|  |
| --- |
| **DEPARTMENT OF COMPUTER SCIENCE**  **“COMPUTER SYSTEM ARCHITECTURE”**  **S.SUNDARESWARI**  **DATE : 6/08/2020** |

# Introduction of Logic Gates

In Boolean Algebra, there are three basic operations,  which are analogous to disjunction, conjunction, and negation in [propositional logic](https://www.geeksforgeeks.org/proposition-logic/). Each of these operations has a corresponding logic gate. Apart from these there are a few other logic gates as well.

**Logic Gates –**

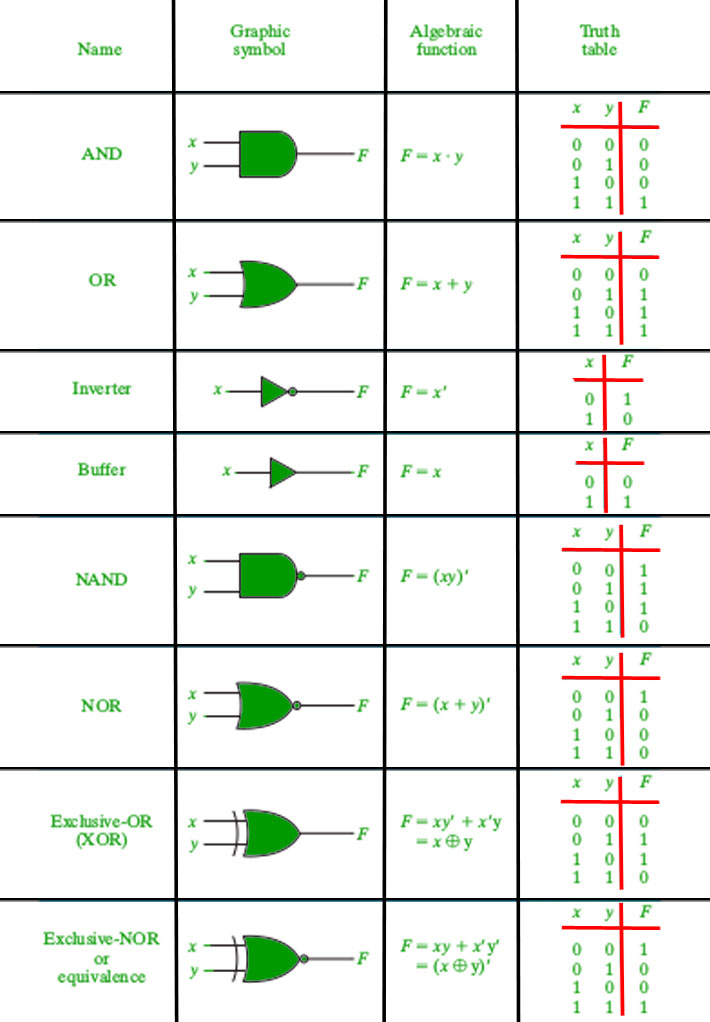
* **AND gate(.) –** The AND gate gives an output of 1 if both the two inputs are 1, it gives 0 otherwise.
* **OR gate(+) –** The OR gate gives an output of 1 if either of the two inputs are 1, it gives 0 otherwise.
* **NOT gate(‘) –** The NOT gate gives an output of 1 input is 0 and vice-versa.
* **XOR gate() –** The XOR gate gives an output of 1 if either both inputs are different, it gives 0 if they are same.

