Unit II: DIGITAL LOGIC CIRCUIT

A Digital computer can be considered as a digital system that performs various computational tasks.

The first electronic digital computer was developed in the late 1940s and was used primarily for numerical computations.

By convention, the digital computers use the binary number system, which has two digits: 0 and 1. A binary digit is called a bit.

A computer system is subdivided into two functional entities: Hardware and Software1.4KFeatures of Java - Javatpoint

The hardware consists of all the electronic components and electromechanical devices that comprise the physical entity of the device.

The software of the computer consists of the instructions and data that the computer manipulates to perform various data-processing tasks.



* The Central Processing Unit (CPU) contains an arithmetic and logic unit for manipulating data, a number of registers for storing data, and a control circuit for fetching and executing instructions.
* The memory unit of a digital computer contains storage for instructions and data.
* The Random Access Memory (RAM) for real-time processing of the data.
* The Input-Output devices for generating inputs from the user and displaying the final results to the user.
* The Input-Output devices connected to the computer include the keyboard, mouse, terminals, magnetic disk drives, and other communication devices.

Logic Gates

* The logic gates are the main structural part of a digital system.
* Logic Gates are a block of hardware that produces signals of binary 1 or 0 when input logic requirements are satisfied.
* Each gate has a distinct graphic symbol, and its operation can be described by means of algebraic expressions.
* The seven basic logic gates includes: AND, OR, XOR, NOT, NAND, NOR, and XNOR.
* The relationship between the input-output binary variables for each gate can be represented in tabular form by a truth table.
* Each gate has one or two binary input variables designated by A and B and one binary output variable designated by x.

AND GATE:

The AND gate is an electronic circuit which gives a high output only if all its inputs are high. The AND operation is represented by a dot (.) sign.

OR GATE:

The OR gate is an electronic circuit which gives a high output if one or more of its inputs are high. The operation performed by an OR gate is represented by a plus (+) sign.



NOT GATE:

The NOT gate is an electronic circuit which produces an inverted version of the input at its output. It is also known as an **Inverter**.

NAND GATE:

The NOT-AND (NAND) gate which is equal to an AND gate followed by a NOT gate. The NAND gate gives a high output if any of the inputs are low. The NAND gate is represented by a AND gate with a small circle on the output. The small circle represents inversion.



NOR GATE:

The NOT-OR (NOR) gate which is equal to an OR gate followed by a NOT gate. The NOR gate gives a low output if any of the inputs are high. The NOR gate is represented by an OR gate with a small circle on the output. The small circle represents inversion.



Exclusive-OR/ XOR GATE:

The 'Exclusive-OR' gate is a circuit which will give a high output if one of its inputs is high but not both of them. The XOR operation is represented by an encircled plus sign.



EXCLUSIVE-NOR/Equivalence GATE:

The 'Exclusive-NOR' gate is a circuit that does the inverse operation to the XOR gate. It will give a low output if one of its inputs is high but not both of them. The small circle represents inversion.



# Boolean algebra

Boolean algebra can be considered as an algebra that deals with binary variables and logic operations. Boolean algebraic variables are designated by letters such as A, B, x, and y. The basic operations performed are AND, OR, and complement.

The Boolean algebraic functions are mostly expressed with binary variables, logic operation symbols, parentheses, and equal sign. For a given value of variables, the Boolean function can be either 1 or 0. For instance, consider the Boolean function:

F = x + y'z

The logic diagram for the Boolean function F = x + y'z can be represented as:



* The Boolean function F = x + y'z is transformed from an algebraic expression into a logic diagram composed of AND, OR, and inverter gates.
* Inverter at input 'y' generates its complement y'.
* There is an AND gate for the term y'z, and an OR gate is used to combine the two terms (x and y'z).
* The variables of the function are taken to be the inputs of the circuit, and the variable symbol of the function is taken as the output of the circuit.

#### **Note: A truth table can represent the relationship between a function and its binary variables. To represent a function in a truth table, we need a list of the 2^n combinations of n binary variables.**

The truth table for the Boolean function F = x + y'z can be represented as:



Examples of Boolean algebra simplifications using logic gates

In this section, we will look at some of the examples of Boolean algebra simplification using Logic gates.

1. F1 = xyz'



2. F2 = x + y'z



3. F3 = xy' + x'z

47.6M

806

History of Java



4. F4 = x'y'z + x'yz + xy'



Truth tables for F1= xyz', F2= x+y'z, F3= xy'+x'z and F4= x'y'z+x'yz+xy'

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **x** | **y** | **z** | **F1** | **F2** | **F3** | **F4** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 0 |

Laws of Boolean algebra

The basic Laws of Boolean Algebra can be stated as follows:

* Commutative Law states that the interchanging of the order of operands in a Boolean equation does not change its result. For example:
	1. OR operator → A + B = B + A
	2. AND operator → A \* B = B \* A
* Associative Law of multiplication states that the AND operation are done on two or more than two variables. For example:
A \* (B \* C) = (A \* B) \* C
* Distributive Law states that the multiplication of two variables and adding the result with a variable will result in the same value as multiplication of addition of the variable with individual variables. For example:
A + BC = (A + B) (A + C).
* Annulment law:
A.0 = 0
A + 1 = 1
* Identity law:
A.1 = A
A + 0 = A
* Idempotent law:
A + A = A
A.A = A
* Complement law:
A + A' = 1
A.A'= 0
* Double negation law:
((A)')' = A
* Absorption law:
A.(A+B) = A
A + AB = A

De Morgan's Law is also known as De Morgan's theorem, works depending on the concept of Duality. Duality states that interchanging the operators and variables in a function, such as replacing 0 with 1 and 1 with 0, AND operator with OR operator and OR operator with AND operator.

