiv \overline{A} \cdot \overline{B}$$



... is equivalent to ...

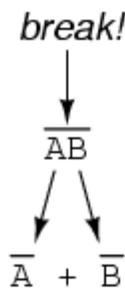


$$\overline{AB} = \overline{A} + \overline{B}$$

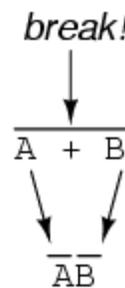
$$(a) \overline{[X_1 + X_2 + X_3 + \dots + X_n]} = \overline{X_1} \cdot \overline{X_2} \cdot \overline{X_3} \cdot \dots \cdot \overline{X_n}$$

$$(b) \overline{[X_1 \cdot X_2 \cdot X_3 \cdot \dots \cdot X_n]} = \overline{X_1} + \overline{X_2} + \overline{X_3} + \dots + \overline{X_n}$$

### DeMorgan's Theorems



NAND to Negative-OR



NOR to Negative-AND

### The rules that govern Boolean algebra

Commutative Laws	$A + B = B + A$	$A \cdot B = B \cdot A$
Associative Laws	$A + (B + C) = (A + B) + C$	$A \cdot (B \cdot C) = (A \cdot B) \cdot C$
Distributive Laws	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$ $(A + B) \cdot (A + C) = A + B \cdot C$	$A + (B \cdot C) = (A + B) \cdot (A + C)$
Tautology/Idempotent Laws	$A \cdot A = A$	$A + A = A$
Tautology/Identity Laws	$1 \cdot A = A$	$0 + A = A$
Tautology/Null Laws	$0 \cdot A = 0$	$1 + A = 1$
Tautology/Inverse Laws	$A \cdot \overline{A} = 0$	$A + \overline{A} = 1$
Absorption Laws	$A \cdot (A + B) = A$ $A + A \cdot B = A$	$A + (A \cdot B) = A$ $A + \overline{A} \cdot B = A + B$
De Morgan's Laws	$\overline{(A \cdot B)} = \overline{A} + \overline{B}$	$\overline{(A + B)} = \overline{A} \cdot \overline{B}$

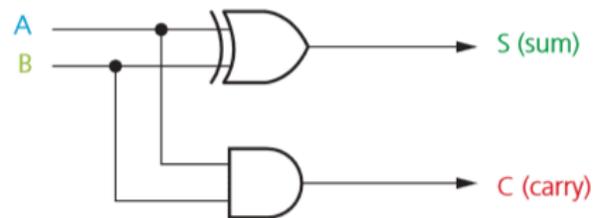
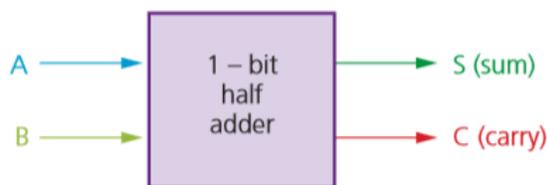
## Syllabus Content:

### 3.3.1 Logic gates and circuit design

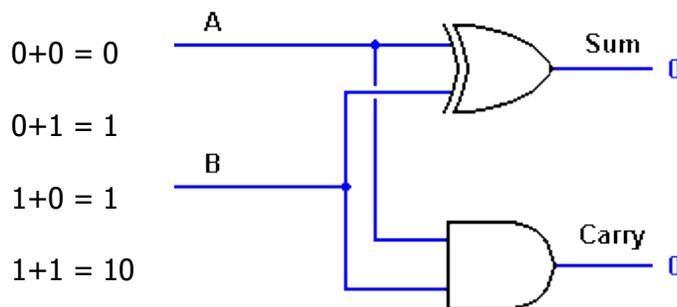
- produce truth tables for common logic circuits including half adders and full adders
- derive a truth table for a given logic circuit

## Half Adder:

The simplest circuit that can be used for binary addition is the half adder. This can be represented by the diagram in the circuit takes two input bits and outputs a sum bit (S) and a carry bit (C).



With the help of half adder, we can design circuits that are capable of performing simple addition with the help of logic gates. Let us first take a look at the addition of single bits.



INPUTS		OUTPUTS	
A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

These are the least possible single-bit combinations. But the result for 1+1 is 10. Though this problem can be solved with the help of an EXOR Gate, if you do care about the output, the sum result must be re-written as a 2-bit output.