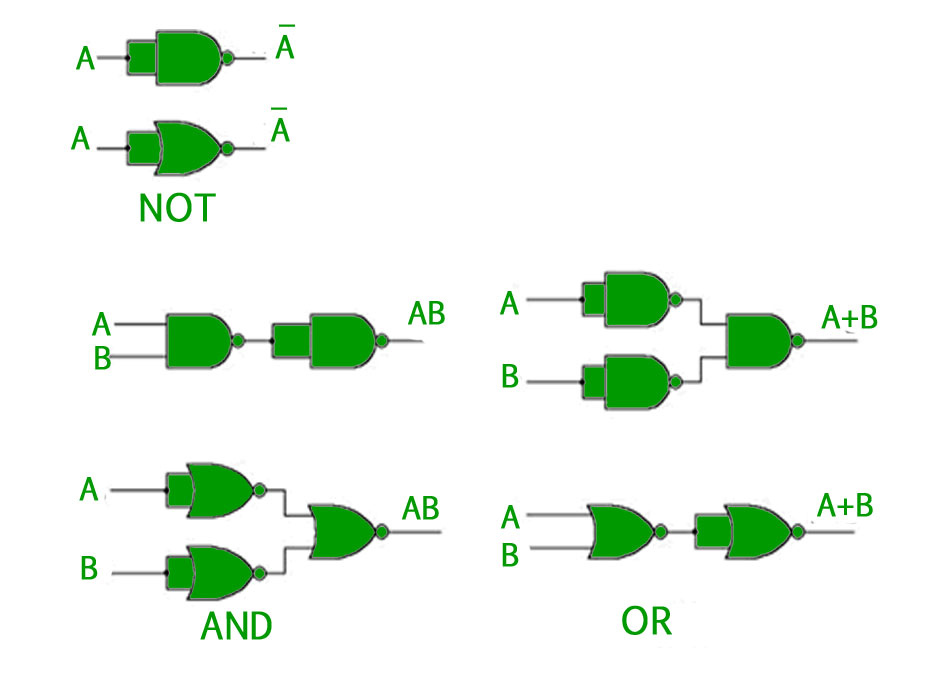
Three more logic gates are obtained if the output of above-mentioned gates is negated.

* **NAND gate()-** The NAND gate (negated AND) gives an output of 0 if both inputs are 1, it gives 1 otherwise.
* **NOR gate()-** The NOR gate (negated OR) gives an output of 1 if both inputs are 0, it gives 0 otherwise.
* **XNOR gate()-** The XNOR gate (negated XOR) gives an output of 1 both inputs are same and 0 if both are different.

Every Logic gate has a graphical representation or symbol associated with it. Below is an image which shows the graphical symbols and truth tables associated with each logic gate.

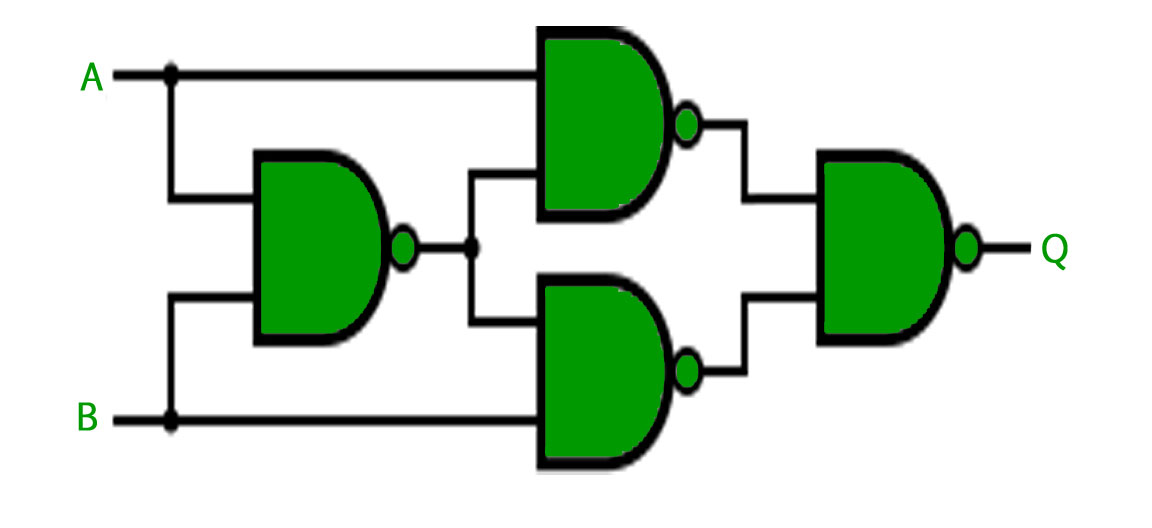
**Universal Logic Gates –**  
Out of the seven logic gates discussed above, NAND and NOR are also known as **universal gates** since they can be used to implement any digital circuit without using any other gate. This means that every gate can be created by NAND or NOR gates only.  
Implementation of three basic gates using NAND and NOR gates is shown below –



