

MAR GREGORIOS COLLEGE OF ARTS & SCIENCE

Block No.8, College Road, Mogappair West, Chennai – 37

Affiliated to the University of Madras
Approved by the Government of Tamil Nadu
An ISO 9001:2015 Certified Institution



DEPARTMENT OF ELECTRONICS & COMMUNICATION SCIENCE

SUBJECT NAME: DIGITAL ELECTRONICS

SUBJET CODE: TAG4C

SEMESTER: IV

PREPARED BY: PROF.V.SAVITHRI / PROF.S.SHANTHA

UNIT I

NUMBER SYSTEMS AND CODES

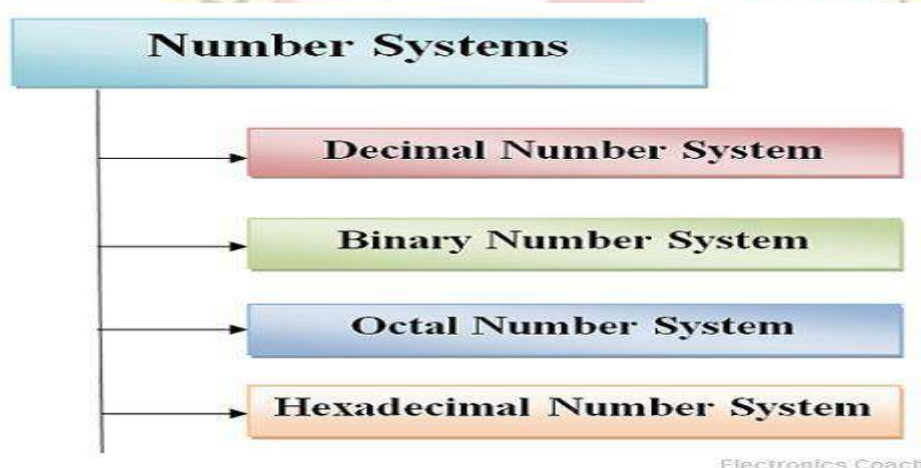
INTRODUCTION

Computers understand **machine language**. Every letter, symbol etc. that we write in the instructions given to computer, it gets converted into machine language. This machine language comprises of numbers. In order to understand the language used by computers and other digital system it is crucial to have a better understanding of number system.

NUMBERSYSTEMS

A number system relates quantities and symbols. The base or radix of a numbersystem represents the number of digits or basic symbols in that particular numbersystem. In decimal system the base is 10, because of use the numbers 0,1,2,3,4,5,6,7,8 and 9.

the classification of numbers systems on the basis of base can be understood from the below diagram.



Decimal Number System

Decimal Number System comprises of 10 digits. These digits are 0, 1, 2, 3, 4, 5, 6, 7, 8 & 9. The base of the decimal number system is 10 because total 10 digits are available in the number system. It does not implies that only 10 digits can be expressed in decimal numbers system but using these 10 digits we can define any number in this system no matter how large it is.

Two significant terms are associated with any number and that are its place value and face value. Face value is the digit itself while the place value is the magnitude that the digit represents.

Consider a number 7896, in this number the face value of 8 is 8 while its place value is 100.

Expansion of Decimal Number: Consider the above number again i.e. 7896.

Now, it can be written as:-

$$7896 = 10007 + 1008 + 109 + 6$$

To write a number in terms of base 10, we can use the positional value as superscript of base 10.

$$7896 = 10^37 + 10^28 + 10^19 + 10^06$$

Binary Number System

Binary Number System consists of two digits only i.e. 0 & 1. This makes it less complicated than any other number system since it comprises of only two digits. Thus, the base of the binary system is 2 as the available digits in this number system is 2. The other numbers can be expressed with these two digits.

Binary digits are called as **Bits**, it is formed from two words **Binary** and **Digits**. 4 Bits together are called as nibble and 8 bits together is called as byte. Binary digits are useful for computation of results of devices which have two states ON and OFF

Decimal equivalent of Binary can be calculated by multiplying the binary digits with 2 to the power of positional values of respective digits.

$$\begin{aligned} 10011 &= 2^41 + 2^30 + 2^2 + 2^11 + 2^01 \\ &= 16 + 2 + 1 \\ &= 19 \end{aligned}$$

Octal Number System

Octal Number System comprises of 8 digits i.e. from 0, 1, 2, 3, 4, 5, 6 & 7. Thus, the base of Octal Number System is 8. It is much easier to handle array of octal numbers in comparison to binary numbers. This is because if we represent any number using binary digits it will be a long array of binary numbers. While in case of Octal numbers the array of numbers will be less.

