Boolean Algebra

Boolean algebra is used to analyze and simplify the digital (logic) circuits. It uses only the binary numbers i.e. 0 and 1. It is also called as **Binary Algebra** or **logical Algebra**. Boolean algebra was invented by **George Boole** in 1854.

Rule in Boolean Algebra-

Following are the important rules used in Boolean algebra.

- Variable used can have only two values. Binary 1 for HIGH and Binary 0 for LOW.
- Complement of a variable is represented by an over bar (-). Thus, complement of variable B is represented as $\overline{B}$. Thus if B = 0 then $\overline{B}$ = 1 and B = 1 then $\overline{B}$ = 0.
- OR ing of the variables is represented by a plus (+) sign between them. For example ORing of A, B, C is represented as A + B + C.
- Logical ANDing of the two or more variable is represented by writing a dot between them such as A.B.C. Sometime the dot may be omitted like ABC.

Boolean Laws

There are six types of Boolean Laws.

Commutative law

Any binary operation which satisfies the following expression is referred to as commutative operation.

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

Commutative law states that changing the sequence of the variables does not have any effect on the output of a logic circuit.

Associative law

This law states that the order in which the logic operations are performed is irrelevant as their effect is the same.

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

Distributive law

Distributive law states the following condition.

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

AND law

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

(i) $A.0 = 0$ (ii) $A.1 = A$

(iii) $A.A = A$ (iv) $A.\overline{A} = 0$

OR law

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

(i) $A + 0 = A$ (ii) $A + 1 = 1$

(iii) $A + A = A$ (iv) $A + \overline{A} = 1$

INVERSION law

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

$\overline{\overline{A}} = A$

De Morgan's Theorems

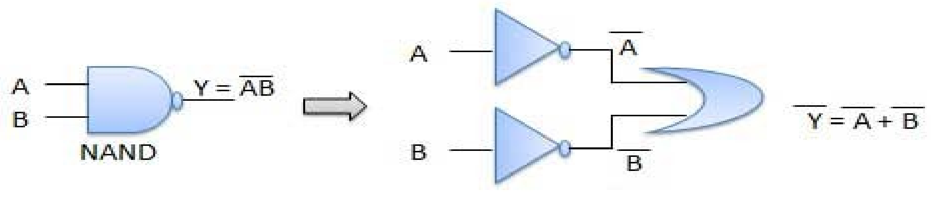
De Morgan has suggested two theorems which are extremely useful in Boolean Algebra. The two theorems are discussed below.

Theorem 1

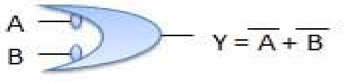
$\overline{A.B} = \overline{A} + \overline{B}$

NAND = Bubbled OR

- The left hand side (LHS) of this theorem represents a NAND gate with inputs A and B, whereas the right hand side (RHS) of the theorem represents an OR gate with inverted inputs.
-
- This OR gate is called as **Bubbles OR**
-
-
-



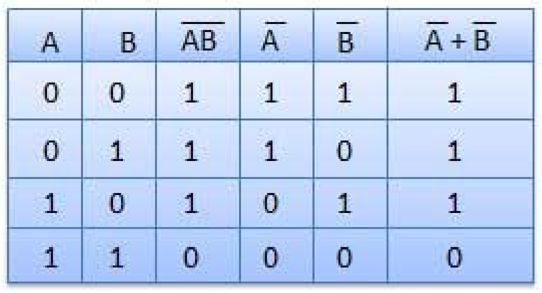
NAND ≡ Bubbled OR



Bubbled OR

-

Table showing verification of the De Morgan's first theorem −



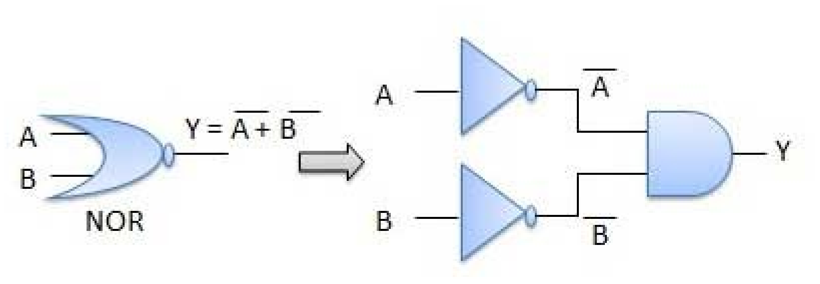
A	B	$\overline{AB}$	$\overline{A}$	$\overline{B}$	$\overline{A}+\overline{B}$
0	0	1	1	1	1
0	1	1	1	0	1
1	0	1	0	1	1
1	1	0	0	0	0

Theorem 2

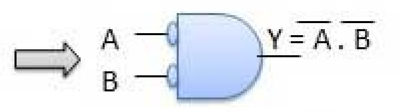
$\overline{A + B} = \overline{A}.\overline{B}$

NOR = Bubbled AND

- The LHS of this theorem represents a NOR gate with inputs A and B, whereas the RHS represents an AND gate with inverted inputs.
- This AND gate is called as **Bubbled AND**.