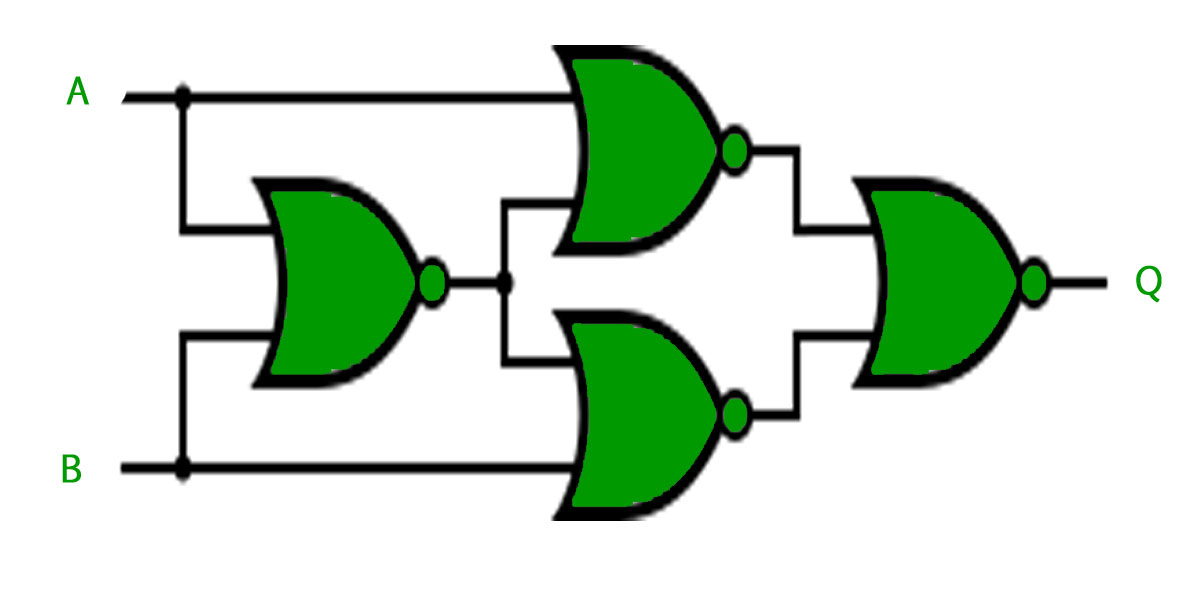


For the **XOR gate**, NAND and NOR implementation is –

* Implemented **Using NAND –**



* Implemented **using NOR –**



**Note –** For implementing XNOR gate, a single NAND or NOR gate can be added to the above circuits to negate the output of the XOR gate.

# Properties of Boolean Algebra:

**Switching algebra** is also known as **Boolean Algebra**. It is used to analyze digital gates and circuits. It is logic to perform mathematical operation on binary numbers i.e., on ‘0’ and ‘1’. Boolean Algebra contains basic operators like AND, OR and NOT etc. Operations are represented by ‘.’ for AND , ‘+’ for OR . Operations can be performed on variables which are represented using capital letter eg ‘A’ , ‘B’ etc.

**Properties of switching algebra**

* **Annulment law –** a variable AND ed with 0 gives 0, while a variable ORed with 1 gives 1, i.e.,

A.0 = 0  
A + 1 = 1

* **Identity law –** in this law variable remain unchanged it is ORed with ‘0’ or ANDed with ‘1’, i.e.,

A.1 = A  
A + 0 = A

* **Idempotent law –**a variable remain unchanged when it is ORed or ANDed with itself, i.e.,

A + A = A  
A.A = A

* **Complement law –** in this Law if a complement is added to a variable it gives one, if a variable is multiplied with its complement it results in ‘0’, i.e.,

A + A’ = 1  
A.A’ = 0

* **Double negation law –** a variable with two negation its symbol gets cancelled out and original variable is obtained, i.e.,

((A)’)’=A

* **Commutative law –** a variable order does not matter in this law, i.e.,

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

* **Associative law –** the order of operation does not matter if the priority of variables are same like ‘\*’ and ‘/’, i.e.,

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

* **Distributive law –** this law governs opening up of brackets, i.e.,

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

* **Absorption law –**:-This law involved absorbing the similar variables, i.e.,

A.(A+B) = A  
A + AB = A

* **De Morgan law –** the operation of an AND or OR logic circuit is unchanged if all inputs are inverted, the operator is changed from AND to OR, the output is inverted, i.e.,

(A.B)’ = A’ + B’  
(A+B)’ = A’.B’

**LOGIC GATES :**

A Digital logic Gate is an electronic circuit which makes logical decisions based on the combination of digital signals present on its inputs.