De Morgan stated 2 theorems, which will help us in solving the algebraic problems in digital electronics. The De Morgan's statements are:

1. "The negation of a conjunction is the disjunction of the negations", which means that the complement of the product of 2 variables is equal to the sum of the compliments of individual variables. For example, (A.B)' = A' + B'.
2. "The negation of disjunction is the conjunction of the negations", which means that compliment of the sum of two variables is equal to the product of the complement of each variable. For example, (A + B)' = A'B'.

## Boolean Expression

A logical statement that results in a Boolean value, either be True or False, is a Boolean expression. Sometimes, synonyms are used to express the statement such as ‘Yes’ for ‘True’ and ‘No’ for ‘False’. Also, 1 and 0 are used for digital circuits for True and False, respectively.

Boolean expressions are the statements that use logical operators, i.e., AND, OR, XOR and NOT. Thus, if we write X AND Y = True, then it is a Boolean expression.

## Boolean Algebra Terminologies

Now, let us discuss the important terminologies covered in Boolean algebra.

**Boolean Algebra**: Boolean algebra is the branch of algebra that deals with logical operations and binary variables.

**Boolean Variables:**A Boolean variable is defined as a variable or a symbol defined as a variable or a symbol, generally an alphabet that represents the logical quantities such as 0 or 1.

**Boolean Function:**A Boolean function consists of binary variables, logical operators, constants such as 0 and 1, equal to the operator, and the parenthesis symbols.

**Literal:**A literal may be a variable or a complement of a variable.

**Complement**: The complement is defined as the inverse of a variable, which is represented by a bar over the variable.

**Truth Table:**The truth table is a table that gives all the possible values of logical variables and the combination of the variables. It is possible to convert the Boolean equation into a truth table. The number of rows in the truth table should be equal to 2n, where “n” is the number of variables in the equation. For example, if a Boolean equation consists of 3 variables, then the number of rows in the truth table is 8. (i.e.,) 23 = 8.

### **Boolean Algebra Truth Table**

Now, if we express the above operations in a truth table, we get;

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | A ∧ B | A ∨ B |
| True | True | True | True |
| True | False | False | True |
| False | True | False | True |
| False | False | False | False |
| A | ¬A |
| True | False |
| False | True |

## Boolean Algebra Rules

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.
* The complement of a variable is represented by an overbar.

Thus, complement of variable B is represented as B¯. Thus if B=0 then B¯=1 and B =1 then B¯=0.

* OR-ing of the variables is represented by a plus (+) sign between them. For example, the OR-ing of A, B, and C is represented as A + B + C.
* Logical AND-ing of the two or more variables is represented by writing a dot between them, such as A.B.C. Sometimes, the dot may be omitted like ABC.

|  |
| --- |
| **Related Links** |
| [Truth Table](https://byjus.com/maths/truth-table/) | [Tautology](https://byjus.com/maths/tautology/) |
| [Conjunction](https://byjus.com/maths/conjunction/) | [Mathematical Logic](https://byjus.com/maths/mathematical-logic/) |

## Laws of Boolean Algebra

There are six types of [Boolean algebra laws](https://byjus.com/maths/boolean-algebra-laws/). They are:

* Commutative law
* Associative law
* Distributive law
* AND law
* OR law
* Inversion law

Those six laws are explained in detail here.

### **Commutative Law**

Any binary operation which satisfies the following expression is referred to as a commutative operation. Commutative law states that changing the sequence of the variables does not have any effect on the output of a logic circuit.

* A. B = B. A
* A + B = B + A

### **Associative Law**

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

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

### **Distributive Law**

Distributive law states the following conditions:

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

### **AND Law**

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

* A .0 = 0
* A . 1 = A
* A. A = A
* A.A¯=0

### **OR Law**

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

* A  + 0 = A
* A + 1 = 1
* A + A = A
* A+A¯=1

### **Inversion Law**

In Boolean algebra, the inversion law states that double inversion of variable results in the original variable itself.

* A¯¯=A

## Boolean Algebra Theorems

The two important theorems which are extremely used in Boolean algebra are De Morgan’s First law and De Morgan’s second law. These two theorems are used to change the Boolean expression. This theorem basically helps to reduce the given Boolean expression in the simplified form. These two De Morgan’s laws are used to change the expression from one form to another form. Now, let us discuss these two theorems in detail.