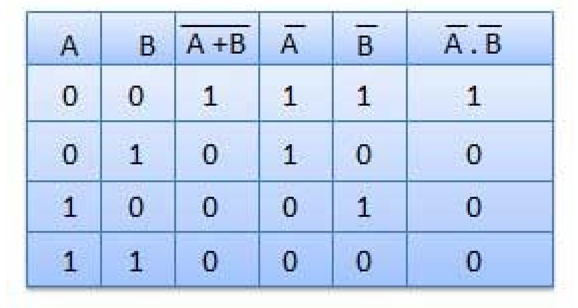


NOR $\equiv$ Bubbled AND



Bubbled AND

Table showing verification of the De Morgan's second theorem −



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

<u>Implementation of Boolean Functions using Logic Gates</u>

Boolean function can be represented easily in SOP (sum of products) form and POS (product of sums) form.

Any Boolean function can be represented by using a number of logic gates by interconnecting them. Logic gates implementation or logic representation of Boolean functions is very simple and easy form.

The implementation of Boolean functions by using logic gates involves in connecting one logic gate's output to another gate's input and involves in using AND, OR, NAND and NOR gates.

Table of Contents

- **SOP Boolean Function Implementation using logic gates**
  Logic gate implementation
  - Implementation for 2 input variables
  - Implementation for 3 input variables

- **POS Boolean Function Implementation using logic gates**
  Logic gate Implementation
  - Implementation for 2 input variables
  - Implementation for 3 input variables
- Implementation of Boolean functions using Universal logic gates
  - Implementation of Boolean functions using NAND gates
  - Implementation of Boolean functions using NOR gates

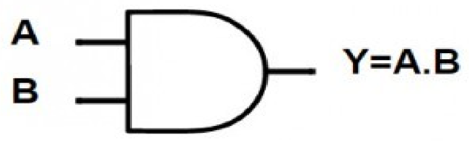
Logic gates

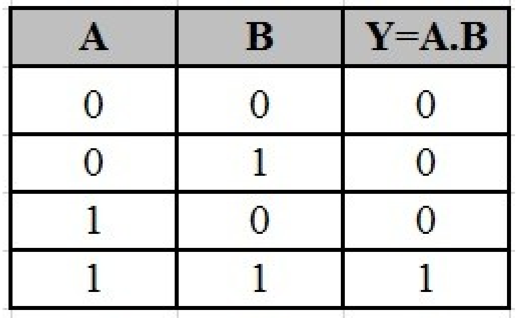
Logic gates are the basic building blocks of digital electronic circuits. A logic gate is a piece of an electronic circuit, that can be used to implement Boolean expressions.Laws and theorems of Boolean logic are used to manipulate the Boolean expressions and logic gates are used to implement these Boolean expressions in digital electronics. AND gate, OR gate and NOT gate are the three basic logic gates used in digital electronics.

AND Gate

Logic AND gate is a basic logic gate of which the output is equal to the product of its inputs. This gate multiplies both of its inputs so this gate is used to find the multiplication of inputs in binary algebra.

The output of an AND gate is HIGH only if both the inputs of the gate are HIGH. The output for all the other cases of the inputs is LOW. The logic symbol and the truth table of an AND gate is shown below.

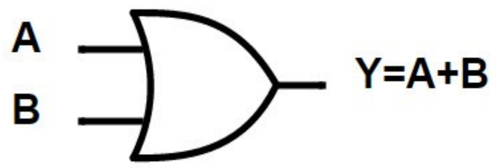




A	B	Y=A.B
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

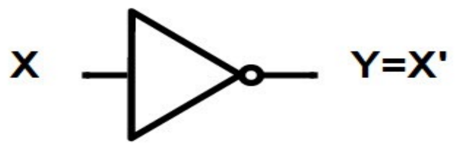
The output of the logic OR gate is equal to the sum of its inputs. This gate adds both of its inputs so this gate is used to find the summation or the addition of inputs in binary algebra. The output of an OR gate is HIGH if either of the inputs are HIGH. The output is LOW only when all the inputs are LOW. The logic symbol and the truth table of an OR gate is shown below.