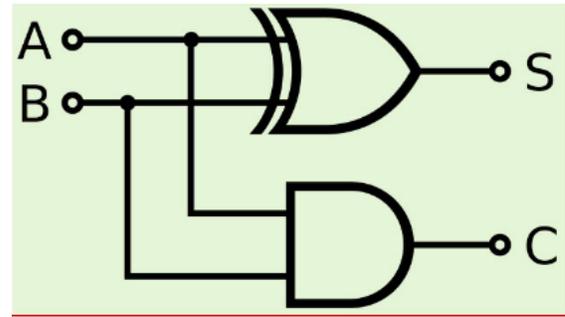
Thus the above equations can be written as

$$0+0 = 00$$

$$0+1 = 01$$

$$1+0 = 01$$

$$1+1 = 10$$



Here the output '1' of '10' becomes the carry-out. The result is shown in a truth-table below. 'SUM' is the normal output and 'CARRY' is the carry-out.

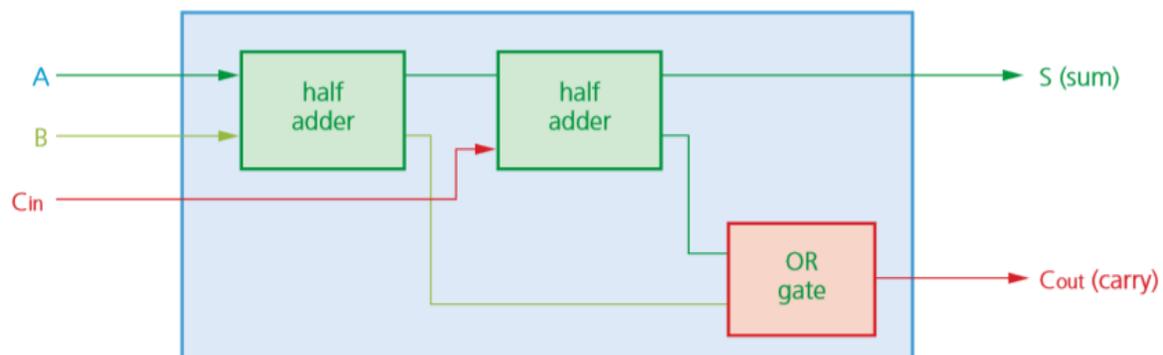
INPUTS		OUTPUTS	
A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

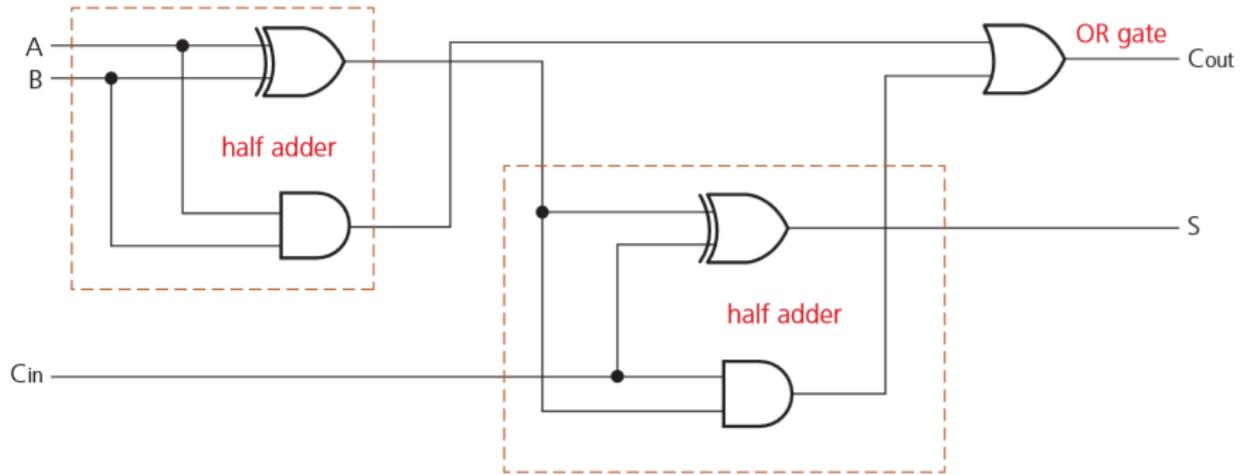
From the equation it is clear that this 1-bit adder can be easily implemented with the help of EXOR Gate for the output 'SUM' and an AND Gate for the carry. Take a look at the implementation below. For complex addition, there may be cases when you have to add two 8-bit bytes together. This can be done only with the help of full-adder logic.