**De Morgan’s First Law:**

De Morgan’s First Law states that  (A.B)’ = A’+B’.

The first law states that the complement of the product of the variables is equal to the sum of their individual complements of a variable.

The truth table that shows the verification of De Morgan’s First law is given as follows:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **A’** | **B’** | **(A.B)’** | **A’+B’** |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 |

The last two columns show that (A.B)’ = A’+B’.

Hence, De Morgan’s First Law is proved.

**De Morgan’s Second Law:**

De Morgan’s Second law states that (A+B)’ = A’. B’.

The second law states that the complement of the sum of variables is equal to the product of their individual complements of a variable.

The following truth table shows the proof for De Morgan’s second law.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **A** | **B** | **A’** | **B’** | **(A+B)’** | **A’. B’** |
| 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 | 0 |

The last two columns show that (A+B)’ = A’. B’.

Hence, De Morgan’s second law is proved.

 The other theorems in Boolean algebra are complementary theorem, duality theorem, transposition theorem, redundancy theorem and so on. All these theorems are used to simplify the given Boolean expression. The reduced Boolean expression should be equivalent to the given Boolean expression.

## Solved Examples

**Question:**Simplify the following expression:

c+BC¯

Solution:

Given:

C+BC¯

According to [Demorgan’s law](https://byjus.com/maths/de-morgans-first-law/), we can write the above expressions as

C+(B¯+C¯)

From Commutative law:

(C+C¯)+B¯

From Complement law

1+B¯=1

Therefore,

C+BC¯=1

**Question 2: Draw a truth table for A(B+D).**

Solution: Given expression A(B+D).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | D | B+D | A(B+D) |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**Basic Identifier**

The algebraic equations which are valid for all values of variables in them are called algebraic identities. They are also used for the factorization of polynomials. In this way, algebraic identities are used in the computation of [algebraic expressions](https://byjus.com/maths/algebraic-expressions/) and solving different polynomials. You have already learned about a few of them in the junior grades. In this article, we will recall them and introduce you to some more standard algebraic identities, along with examples.

Standard Algebraic Identities List

All the standard Algebraic Identities are derived from the Binomial Theorem, which is given as:

(a+b)n=nC0.an.b0+nC1.an−1.b1+……..+nCn−1.a1.bn−1+nCn.a0.bn

Some Standard Algebraic Identities list are given below:

***Identity I:***(a + b)2 = a2 + 2ab + b2

***Identity II:***(a – b)2 = a2 – 2ab + b2

***Identity III:***a2 – b2= (a + b)(a – b)

***Identity IV:***(x + a)(x + b) = x2 + (a + b) x + ab

***Identity V:***(a + b + c)2 = a2 + b2 + c2 + 2ab + 2bc + 2ca

***Identity VI:***(a + b)3 = a3 + b3 + 3ab (a + b)

***Identity VII:***(a – b)3 = a3 – b3 – 3ab (a – b)

***Identity VIII:***a3 + b3 + c3– 3abc = (a + b + c)(a2 + b2 + c2 – ab – bc – ca)

Solved Examples of Algebraic Identities

**Example 1:**

Find the product of (x + 1)(x + 1) using standard algebraic identities.

**Solution:**

(x + 1)(x + 1) can be written as (x + 1)2. Thus, it is of the form Identity I where a = x and b = 1. So we have,

(x + 1)2 = (x)2 + 2(x)(1) + (1)2= x2+ 2x + 1

**Example 2:**

Factorise (x4 – 1) using standard algebraic identities.

**Solution:**

(x4 – 1) is of the form Identity III where a = x2 and b = 1. So we have,

(x4 – 1) = ((x2)2– 12) = (x2+ 1)(x2– 1)

The factor (x2– 1) can be further factorised using the same Identity III where a = x and b = 1. So,

(x4 – 1) = (x2+ 1)((x)2–(1)2) = (x2+ 1)(x + 1)(x – 1)

**Eample 3:**

Factorise 16x2 + 4y2+ 9z2 – 16xy + 12yz – 24zx using standard algebraic identities.

**Solution:**

16x2 + 4y2+ 9z2– 16xy + 12yz – 24zx is of the form Identity V. So we have,

16x2 + 4y2+ 9z2 – 16xy + 12yz – 24zx = (4x)2 + (-2y)2 + (-3z)2 + 2(4x)(-2y) + 2(-2y)(-3z) + 2(-3z)(4x)= (4x – 2y – 3z)2 = (4x – 2y – 3z)(4x – 2y – 3z)

**Example 4:**

Expand (3x – 4y)3using standard algebraic identities.

**Solution:**

(3x– 4y)3is of the form Identity VII where a = 3x and b = 4y. So we have,

(3x – 4y)3 = (3x)3 – (4y)3– 3(3x)(4y)(3x – 4y) = 27x3 – 64y3 – 108x2y + 144xy2

**Example 5:**

Factorize (x3 + 8y3+ 27z3 – 18xyz) using standard algebraic identities.