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

NOT Gate

Logic NOT gate is a basic logic gate of which the output is equal to the inverse of its input. This gate produces the complement of the input. So this gate is used to represent the complement of variables in binary algebra. If the input is HIGH, the output is LOW and if the input is LOW, the output is HIGH. The logic symbol and the truth table of a NOT gate is shown below.



X	Y=X'
0	1
1	0

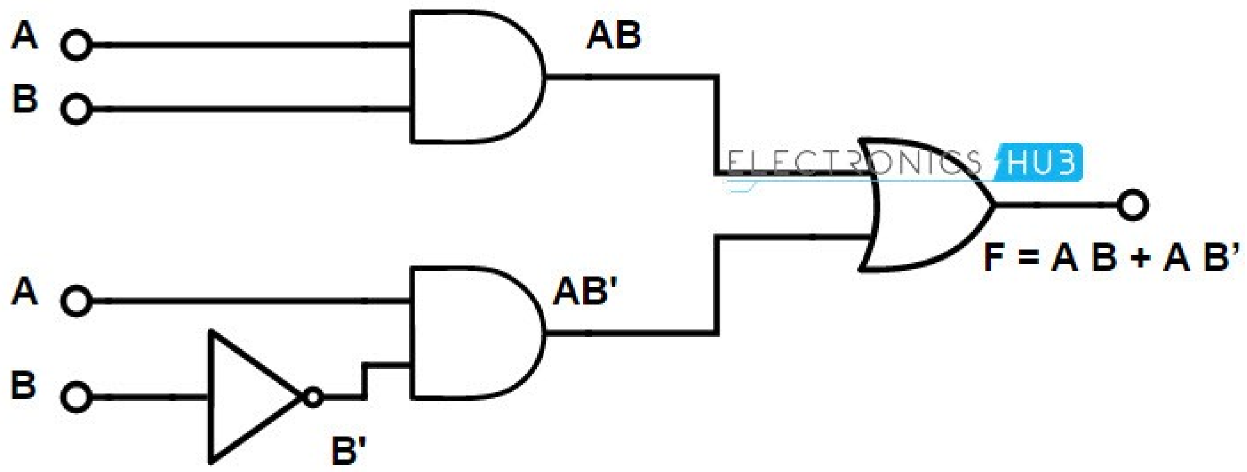
SOP Boolean Function Implementation using logic gates

The sum of product or SOP form is represented by using basic logic gates like AND gate and OR gate. The SOP form implementation will have the AND gate at its input side and as the output of the function is the sum of all product terms, it has an OR gate at its output side. This is important to remember that we use NOT gate to represent the inverse or complement of the variables.

Implementation for 2 input variables

Implement the Boolean function by using basic logic gates. F = A B + A B’

In the given SOP function, we have one compliment term, AB’. So to represent the compliment input, we are using the NOT gates at the input side. And to represent the product term, we use AND gates. See the below given logic diagram for representation of the Boolean function.