## Full Adder

This type of adder is a little more difficult to implement than a half-adder. The main difference between a half-adder and a full-adder is that the full-adder has three inputs and two outputs. The first two inputs are **A** and **B** and the third input is an input **carry designated as CIN**.

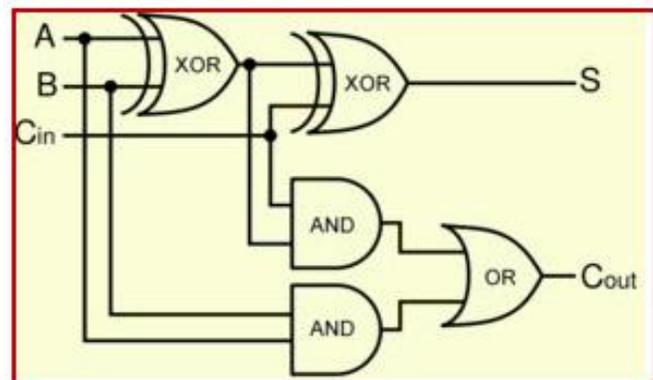
The sum shows how we have to deal with **CARRY** from the previous column. This is why we need to join two half adders together to form a full adder:





The output carry is designated as COUT and the normal output is designated as S. Take a look at the truth-table.

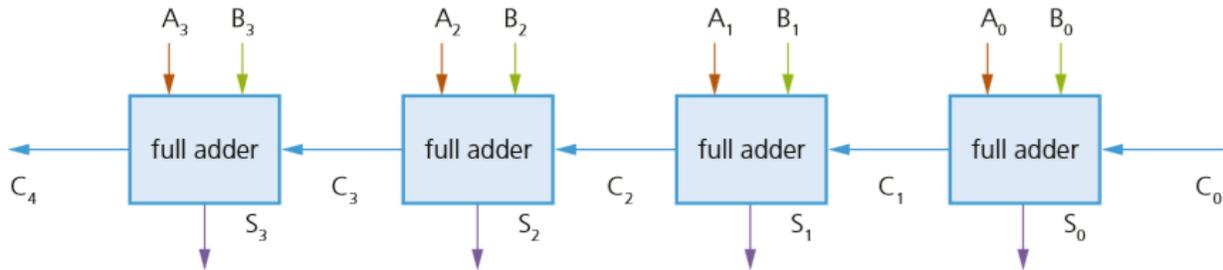
INPUTS			OUTPUTS	
A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



From the above truth-table, the full adder logic can be implemented. We can see that the output S is an EXOR between the input A and the half-adder SUM output with B and CIN inputs. We must also note that the COUT will only be true if any of the two inputs out of the three are HIGH.

Thus, we can implement a full adder circuit with the help of two half adder circuits. The first will half adder will be used to add A and B to produce a partial Sum. The second half adder logic can be used to add CIN to the Sum produced by the first half adder to get the final S output. If any of the half adder logic produces a carry, there will be an output carry. Thus, COUT will be an OR function of the half-adder Carry outputs. Take a look at the implementation of the full adder circuit shown below.

- As with the half adder circuits, different logic gates can be used to produce the full adder circuit.
- The full adder is the basic building block for multiple binary additions. For example, Figure below shows how two 4-bit numbers can be summed using four full adder circuits.



www.majidtahir.com by Majid

## Syllabus Content:

### 3.3.4 Flip-flops

- show understanding of how to construct a flip-flop (SR and JK)
- describe the role of flip-flops as data storage elements

## Flip-Flop Circuits

All of the logic circuits you have encountered up to now are **combination circuits** (the output depends entirely on the input values).

We will now consider a second type of logic circuit, known as a **sequential circuit** (the output depends on the input value produced from a previous output value). Examples of sequential circuits include **flip-flop circuits**. This chapter will consider two types of flip-flops: **SR flip-flops** and **JK flip-flops**.