Decimal Equivalent of Octal Number: The decimal equivalent of Octal number can be evaluated by multiplying the digits with 8, and the base 8 will be raised to the positional value of the respective digit.

Let's consider an octal number 431, now its decimal equivalent can be described as:-

$$\begin{aligned} 431 &= 8^24 + 8^13 + 8^01 \\ &= 256 + 24 + 1 \\ &= 281 \end{aligned}$$

Hexadecimal Number System

It comprises of **10 digits** and **6 alphabets**. The 10 digits from 0, 1, 2, 3, 4, 5, 6, 7, 8 & 9 and the alphabets used are A, B, C, D, E & F. All the other numbers can be expressed with the

help of combination of these digits and alphabets. A, B, C, D, E, F represents 10, 11, 12, 13, 14 & 15 respectively.

The base of the hexadecimal number system is 16 as total 16 elements are available in this number system. Thus, Hexadecimal number system is used mostly in case of **microprocessors** and **microcontrollers**.

Decimal Equivalent of Hexadecimal Numbers: Let's consider a hexadecimal number 5B52, now its decimal equivalent can be calculated by multiplying each digit with 16 and 16 will be raised to the power of positional value of the respective digit.

$$\begin{aligned} 5B52 &= 16^3 5 + 16^2 11 + 16^1 5 + 16^0 2 \\ &= 54096 + 11256 + 80 + 2 \\ &= 23378 \end{aligned}$$

Binary to decimal conversion:

A binary number system is a code that uses only two basic symbols. The digits can be any two distinct characters, but it should be 0 or 1. The binary equivalent for some decimal numbers are given below

decimal	0	1	2	3	4	5	6	7	8	9	10	11
binary	0	1	10	11	100	101	110	111	1000	1001	1010	1011

Each digit in a binary number has a value or weight. The LSB has a value of 1. The second from the right has a value of 2, then next 4, etc.,

16	8	4	2	1
2^4	2^3	2^2	2^1	2^0

Binary to decimal conversion:

$$(1001)_2 = X_{10}$$

$$\begin{aligned} 1001 &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &= 8 + 0 + 0 + 1 \end{aligned}$$

$$(1001)_2 = (9)_{10}$$

Fractions:

For fractions the weights of the digit positions are written from right of the binary point and weights are given as follows.

2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
----------	----------	----------	----------	----------

g

$$(0.0110)_2 = X_{10}$$

$$= 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} + 0 \times 2^{-4}$$

$$= 0 \times 0.5 + 1 \times 0.25 + 1 \times 0.125 + 0 \times 0.0625$$

$$= (0.375)_{10}$$

$$(1011.101)_2 = X_{10}$$

$$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3}$$

$$= 8 + 0 + 2 + 1 + 0.5 + 0 + 0.125$$

$$= (11.625)_{10}$$

Decimal to binary conversion:**(Double Dabble method)**

In this method the decimal number is divided by 2 progressively and the remainder is written after each division. Then the remainders are taken in the reverse order to form the binary number.

$$\begin{array}{r}
 2 \overline{) 12} \\
 \underline{2 } \\
 2 \\
 \underline{2 } \\
 0 \\
 \underline{0} \\
 0 \\
 \underline{0} \\
 0 \\
 \underline{0} \\
 0 \\
 \underline{0} \\
 0
 \end{array}
 \begin{array}{l}
 - 0 \\
 - 0 \\
 - 1
 \end{array}$$

$$(12)_{10} = X_2$$

$$(12)_{10} = (1100)_2$$

$$(21)_2 = X_2$$

$$\begin{array}{r}
 2 \overline{) 21} \\
 \underline{2 } \\
 2 \overline{) 10} \quad -1 \\
 \underline{2 } \\
 2 \overline{) 5} \quad -0 \\
 \underline{2} \\
 2 \overline{) 2} \quad -1 \\
 \underline{2} \\
 1 \quad -0
 \end{array}$$

$$(21)_{10} = (10101)_2$$

Fractions:

The fraction is multiplied by 2 and the carry in the integer position is written after each multiplication. Then they are written in the forward order to get the corresponding binary equivalent.