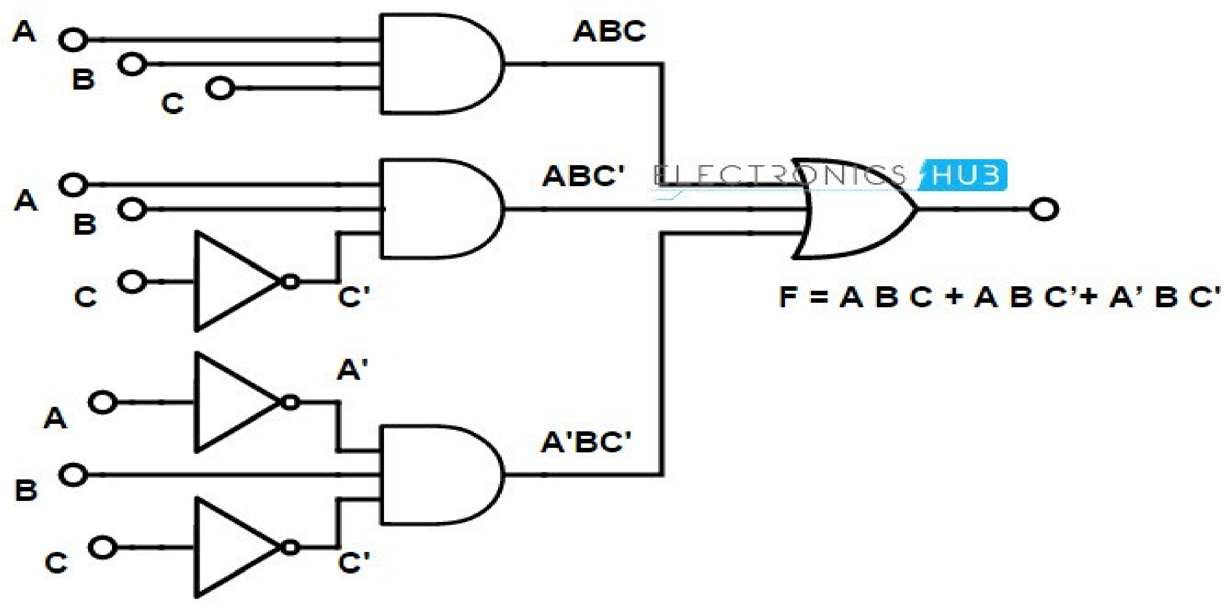


Implementation for 3 input variables

Implement the Boolean function by using basic logic gates.

F = A B C + A B C’+ A’ B C’

In the given function, we have two compliment terms, A’B C’ and ABC’. So to represent the compliment input, we are using the NOT gates at the input side. And to represent the product term, we use AND gates. See the below given logic diagram for representation of the Boolean function.



POS Boolean Function Implementation using logic gates

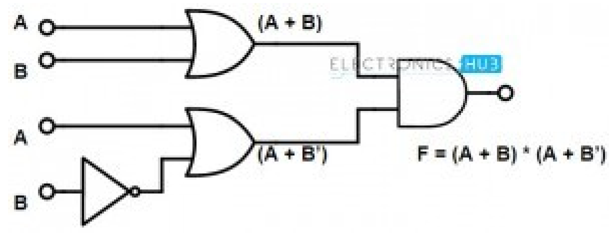
The product of sums or POS form can be represented by using basic logic gates like AND gate and OR gates. The POS form implementation will have the OR gate at its input side and as the output of the function is product of all sum terms, it has AND gate at its output side. In POS form implementation, we use NOT gate to represent the inverse or complement of the variables.

Ex 1:

Implementation for 2 input variables

Implement the Boolean function by using basic logic gates. F = (A + B) * (A + B')

In the given function, we have a complement term, (A + B) and (A + B'). So to represent the compliment input, we are using the NOT gates at the input side. And to represent the sum term, we use OR gates. See the below given logic diagram for representation of the Boolean function.

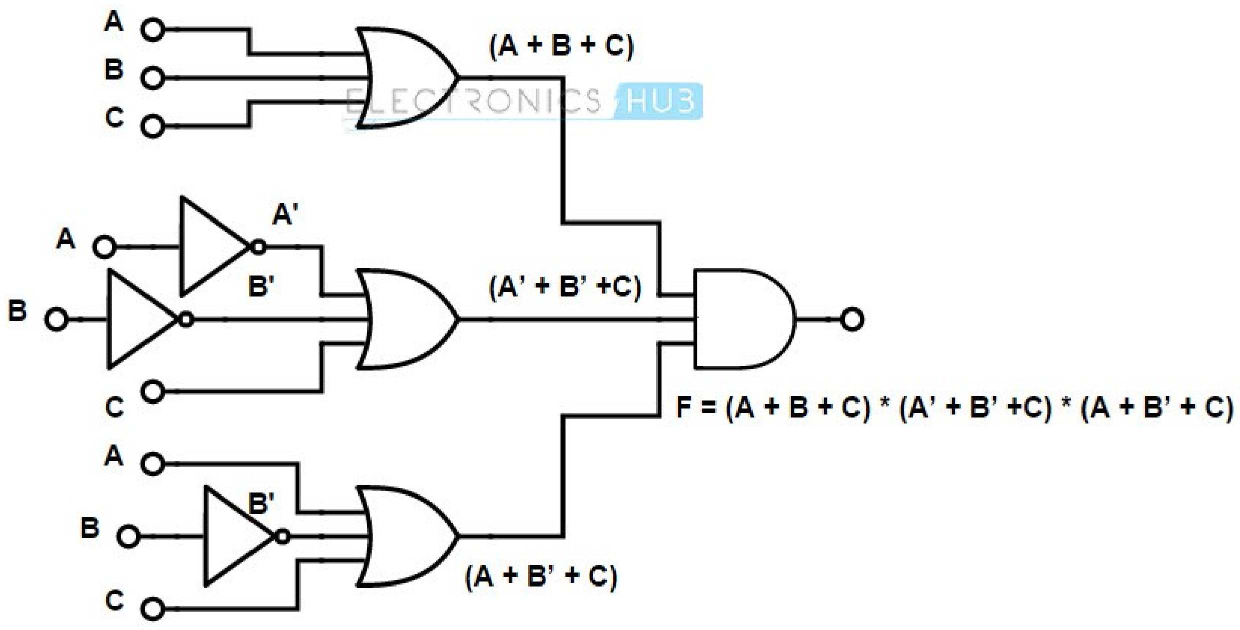


Implementation for 3 input variables

Implement the Boolean function by using basic logic gates.