**Solution:**

(x3 + 8y3+ 27z3 – 18xyz)is of the form Identity VIII where a = x, b = 2y and c = 3z. So we have,

(x3 + 8y3+ 27z3 – 18xyz) = (x)3 + (2y)3+ (3z)3 – 3(x)(2y)(3z)= (x + 2y + 3z)(x2+ 4y2 + 9z2 – 2xy – 6yz – 3zx)

# DeMorgan’s Theorem

DeMorgan´s Theorem and Laws can be used to to find the equivalency of the NAND and NOR gates



DeMorgan’s Theorem uses two sets of rules or laws to solve various Boolean algebra expressions by changing OR’s to AND’s, and AND’s to OR’s

Boolean Algebra uses a set of laws and rules to define the operation of a digital logic circuit with “0’s” and “1’s” being used to represent a digital input or output condition. Boolean Algebra uses these zeros and ones to create truth tables and mathematical expressions to define the digital operation of a logic AND, OR and NOT (or inversion) operations as well as ways of expressing other logical operations such as the XOR (Exclusive-OR) function.

While George Boole’s set of laws and rules allows us to analyise and simplify a digital circuit, there are two laws within his set that are attributed to **Augustus DeMorgan** (a nineteenth century English mathematician) which views the logical NAND and NOR operations as separate NOT AND and NOT OR functions respectively.

But before we look at **DeMorgan’s Theory** in more detail, let’s remind ourselves of the basic logical operations where A and B are logic (or Boolean) input binary variables, and whose values can only be either “0” or “1” producing four possible input combinations, 00, 01, 10, and 11.

### Truth Table for Each Logical Operation

|  |  |
| --- | --- |
| Input Variable | Output Conditions |
| A | B | AND | NAND | OR | NOR |  |  |
| 0 | 0 | 0 | 1 | 0 | 1 |  |  |
| 0 | 1 | 0 | 1 | 1 | 0 |  |  |
| 1 | 0 | 0 | 1 | 1 | 0 |  |  |
| 1 | 1 | 1 | 0 | 1 | 0 |  |  |

The following table gives a list of the common logic functions and their equivalent Boolean notation where a “.” (a dot) means an AND (product) operation, a “+” (plus sign) means an OR (sum) operation, and the complement or inverse of a variable is indicated by a bar over the variable.

|  |  |
| --- | --- |
| Logic Function | Boolean Notation |
| AND | A.B |
| OR | A+B |
| NOT | A |
| NAND | A .B |
| NOR | A+B |

## DeMorgan’s Theory

DeMorgan’s Theorems are basically two sets of rules or laws developed from the Boolean expressions for AND, OR and NOT using two input variables, A and B. These two rules or theorems allow the input variables to be negated and converted from one form of a Boolean function into an opposite form.

DeMorgan’s first theorem states that two (or more) variables NOR´ed together is the same as the two variables inverted (Complement) and AND´ed, while the second theorem states that two (or more) variables NAND´ed together is the same as the two terms inverted (Complement) and OR´ed. That is replace all the OR operators with AND operators, or all the AND operators with an OR operators.

### DeMorgan’s First Theorem

DeMorgan’s First theorem proves that when two (or more) input variables are AND’ed and negated, they are equivalent to the OR of the complements of the individual variables. Thus the equivalent of the NAND function will be a negative-OR function, proving that A.B = A+B. We can show this operation using the following table.

### Verifying DeMorgan’s First Theorem using Truth Table

|  |  |
| --- | --- |
| Inputs | Truth Table Outputs For Each Term |
| B | A | A.B | A.B | A | B | A + B |  |
| 0 | 0 | 0 | **1** | 1 | 1 | **1** |  |
| 0 | 1 | 0 | **1** | 0 | 1 | **1** |  |
| 1 | 0 | 0 | **1** | 1 | 0 | **1** |  |
| 1 | 1 | 1 | **0** | 0 | 0 | **0** |  |

We can also show that A.B = A+B using logic gates as shown.

### DeMorgan’s First Law Implementation using Logic Gates



The top logic gate arrangement of: A.B can be implemented using a standard NAND gate with inputs A and B. The lower logic gate arrangement first inverts the two inputs producing A and B. These then become the inputs to the OR gate. Therefore the output from the OR gate becomes: A+B

Then we can see here that a standard OR gate function with inverters (NOT gates) on each of its inputs is equivalent to a NAND gate function. So an individual NAND gate can be represented in this way as the equivalency of a NAND gate is a negative-OR.

### DeMorgan’s Second Theorem

DeMorgan’s Second theorem proves that when two (or more) input variables are OR’ed and negated, they are equivalent to the AND of the complements of the individual variables. Thus the equivalent of the NOR function is a negative-AND function proving that A+B = A.B, and again we can show operation this using the following truth table.

### Verifying DeMorgan’s Second Theorem using Truth Table

|  |  |
| --- | --- |
| Inputs | Truth Table Outputs For Each Term |
| B | A | A+B | A+B | A | B | A . B |  |
| 0 | 0 | 0 | **1** | 1 | 1 | **1** |  |
| 0 | 1 | 1 | **0** | 0 | 1 | **0** |  |
| 1 | 0 | 1 | **0** | 1 | 0 | **0** |  |
| 1 | 1 | 1 | **0** | 0 | 0 | **0** |  |

We can also show that A+B = A.B using the following logic gates example.