More than One inputs like A,B,C,D. Only one output.

TTL - **TRANSISTOR TRANSISTOR LOGIC – 7400 SERIES**

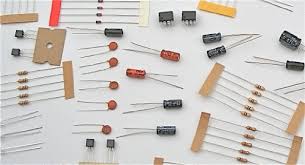
CMOS – **COMPLEMENTARY METAL OXIDE SILICON – 4000 SERIES OF CHIPS**

**TTL LOGIC IC’S** – NPN AND PNP TYPE BIPOLAR FUNCTION TRANSISOR.

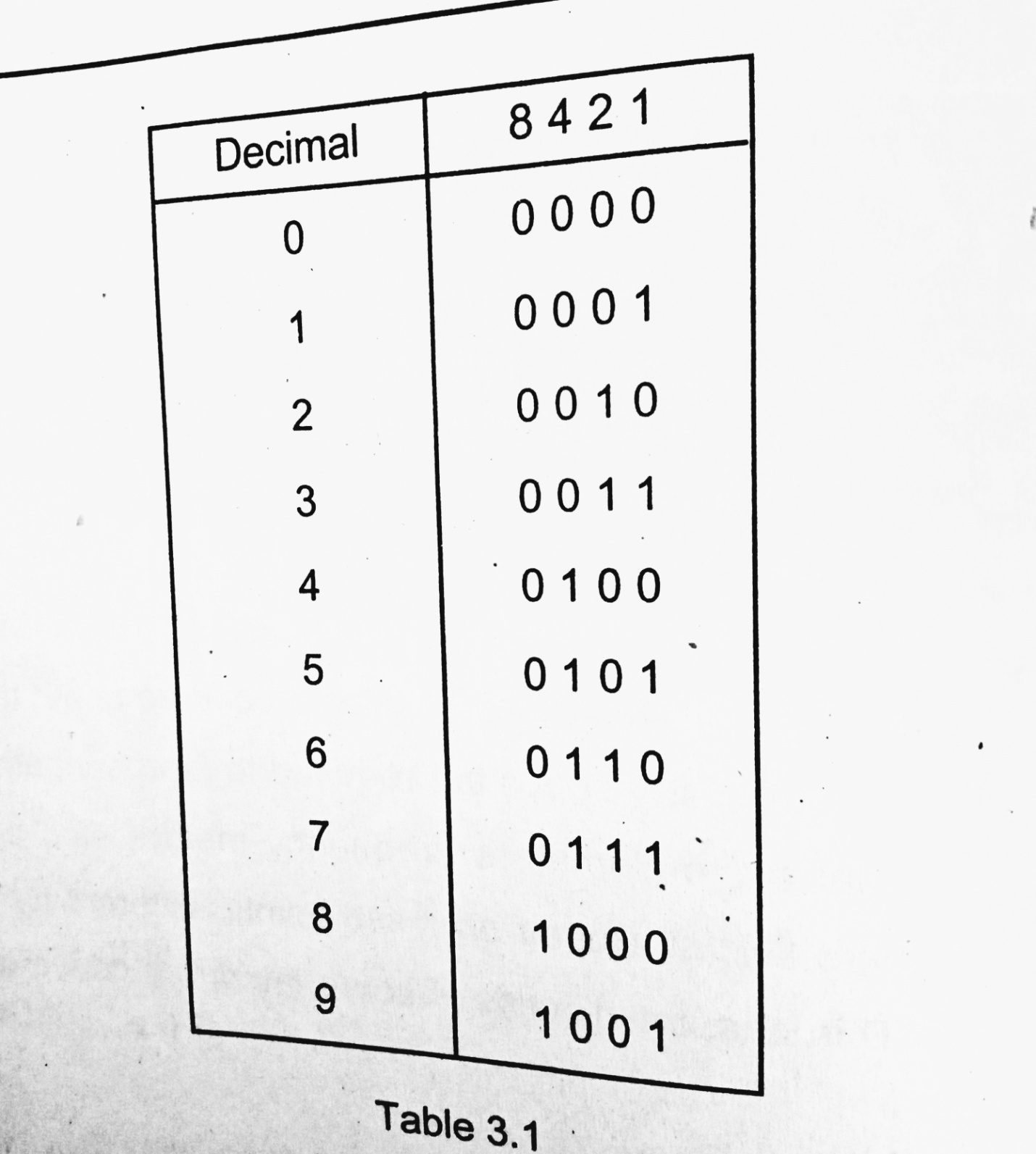
**CMOS LOGIC IC’S** – COMPLEMTARY MOSFET OR JFET TYPE FIELD TRANSISORS(INPUT AND OUTPUT)