### KEY TERMS

**Combinational circuit:** a circuit in which the output is dependent only on the input values

**Sequential circuit:** a circuit in which the output depends on the input values and the previous output

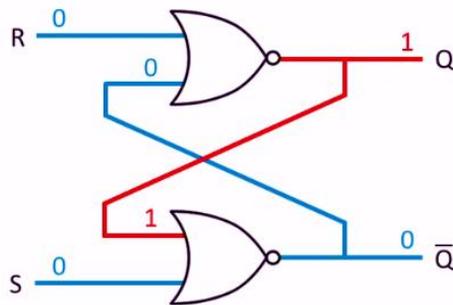
## SR Flip-Flop

The **SR flip-flop**, also known as a **SR Latch**, 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 another which will "RESET" the device (meaning the output = "0"), labelled R.

SR flip-flops consist of two cross-coupled NAND gates (**note: they can equally well be produced from NOR gates**). The two inputs are labelled 'S' and 'R', and the two outputs are labelled 'Q' and 'Q̄' (remember Q is equivalent to NOT Q).

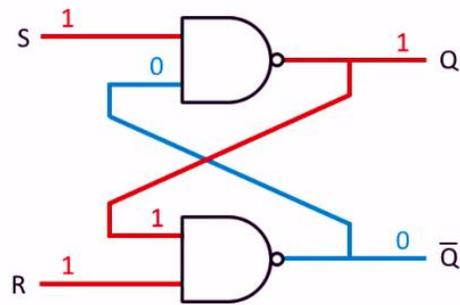
## The Basic SR Flip-flop

We can use **SR flip-flop circuits** constructed from both **NOR gates** or **NAND gates**, as shown in Figure.