### DeMorgan’s Second Law Implementation using Logic Gates



The top logic gate arrangement of: A+B can be implemented using a standard NOR gate function using inputs A and B. The lower logic gate arrangement first inverts the two inputs, thus producing A and B. Thus then become the inputs to the AND gate. Therefore the output from the AND gate becomes: A.B

Then we can see that a standard AND gate function with inverters (NOT gates) on each of its inputs produces an equivalent output condition to a standard NOR gate function, and an individual NOR gate can be represented in this way as the equivalency of a NOR gate is a negative-AND.

Although we have used DeMorgan’s theorems with only two input variables A and B, they are equally valid for use with three, four or more input variable expressions, for example:

For a 3-variable input

A.B.C = A+B+C

and also

A+B+C = A.B.C

For a 4-variable input

A.B.C.D = A+B+C+D

and also

A+B+C+D = A.B.C.D

and so on.

## DeMorgan’s Equivalent Gates

We have seen here that by using DeMorgan’s Theorems we can replace all of the AND (.) operators with an OR (+) and vice versa, and then complements each of the terms or variables in the expression by inverting it, that is 0’s to 1’s and 1’s to 0’s before inverting the entire function.

Thus to obtain the DeMorgan equivalent for an AND, NAND, OR or NOR gate, we simply add inverters (NOT-gates) to all inputs and outputs and change an AND symbol to an OR symbol or change an OR symbol to an AND symbol as shown in the following table.

### DeMorgan’s Equivalent Gates

|  |  |
| --- | --- |
| Standard Logic Gate | DeMorgan’s Equivalent Gate |
| and gate symbol | demorgans theorem negative-nor gate |
| nand gate symbol | demorgans theorem negative-or gate |
| or gate symbol | demorgans theorem negative-nand gate |
| nor gate symbol | demorgans theorem negative-and gate |

Then we have seen in this tutorial about DeMorgan’s Thereom that the complement of two (or more) AND’ed input variables is equivalent to the OR of the complements of these variables, and that the complement of two (or more) OR’ed variables is equivalent to the AND of the complements of the variables as defined by DeMorgan.

# Map Simplification

The Map method involves a simple, straightforward procedure for simplifying Boolean expressions.

Map simplification may be regarded as a pictorial arrangement of the truth table which allows an easy interpretation for choosing the minimum number of terms needed to express the function algebraically. The map method is also known as Karnaugh map or K-map.

Each combination of the variables in a truth table is called a mid-term.

#### **Note: When expressed in a truth table a function of n variables will have 2^n min-terms, equivalent to the 2^n binary numbers obtained from n bits.**

There are four min-terms in a two variable map. Therefore, the map consists of four squares, one for each min-term. The 0's and 1's marked for each row, and each column designates the values of variable x and y, respectivelyJava Try Catch

**Two-variable map:**



**Representation of functions in the two-variable map:**



## Three variable map

There are eight min-terms in a three-variable map. Therefore, the map consists of eight squares.

**Three variable map:**



* The map was drawn in part (b) in the above image is marked with numbers in each row and each column to show the relationship between the squares and the three variables.
* Any two adjacent squares in the map differ by only one variable, which is primed in one square and unprimed in the other. For example, m5 and m7 lie in the two adjacent squares. Variable y is primed in m5 and unprimed in m7, whereas the other two variables are the same in both the squares.
* From the postulates of Boolean algebra, it follows that the sum of two min-terms in adjacent squares can be simplified to a single AND term consisting of only two literals. For example, consider the sum of two adjacent squares say m5 and m7:
m5+m7 = xy'z+xyz= xz(y'+y)= xz.

A Karnaugh map or a K-map refers to a pictorial method that is utilised to minimise various Boolean expressions without using the Boolean algebra theorems along with the equation manipulations. A Karnaugh map can be a special version of the truth table. We can easily minimise various expressions that have 2 to 4 variables using a K-map.

## Introduction of K-Map (Karnaugh Map)

In numerous digital circuits and other practical problems, finding expressions that have minimum variables becomes a prerequisite. In such cases, minimisation of Boolean expressions is possible that have 3, 4 variables. It can be done using the Karnaugh map without using any theorems of Boolean algebra. The K-map can easily take two forms, namely, Sum of Product or SOP and Product of Sum or POS, according to what we need in the problem. K-map is a representation that is table-like, but it gives more data than the TRUTH TABLE. Fill a grid of K-map with 1s and 0s, then solve it by creating various groups.

## Solving an Expression Using K-Map

Here are the steps that are used to solve an expression using the K-map method:

1. Select a K-map according to the total number of variables.

2. Identify maxterms or minterms as given in the problem.

3. For SOP, put the 1’s in the blocks of the K-map with respect to the minterms (elsewhere 0’s).

4. For POS, putting 0’s in the blocks of the K-map with respect to the maxterms (elsewhere 1’s).

5. Making rectangular groups that contain the total terms in the power of two, such as 2,4,8 ..(except 1) and trying to cover as many numbers of elements as we can in a single group.