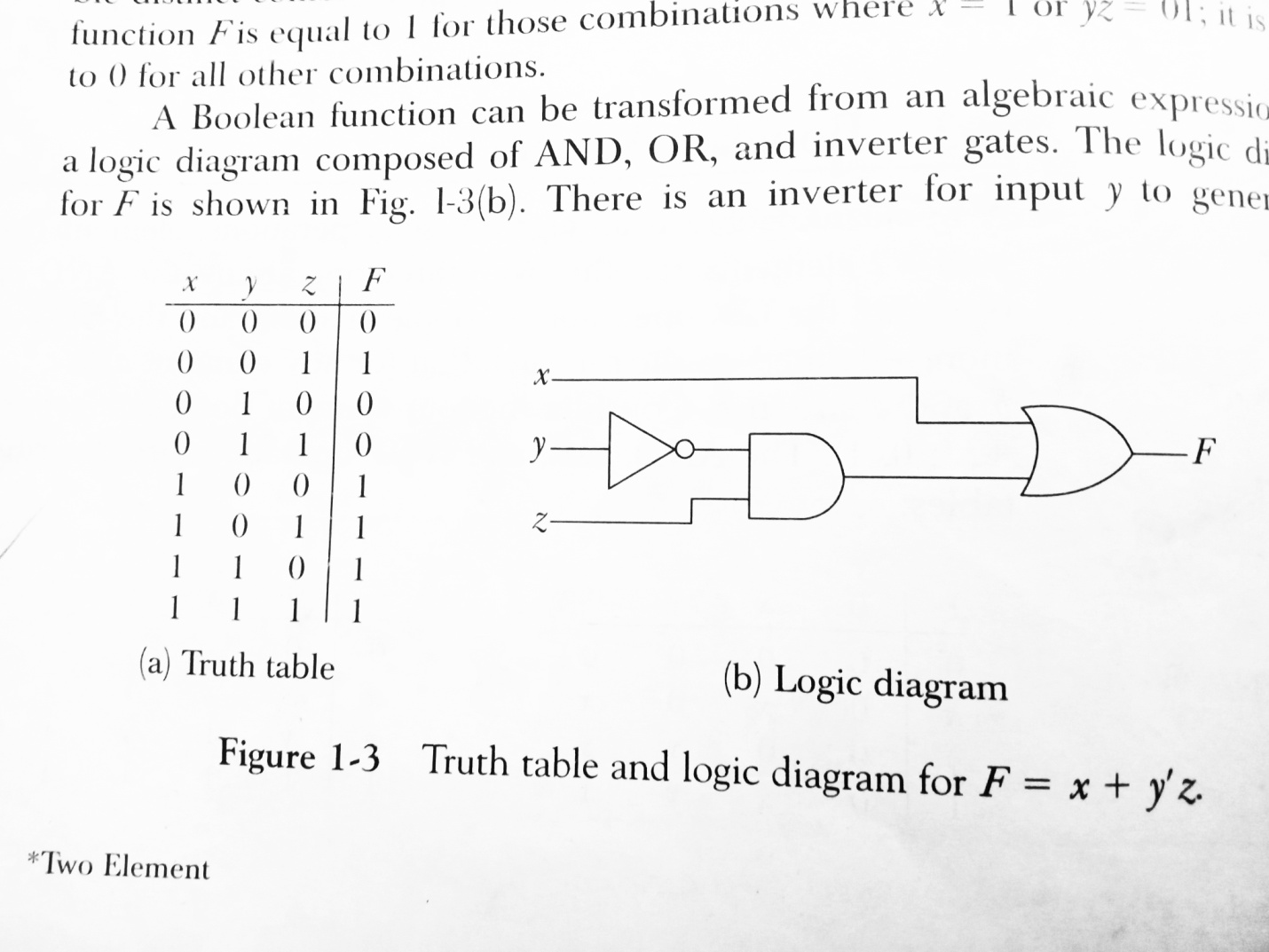
DIODES,,TRANSISORS AND RESISOTORS

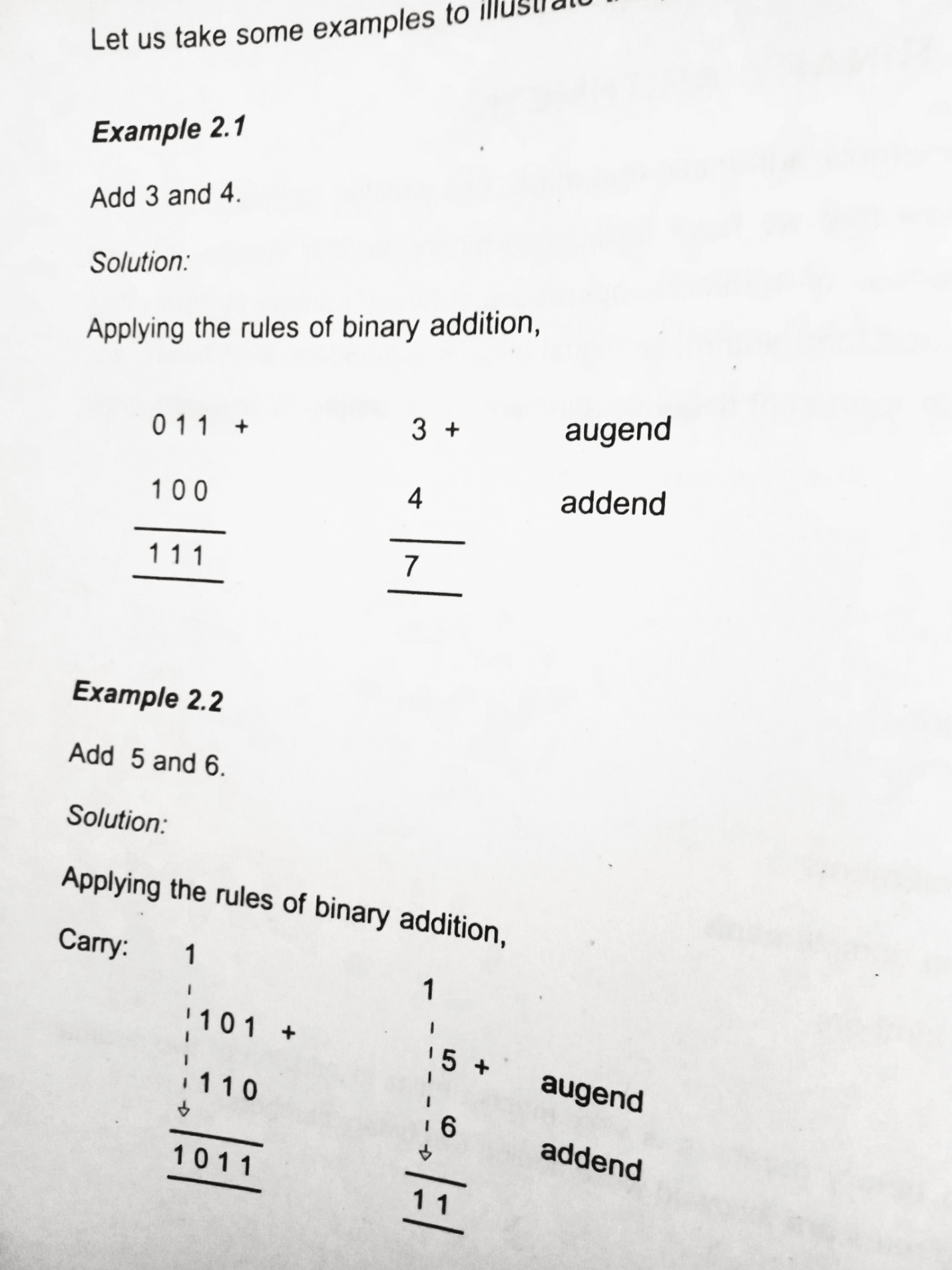




**BCD – Binary Coded Decimal**



****

****

**BASIC IDENTITIES OF BOOLEAN ALGEBRA :**

**Rule 1a**  : A+ 0 = A 1b : A.1 =A

**Rule 2a** : A+1 = 1 2b : A.0 =0

**Rule 3a :** A+A =A 3b : A.A = A

Rule 4a : A+Ā = 1 4b : A.Ā = 0

Rule 5a : = A 5a : = A

Rule 6a : A + AB =A 6a : A (A+B) = A

Rule 7a : A+ = A+B 7a : A( + B) = A.B

Rule 8a : A + BC = (A+B) (A+C) 8b : A(B+C) = AB +BC

Rule 9a : AB + +BC = AB + 9b : (A+B)( +C)(B+C) = (A+B) ( +C)