S	R	Q	Q̄
0	0	1	0
0	0	0	1
0	1	0	1
1	0	1	0
1	1	0	0

Invalid

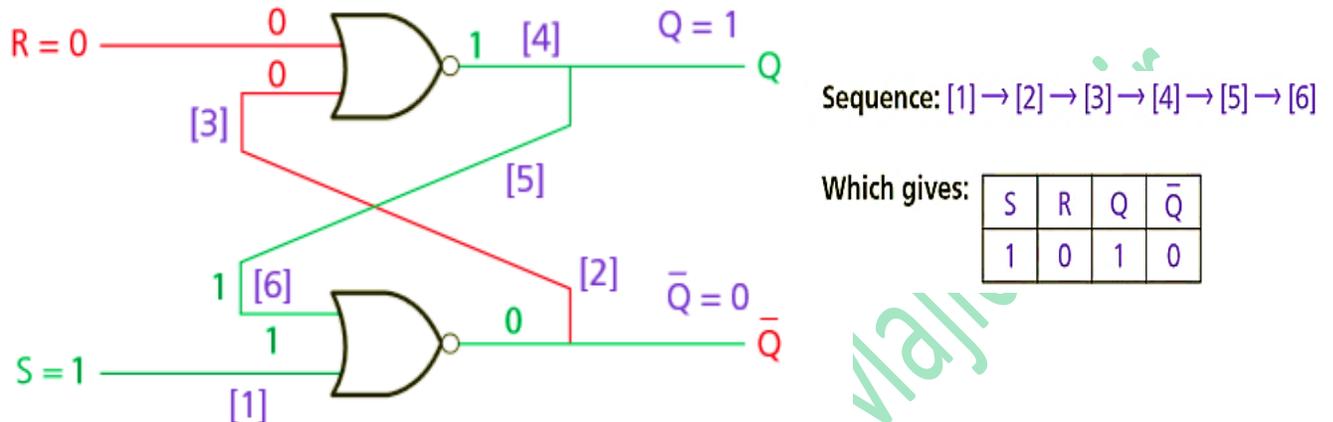


S	R	Q	Q̄
0	0	1	1
0	1	1	0
1	0	0	1
1	1	0	1
		1	0

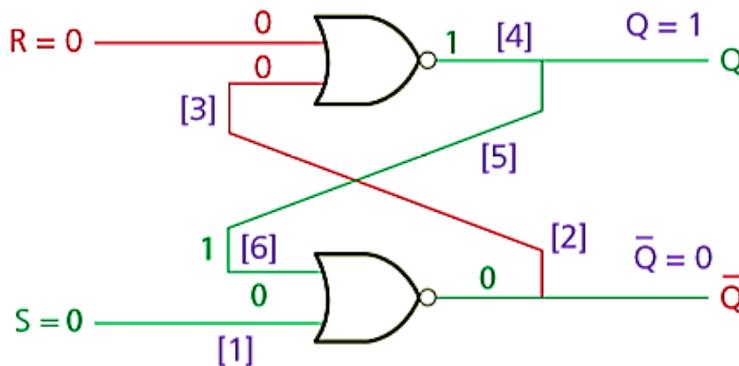
Invalid

### How SR Flip-flop works:

We will consider SR-Flip Flip using NOR Gates. We will now consider the truth table to match our SR flip-flop using the initial states of  $R = 0$ ,  $S = 1$  and  $Q = 1$ . The sequence of the stages in the process is shown in Figure



- We have to start with two inputs given in **red colour**.
- We can take  $R = 0$  and  $\bar{Q} = 0$  in First Gate which produces  $Q = 1$
- When  $Q = 1$  and Set  $S = 1$  is input in second gate in **green colour**, it produces  $\bar{Q} = 0$
- So in SR Flip-Flop when Set value  $Q = 1$ , **Reset value  $\bar{Q} = 0$** .
- If  $S = 0$  and  $R = 0$ , No change. Flip Flop will remain in present state as shown below
- If  $S = 1$  and  $R = 1$ , An invalid condition will happen as  $Q$  and  $\bar{Q}$  both have to be opposite to each other which are same in invalid state.
- Now consider what happens if we change the value of  $S$  from **1 to 0**.
- See when  $R = 0$  and  $S = 0$ , No change in Output, its same as previous state.



S	R	Q	Q'	
0	0	NC	NC	No change. Latch remained in present state.
1	0	1	0	Latch SET.
0	1	0	1	Latch RESET.
1	1	0	0	Invalid condition.

The reader is left to consider the other options which lead to the truth table, Table below, for the flip-flop circuit.

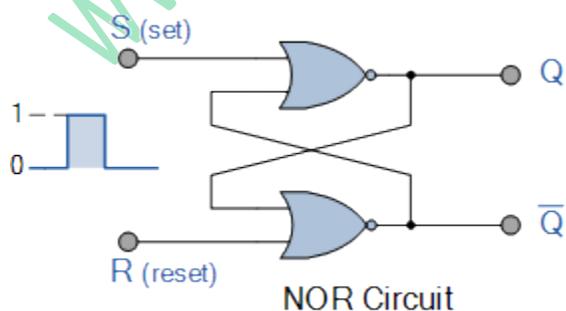
	INPUTS		OUTPUTS		Comment
	S	R	Q	$\bar{Q}$	
(a)	1	0	1	0	
(b)	0	0	1	0	following S = 1 change
(c)	0	1	0	1	
(d)	0	0	0	1	following R = 1 change
(e)	1	1	0	0	

**Explanation:**

-  S = 1, R = 0, Q = 1,  $\bar{Q}$  = 0 is the set state in this example
-  S = 0, R = 0, Q = 1,  $\bar{Q}$  = 0 is the re-set state in this example
-  S = 0, R = 1, Q = 0,  $\bar{Q}$  = 1 here the value of Q in line (b) remembers the value of Q from line (a); the value of Q in line (d) remembers the value of Q in line (c)
-  S = 0, R = 0, Q = 0,  $\bar{Q}$  = 1 R changes from 1 to 0 and has no effect on outputs (these values are remembered from line (c))
-  S = 1, R = 1, Q = 0,  $\bar{Q}$  = 0 Invalid case since  $\bar{Q}$  should be the (opposite) of Q.
-  The truth table shows how an input value of S = 0 and R = 0 causes no change to the two output values; S = 0 and R = 1 reverses the two output values; S = 1 and R = 0 always gives Q = 1 and  $\bar{Q}$  = 0 which is the set value.
-  The truth table shows that SR flip-flops can be used as a storage/memory device for one bit; because a value can be remembered but can also be changed it could be used as a component in a memory device such as a RAM chip.
-  It is important that the fault condition in line (e) is considered when designing and developing storage/memory devices.