6. From the groups that have been created in step 5, find the product terms and then sum them up for the SOP form.

### **SOP FORM**

#### **1. 3 variables K-map:**

Z = ∑P, Q, R (1, 3, 6, 7)



From the red group, the product term would be —

P’R

From the green group, the product term would be —

PQ

If we sum these product terms, then we will get this final expression (P’R + PQ)

#### **2. 4 variables K-map:**

F (A, B, C, D) = ∑(0, 2, 5, 7, 8, 10, 13, 15)



From the red group, the product term would be —

BD

From the lilac group, the product term would be —

B’D’

If we sum these product terms, then we will get this final expression (BD + B’D’)

### **POS FORM**

#### **1. 3 variables K-map**

F (P, Q, R) = π(0,3,6,7)



From the lilac group, the terms would be

P Q

If we take the complement of these two

P’ Q’

And then sum up them

(P’ + Q’)

From the blue group, the terms would be

B R

When we take the complement of these terms

B’ R’

And then sum them up

(B’ + R’)

From the red group, the terms would be

P’ Q’ R’

If we take the complement of the two terms

P Q R

And then sum them up

(P + Q + R)

If we take the product of these three terms, then we will get this final expression –

(P’ + Q’) (P’ + R’) (P + Q + R)

#### **2. 4 variables K-map**

F (P, Q, R, S) = π (3, 5, 7, 8, 10, 11, 12, 13)



From the blue group, the terms would be

R’ S Q

We take their complement and then sum them

(R + S’+ Q’)

From the purple group, the terms would be

R S P’

We take their complement and then sum them

(R’ + S’+ P) S

From the red group, the terms would be

P R’ S’

We take their complement and then sum them

(P’ + R + S)

From the lilac group, the terms would be

P Q’ R

We take their complement and then sum them

(P’ + Q + R’)

Finally, we will express these in the form of the product –

(R + S’+ Q’).(R’ + S’+A).(P’+ R + S).(P’+ Q + R’)

Pitfall – Always remember that POS ≠ (SOP)’

\*Here, the correct form would be (POS of F) = (SOP of F’)’

# Minterm

There are two ways in which we can put the Boolean function. These ways are the minterm canonical form and maxterm canonical form.

## Literal

A Literal signifies the Boolean variables including their complements. Such as B is a boolean variable and its complements are ~B or B', which are the literals.

## Minterm

The product of all literals, either with complement or without complement, is known as **minterm**

he minterm for the Boolean variables A and B is:

1. A.B
2. A.~B
3. ~A.B

The complement variables ~A and ~B can also be written as A' and B' respectively. Thus, we can write the minterm as:

1. A.B'
2. A'.B

Minterm from values

Using variable values, we can write the minterms as:

1. If the variable value is 1, we will take the variable without its complement.
2. If the variable value is 0, take its complement.

**Example**

Let's assume that we have three Boolean variables A, B, and C having values

A=1
B=0
C=0

Now, we will take the complement of the variables B and C because these values are 0 and will take A without complement. So, the minterm will be:

Minterm=A.B'C'

Let's take another example in which we have two variables B and C having the value

B = 0
C = 1

Minterm=B'C

Shorthand notation for minterm

We know that, when Boolean variables are in the form of minterm, the variables will appear in the product. There are the following steps for getting the shorthand notation for minterm.

* In the first step, we will write the term consisting of all the variables
* Next, we will write 0 in place of all the complement variables such as ~A or A'.
* We will write 1 in place of all the non-complement variables such as A or b.
* Now, we will find the decimal number of the binary formed from the above steps.
* In the end, we will write the decimal number as a subscript of letter **m**(minterm). Let's take some example to understand the theory of shorthand notation

**Example 1: Minterm = AB'**

* First, we will write the minterm:
Minterm = AB'
* Now, we will write 0 in place of complement variable B'.
Minterm = A0
* We will write 1 in place of non-complement variable A.
Minterm = 10
* The binary number of the minterm AB' is 10. The decimal point number of (10)2 is 2. So, the shorthand notation of AB' is
Minterm = m2

**Example 2: Minterm = AB'C'**

* First, we will write the minterm:
Minterm = AB'C'
* Now, we will write 0 in place of complement variables B' and C'.
Minterm = A00
* We will write 1 in place of non-complement variable A.
Minterm = 100
* The binary number of the minterm AB'C' is 100. The decimal point number of (100)2 is 4. So, the shorthand notation of AB'C' is
Minterm = m4

Product of Sums Simplification

The Product of Sum expression is equivalent to the logical OR-AND fuction which gives the AND Product of two or more OR Sums to produce an output



n the tutorial about the [**Sum-of-Products**](https://www.electronics-tutorials.ws/boolean/sum-of-product.html) (SOP) expression, we saw that it represents a standard Boolean (switching) expression which “Sums” two or more “Products” by taking the output from two or more logic AND gates and OR’s them together to create the final output. But we can also take the outputs of two or more OR gates and connect them as inputs to an AND gate to produce a “Product of the Sum” (OR-AND logic) output.