# Representation of Boolean Functions:

A **Boolean function** is described by an algebraic expression consisting of binary variables, the constants 0 and 1, and the logic operation symbols .

# Binary Representations in Digital Logic

[Binary](https://www.geeksforgeeks.org/binary-representation-of-a-given-number/) is a base-2 number system that uses two states 0 and 1 to represent a number. We can also call it to be a true state and a false state. A binary number is built the same way as we build the normal [decimal number](https://www.geeksforgeeks.org/number-system-and-base-conversions/).

For example, a decimal number 45 can be represented as 4\*10^1+5\*10^0 = 40+5

Now in binary 45 is represented as 101101. As we have powers of 10 in decimal number similarly there are powers of 2 in binary numbers. Hence 45 which is 101101 in binary can be represented as:

2^0\*1+2^1\*0+2^2\*1+2^3\*1+2^4\*0+2^5\*1 = 45

The binary number is traversed from left to right.

**Sign and Magnitude representation –**  
There are many ways for representing negative integers. One of the way is sign-magnitude. This system uses one bit to indicate the sign. Mathematical numbers are generally made up of a sign and a value. The sign indicates whether the number is positive, (+) or negative, (–) while the value indicates the size of the number.

For example 13, +256 or -574. Presenting numbers is this way is called sign-magnitude representation since the left most digit can be used to indicate the sign and the remaining digits the magnitude or value of the number.

Sign-magnitude notation is the simplest and one of the most common methods of representing positive and negative numbers. Thus negative numbers are obtained simply by changing the sign of the corresponding positive number, for example, +2 and -2, +10 and -10, etc. Similarly adding a 1 to the front of a binary number is negative and a 0 makes it positive.

For example 0101101 represents +45 and 1101101 represents -45 if 6 digits of a binary number are considered and the leftmost digit represents the sign.

But a problem with the sign-magnitude method is that it can result in the possibility of two different bit patterns having the same binary value. For example, +0 and -0 would be 0000 and 1000 respectively as a signed 4-bit binary number. So using this method there can be two representations for zero, a positive zero 0000 and also a negative zero 1000 which can cause big complications for computers and digital systems.

The 2 *compliment* notations used to represent signed magnitude numbers are:

1. **One’s compliment –**  
   One’s Complement is a method which can be used to represent negative binary numbers in a signed binary number system. In one’s complement, positive numbers remain unchanged as before.

Negative numbers however, are represented by taking the one’s complement of the unsigned positive number. Since positive numbers always start with a 0, the complement will always start with a 1 to indicate a negative number.

The one’s complement of a negative binary number is the complement of its positive, so to take the one’s complement of a binary number, all we need to do is subtract 1’s equal to the number of digits present in the number from that number. This can also be achieved by just interchanging the digits of the number. Thus the one’s complement of 1 is 0 and vice versa.

1. For example One’s Compliment of 1010100:
2. 1111111
3. -1010100

0101011

*The one’s compliment of number can also be obtained by just interchanging the digits of the binary number*.

1. **Two’s compliment –**  
   Two’s Complement is another method like one’s complement form, which we can use to represent negative binary numbers in a signed binary number system. In two’s complement, the positive numbers are exactly the same as before for unsigned binary numbers. A negative number, however, is represented by a binary number, which when added to its corresponding positive equivalent results in zero.

In two’s complement representation, a negative number is the 2’s complement of its positive number. If the subtraction of two numbers is X – Y then it can be represented as X + (2’s complement of Y).

*The two’s complement is one’s complement + 1 of a number in binary*.

The main **advantage** of two’s complement over the previous one’s complement is that there is no double-zero problem and it is a lot easier to generate the two’s complement of a signed binary number. In two’s compliment arithmetic operations are relatively easier to perform when the numbers are represented in the two’s complement format.