**The NOR Gate SR Flip-flop**

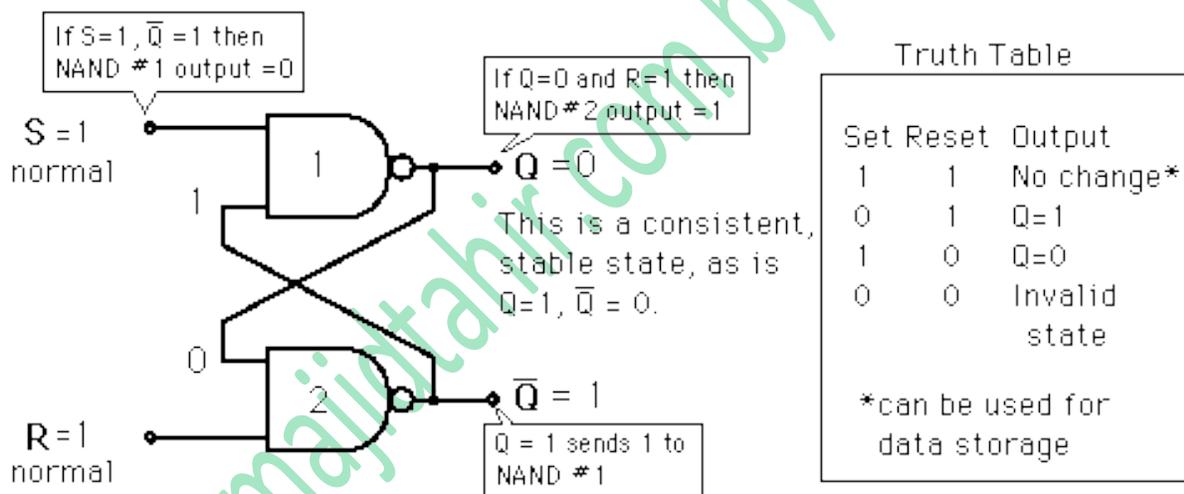
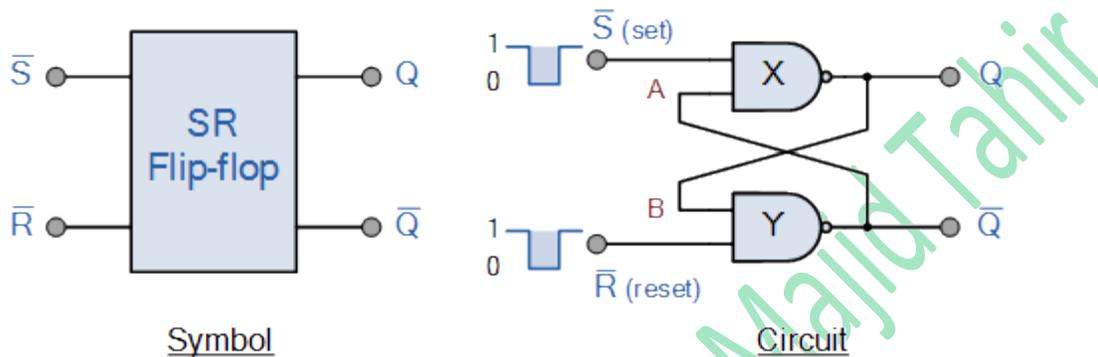
Below are SR flip flops with NOR Gates and NAND Gates along with truth tables, also showing invalid state:



S	R	Q	$\bar{Q}$
0	0	No change	
0	1	1	0
1	0	0	1
1	1	X	X
(Invalid)			

## The NAND Gate SR Flip-Flop

As well as using NOR gates, it's also possible to construct simple one-bit **SR Flip-flops** using two cross-coupled NAND gates connected in the same configuration. The circuit will work in a similar way to the NOR gate circuit above, except that the inputs are active HIGH and the invalid condition exists when both its inputs are at logic level "1", and this is shown below.



The only difference is that in **NOR Gate SR Flip Flop Invalid Condition** happens when **S=1** and **R=1** and **No-Change** happens when **S=0** and **R=0**

In **NAND Gate SR Flip Flop Invalid Condition** happens when **S=0** and **R=0** and **No-Change** happens when **S=1** and **R=1**

The **SR flip-flop** has few problems due to which **JK flip-flop** has been developed.

## The JK flip-flop

The SR flip-flop has the following problems:

-  In addition to the possibility of entering an invalid state there is also the potential for a circuit to arrive in an uncertain state.
-  If inputs do not arrive quite at the same time, the circuit can become unstable.