In Boolean Algebra, the addition of two values is equivalent to the logical OR function thereby producing a “Sum” term when two or more input variables or constants are “OR’ed” together. In other words, in Boolean Algebra the OR function is the equivalent of addition and so its output state represents the “Sum” of its inputs.

**Product of Sum** expressions are Boolean expressions made up of sums consisting of one or more variables, either in its normal true form or complemented form or combinations of both, which are then AND’ed together. If a Boolean function of multiple variables is expressed in Product-of-Sum terms, then each term is called the max term. That is the variable is taken as a logic “0” as we will see later. But first let us understand more what represents a Sum Term.

## The Sum (OR) Term

While the AND function is commonly referred to as the Product term, the OR function is referred to as a sum term. The OR function is the mathemetical equivalent of addition which is denoted by a plus sign, (+). Thus a 2-input OR gate has an output term represented by the Boolean expression of A+B because it is the logical sum of A and B.

### OR Gate (Sum)



This logical sum is known commonly as Boolean addition as an OR function produces the summed term of two or more input variables, or constants. Thus the Boolean equation for a 2-input OR gate is given as: Q = A+B, that is Q equals both A OR B. For a sum term these input variables can be either “true” or “false”, “1” or “0”, or be of a complemented form, so A+B, A+B or A+B are all classed as sum terms.

So we now know that in Boolean Algebra, “sum” means the OR’ing of the terms with the variables in a sum term having one instance in its true uncomplemented form or in its complemented form so that the resulting sum expression cannot be simplified any further. These sum terms are known as maxterms, that is a max term is a complete sum of all the variables and constants with or without inversion within the Boolean expression. So how can we show the operation of this “sum” function in Boolean Albegra.

A sum term can have one or two independant variables, such as A and B, or it can have one or two fixed constants, again 0 and 1. We can use these variables and constants in a variety of different combinations producing a sum result as shown in the following lists.

### Boolean Algebra Sum Terms

* Variable and Constants
* A + 0 = A
* A + 1 = 1
* A + A = A
* A + A = 1
* Constants Only
* 0 + 0 = 0
* 0 + 1 = 1
* 1 + 0 = 1
* 1 + 1 = 1

Note that a Boolean “variable” can have one of two values, either “1” or “0”, and can change its value. For example, A = 0, or A = 1 whereas a Boolean “constant” which can also be in the form of a “1” or “0”, is a fixed value and therefore cannot change.

Then we can see that any given Boolean sum can be simplified to a single constant or variable with a brief description of the various Boolean Laws given below where “A” represents a variable input.

* **Identity Law** – A term OR’ed with 0 is always equal to the term (A+0 = A)
* **Annulment Law** – A term OR’ed with 1 is always equal to 1 (A+1 = 1)
* **Idempotent Law** – A term OR’ed with itself is always equal to the term (A+A = A)
* **Complement Law** – A term OR’ed with its complement is always equal to 1 (A+A = 1)
* **Commutative Law** – The order in which two terms are OR’ed is the same (A+1 = 1+A)

## The Product (AND) Term

While the OR function is commonly referred to as the sum term, the AND function is referred to as the product term. The AND function is the mathemetical equivalent of multiplication which is denoted by a Cross (x), or a Star (\*) sign. Thus a 2-input AND gate has an output term represented by the Boolean expression of A.B because it is the logical product of A and B.

### AND Gate (Product)



This logical product is known commonly as Boolean multiplication as the AND function produces the multiplied term of two or more input variables, or constants. But for now we will remember that the AND function represents the **Product Term**.

## Product of Sum

So we have seen that the OR function produces the logical sum of Boolean addition, and that the AND function produces the logical sum of Boolean multiplication. But when dealing with combinational logic circuits in which AND gates, OR gates and NOT gates are connected together, the expressions of **Product-of-Sum** is widely used.

The Product of Sum (POS) expression comes from the fact that two or more sums (OR’s) are added (AND’ed) together. That is the outputs from two or more OR gates are connected to the input of an AND gate so that they are effectively AND’ed together to create the final (OR AND) output. For example, the following Boolean function is a typical product-of-sum expression:

### Product of Sum Expressions

Q = (A + B).(B + C).(A + 1)

and also

(A + B + C).(A + C).(B + C)

However, Boolean functions can also be expressed in nonstandard product of sum forms like that shown below but they can be converted to a standard POS form by using the distributive law to expand the expression with respect to the sum. Therefore:

Q = A + (BC)

Becomes in expanded product-of-sum terms:

Q = (A + B)(A + C)

Another nonstandard example is:

Q = (A + B) + (A.C)

Becomes as an expanded product-of-sum expession:

Q = (A + B + A)(A + B + C)

which can, if required be reduced using distributive law and absorption law too:

Q = (A + B)(A + B + C)

Q = A + B + C

Q = A + B

## Converting an POS Expression into a Truth Table

We can display any product-of-sum term in the form of a truth table as each input combination that produces a logic “0” output is an OR or sum term as shown below.