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

In the given Boolean function, we have two compliment terms, (A' + B' +C) and (A + B' + C). So to represent the compliment input, we are using the NOT gates at the input side. And to represent the sum term, we use OR gates. See the below given logic diagram for representation of the Boolean function.



K-Map Method

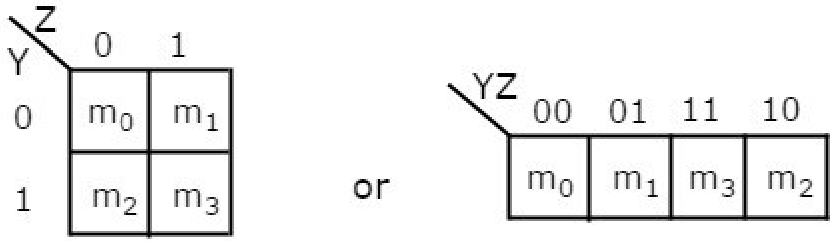
simplified the Boolean functions using Boolean postulates and theorems. It is a time consuming process and we have to re-write the simplified expressions after each step.To overcome this difficulty, **Karnaugh** introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit position.

K-Maps for 2 to 4 Variables

K-Map method is most suitable for minimizing Boolean functions of 2 variables to 5 variables. Now, let us discuss about the K-Maps for 2 to 5 variables one by one.

2 Variable K-Map

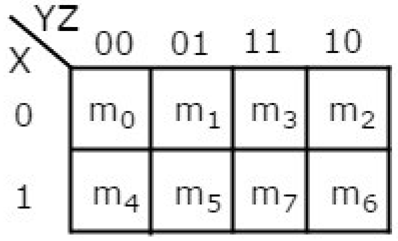
The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



- There is only one possibility of grouping 4 adjacent min terms.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_2, m_3), (m_0, m_2) and (m_1, m_3)}.

3 Variable K-Map

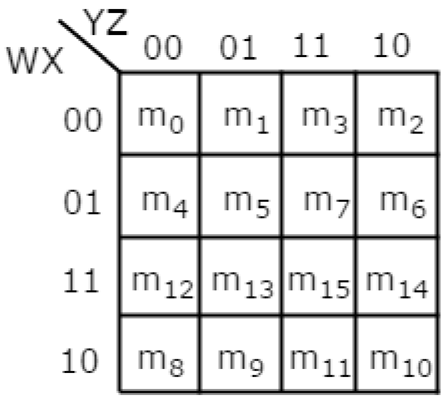
The number of cells in 3 variable K-map is eight, since the number of variables is three. The following figure shows **3 variable K-Map**.



- There is only one possibility of grouping 8 adjacent min terms.
- The possible combinations of grouping 4 adjacent min terms are {(m_0, m_1, m_3, m_2), (m_4, m_5, m_7, m_6), (m_0, m_1, m_4, m_5), (m_1, m_3, m_5, m_7), (m_3, m_2, m_7, m_6) and (m_2, m_0, m_6, m_4)}.
- The possible combinations of grouping 2 adjacent min terms are {(m_0, m_1), (m_1, m_3), (m_3, m_2), (m_2, m_0), (m_4, m_5), (m_5, m_7), (m_7, m_6), (m_6, m_4), (m_0, m_4), (m_1, m_5), (m_3, m_7) and (m_2, m_6)}.
- If x=0, then 3 variable K-map becomes 2 variable K-map.

4 Variable K-Map

The number of cells in 4 variable K-map is sixteen, since the number of variables is four. The following figure shows **4 variable K-Map**.



WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

- There is only one possibility of grouping 16 adjacent min terms.
- Let R_1, R_2, R_3 and R_4 represents the min terms of first row, second row, third row and fourth row respectively. Similarly, C_1, C_2, C_3 and C_4 represents the min terms of first column, second column, third column and fourth column respectively. The possible combinations of grouping 8 adjacent min terms are {(R_1, R_2), (R_2, R_3), (R_3, R_4), (R_4, R_1), (C_1, C_2), (C_2, C_3), (C_3, C_4), (C_4, C_1)}.
- If w=0, then 4 variable K-map becomes 3 variable K-map.