For example to represent -27   
27 in binary is: 00011011

11111111

-00011011

11100100 <-- 1's Compliment

+1

11100101 <-- The 2's Compliment

The above 2 are the formats that can be too long practically. So real number representations are used.

**Real number representations –**  
The goal is to represent a number with a decimal point in binary using the form. IEEE 754 standard defines how to encode a real number. This standard offers a way to code a number using 32 bits (as well as 64 bits), and defines three components:

1. The plus/minus sign is represented by one bit, the highest-weighted bit (furthest to the left).
2. The exponent is encoded using 8 bits (11 bits in 64 bit representation) immediately after the sign.
3. The mantissa (the bits after the decimal point) with the remaining 23 bits(52 bits in 64 bit representation).

# Introduction of K-Map (Karnaugh Map)

In many digital circuits and practical problems we need to find expression with minimum variables. We can minimize Boolean expressions of 3, 4 variables very easily using K-map without using any Boolean algebra theorems. K-map can take two forms Sum of Product (SOP) and Product of Sum (POS) according to the need of problem. K-map is table like representation but it gives more information than TRUTH TABLE. We fill grid of K-map with 0’s and 1’s then solve it by making groups.

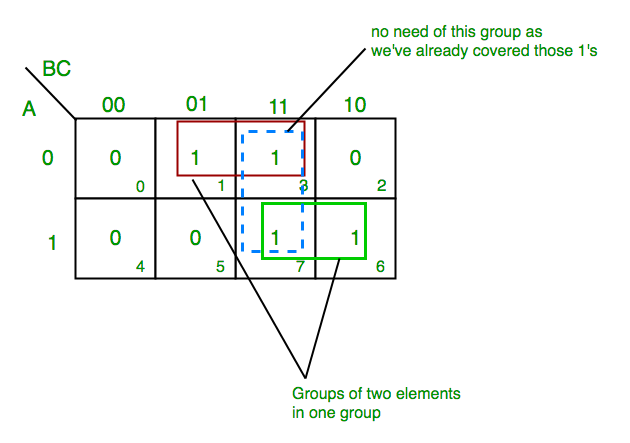
**Steps to solve expression using K-map-**

1. Select K-map according to the number of variables.
2. Identify minterms or maxterms as given in problem.
3. For SOP put 1’s in blocks of K-map respective to the minterms (0’s elsewhere).
4. For POS put 0’s in blocks of K-map respective to the maxterms(1’s elsewhere).
5. Make rectangular groups containing total terms in power of two like 2,4,8 ..(except 1) and try to cover as many elements as you can in one group.
6. From the groups made in step 5 find the product terms and sum them up for SOP form.

## ****SOP FORM****

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

Z= ∑A,B,C(1,3,6,7)



From **red** group we get product term—

A’C

From**green** group we get product term—

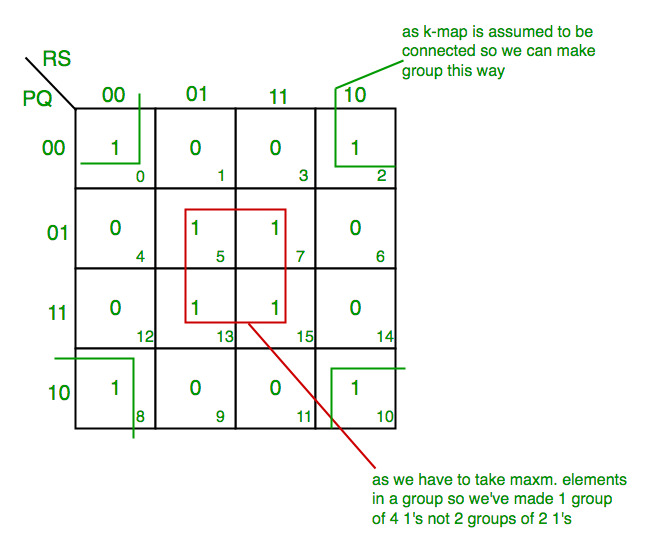
AB

Summing these product terms  we get-

**Final expression (A’C+AB)**

1. **K-map for 4 variables**

F(P,Q,R,S)=∑(0,2,5,7,8,10,13,15)



From **red** group we get product term—

QS

From **green** group we get product term—

Q’S’

umming  these product terms  we get- **Final expression (QS+Q’S’)**