Consider the following product of sum expression:

Q = (A + B + C)(A + B + C)(A + B + C)

We can now draw up the truth table for the above expression to show a list of all the possible input combinations for A, B and C which will result in an output “0”.

### Product of Sum Truth Table Form

|  |  |  |
| --- | --- | --- |
| Inputs | Output | Product |
| C | B | A | Q |   |
| 0 | 0 | 0 | 0 | A + B + C |
| 0 | 0 | 1 | 1 |   |
| 0 | 1 | 0 | 0 | A + B + C |
| 0 | 1 | 1 | 1 |   |
| 1 | 0 | 0 | 1 |   |
| 1 | 0 | 1 | 1 |   |
| 1 | 1 | 0 | 0 | A + B + C |
| 1 | 1 | 1 | 1 |   |

Then we can clearly see from the truth table that each row which produces a “0” for its output corresponds to its Boolean addition expression with all of the other rows having a “1” output. The advantage here is that the truth table gives us a visual indication of the Boolean expression allowing us to simplify the expression remembering that a sum term produces a “0” output when all of its inputs are equal to “0”. So to make a sum term row equal to “0”, the we must invert all the inputs which are equal to “1”.

## Product-of-Sum Example

The following Boolean Algebra expression is given as:

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

1. Use a truth table to show all the possible combinations of input conditions that will produces a “0” output.

2. Draw a logic gate diagram for the POS expression.

1. Truth Table

### Product of Sum Truth Table Form

|  |  |  |
| --- | --- | --- |
| Inputs | Output | Product |
| C | B | A | Q |   |
| 0 | 0 | 0 | **0** | A + B + C |
| 0 | 0 | 1 | 1 |   |
| 0 | 1 | 0 | **0** | A + B + C |
| 0 | 1 | 1 | 1 |   |
| 1 | 0 | 0 | 1 |   |
| 1 | 0 | 1 | **0** | A + B + C |
| 1 | 1 | 0 | **0** | A + B + C |
| 1 | 1 | 1 | 1 |   |

2. Logic Gate Diagram



Then we have seen in this tutorial that the **Product-of-Sum** (POS) expression is a standard boolean expression that takes the “Product” of two or more “Sums”. For a digital logic circuit the POS expression takes the output of two or more logic OR gates and AND’s them together to create the final OR-AND logic output.

# Don't Care Condition

The "Don't care" condition says that we can use the blank cells of a K-map to make a group of the variables. To make a group of cells, we can use the "don't care" cells as either 0 or 1, and if required, we can also ignore that cell. We mainly use the "don't care" cell to make a large group of cells.

The cross(×) symbol is used to represent the "don't care" cell in K-map. This cross symbol represents an invalid combination. The "don't care" in excess-3 code are 0000, 0001, 0010, 1101, 1110, and 1111 because they are invalid combinations. Apart from this, the 4-bit BCD to Excess-3 code, the "don't care" are 1010, 1011, 1100, 1101, 1110, and 1111.
We can change the standard SOP function into a POS expression by making the "don't care" terms the same as they are. The missing minterms of the POS form are written as maxterms of the POS form. In the same way, we can change the standard POS function into an SOP expression by making the "don't care" terms the same as they are. The missing maxterms of the SOP form are written as minterm of the SOP form.

**Example 1: Minimize f = m(1,5,6,12,13,14) + d(4) in SOP minimal form**

**Solution:**

The k-map of the given function in the SOP form is as follows:

So, the minimized SOP form of the function is:

f = BC' + BD' + A'C'D

**Example 2: Minimize F(A,B,C,D) = m(0,1,2,3,4,5) + d(10,11,12,13,14,15) in SOP minimal form**

Solution:

The POS form of the given function is:

F(A,B,C,D) = M(6,7,8,9) + d(10,11,12,13,14,15)

The POS K-map for the given expression is:

So, the minimized POS form of the function is:

F = A'(B' + C')

**Example-3:**
**Minimize the following function in SOP minimal form using K-Maps: F(A, B, C, D) = m(1, 2, 6, 7, 8, 13, 14, 15) + d(3, 5, 12)**

**Explanation:**

The SOP K-map for the given expression is:

Therefore,

f = AC'D' + A'D + A'C + AB

### **Significance of "Don't Care" Conditions:**

Don't Care conditions has the following significance with respect to the digital circuit design:

**Simplification:**

These conditions denote the set of inputs that never occurs for given digital circuits. Therefore, to simplify the boolean output expressions, the 'don't care' are used.

**Reduced Power Consumption:**

The switching of the state is reduced when we group the terms long with "don't care". This reduces the required memory space resulting in lower power consumption.

**Lesser number of gates:**

For reducing the number of gates that are used to implement the given expression, simplification places an important role. So, the 'don't care' makes the logic design more economical.

**Prevention of Hazards:**

In the digital system, the 'don't care' place an important role in hazards prevention.