In order to prevent this, the JK flip-flop has been developed. A circuit may include a clock pulse input to give a better chance of synchronizing inputs and additional gates are added.

The addition of the synchronised input gives four possible input conditions to the JK flip-flop:

-  » 1
-  » 0
-  » no change
-  » toggle (which takes care of the invalid S, R states).

The JK flip-flop can be illustrated by the symbol shown in **Figure** JK flip-flop symbol (left) and JK flip-flop using NAND gates only (right)

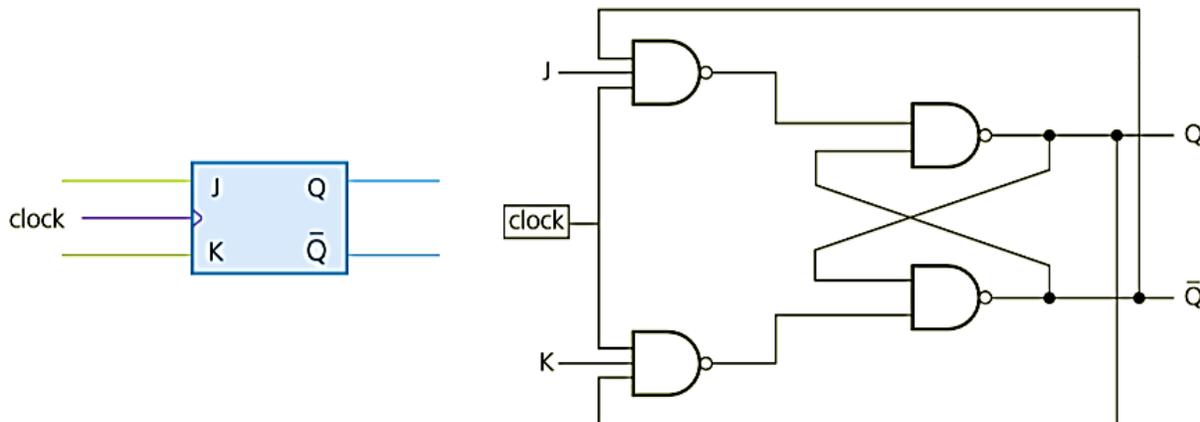


Table below is the simplified truth table for the JK flip-flop.

J	K	Value of Q before clock pulse	Value of Q after clock pulse	OUTPUT
0	0	0	0	Q is unchanged after clock pulse
0	0	1	1	
1	0	0	1	Q = 1
1	0	1	1	
0	1	0	0	Q = 0
0	1	1	0	
1	1	0	1	Q value toggles between 0 and 1
1	1	1	0	

- » When  $J = 0$  and  $K = 0$ , there is no change to the output value of  $Q$ .
- » If the values of  $J$  or  $K$  change, then the value of  $Q$  will be the same as the value of  $J$  ( $Q$  will be the value of  $K$ ).
- » When  $J = 1$  and  $K = 1$ , the  $Q$ -value toggles after each clock pulse, thus preventing illegal states from occurring (in this case, toggle means the flipflop will change from the 'Set' state to the 'Re-set' state or the other way round).

J	K	Clock	Q
0	0	↑	Q unchanged
1	0	↑	1
0	1	↑	0
1	1	↑	Q toggles

Truth table for a JK flip-flop

### Use of JK flip-flops

- » Several JK flip-flops can be used to produce shift registers in a computer.
- » A simple binary counter can be made by linking up several JK flip-flop circuits (this requires the toggle function).

### References:

AS & A level by Silvia Langfield and Dave Duddell  
 Cambridge International AS & A level by David Watson and Hellen Williams (Hodder Education)  
<http://study.com/academy/lesson/how-star-topology-connects-computer-networks-in-organizations.html>  
<https://www.allaboutcircuits.com/textbook/digital/chpt-7/demorgans-theorems/>  
[https://en.wikipedia.org/wiki/De\\_Morgan%27s\\_laws](https://en.wikipedia.org/wiki/De_Morgan%27s_laws)  
<http://www.electronicshub.org/boolean-algebra-laws-and-theorems/>  
[http://www.electronics-tutorials.ws/sequential/seq\\_1.html](http://www.electronics-tutorials.ws/sequential/seq_1.html)  
<https://www.elprocus.com/half-adder-and-full-adder/>