E.g.:

$$(0.4375)_{10} = X_2$$

$$2 \times 0.4375 = 0.8750 \Rightarrow 0$$

$$2 \times 0.8750 = 1.7500 \Rightarrow 1$$

$$2 \times 0.7500 = 1.5000 \Rightarrow 1$$

$$2 \times 0.5000 = 1.0000 \Rightarrow 1$$

$$(0.4375)_{10} = (0.0111)_2$$

Octal Number System

Octal number system has a base of 8 i.e., it has eight basic symbols. First eight decimal digits 0, 1, 2, 3, 4, 5, 6, 7 are used in this system.

Octal to decimal conversion:

In the octal number system each digit corresponds to the powers of 8. The weight of digital position in octal number is as follows

8^4	8^3	8^2	8^1	8^0	8^{-1}	8^{-2}	8^{-3}
-------	-------	-------	-------	-------	----------	----------	----------

To convert from octal to decimal multiply each octal digit by its weight and add the resulting products.

E.g.:

$$(48)_8 = X_{10}$$

$$\begin{aligned} 48 &= 4 \times 8^1 + 7 \times 8^0 \\ &= 32 + 7 \\ &= 39 \end{aligned}$$

$$(48)_8 = (39)_{10}$$

E.g.:

$$(22.34)_8 = X_{10}$$

$$\begin{aligned} 22.34 &= 2 \times 8^1 + 2 \times 8^0 + 3 \times 8^{-1} + 4 \times 8^{-2} \\ &= 16 + 2 + 3 \times 1/8 + 4 \times 1/64 \\ &= (18.4375) \end{aligned}$$

Decimal to octal conversion:

Here the number is divided by 8 progressively and each time the remainder is written and finally the remainders are written in the reverse order to form the octal number. If the number has a fraction part, that part is multiplied by 8 and carry in the integer part is taken. Finally the carries are taken in the forward order.

E.g.:

$$(19.11)_{10} = X_8$$

$$\begin{array}{r} 8 \overline{) 19} \\ \underline{2 } \\ 2 - 3 \end{array}$$

$$0.11 \times 8 = 0.88 \Rightarrow 0$$

$$0.88 \times 8 = 7.04 \Rightarrow 7$$

$$0.04 \times 8 = 0.32 \Rightarrow 0$$

$$0.32 \times 8 = 2.56 \Rightarrow 2$$

$$0.56 \times 8 = 4.48 \Rightarrow 4$$

$$(19.11)_{10} = (23.07024)_8$$

Octal to binary conversion:

Since the base of octal number is 8, i.e., the third power of 2, each octal number is converted into its equivalent binary digit of length three.

E.g.:

$$(57.127)_8 = X_2$$

$$\begin{array}{ccccccc} 5 & 7 & . & 1 & 2 & 7 & \\ 101 & 111 & . & 001 & 010 & 111 & \end{array}$$

$$(57.127)_8 = (101111001010111)_2$$

Binary to octal:

The given binary number is grouped into a group of 3 bits, starting at the octal point and each group is converted into its octal equivalent.

E.g.:

$$(1101101.11101)_2 = X_8$$

$$\begin{array}{ccccccc} 001 & 101 & 101 & . & 111 & 010 & \\ 1 & 5 & 5 & . & 7 & 2 & \end{array}$$

$$(1101101.11101)_2 = (155.72)_8$$

The hexadecimal number system has a base of 16. It has 16 symbols from 0 through 9 and A through F.

Decimal	Hexadecimal	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111

Binary to hexadecimal:

The binary number is grouped into bits of 4 from the binary point then the corresponding hexadecimal equivalent is written.

E.g.:

$$(100101110.11011)_2 = X_{16}$$

$$0001\ 0010\ 1110.1101\ 1000$$

$$1\quad 2\quad E.D\ 8$$

$$(100101110.11011)_2 = (12E.D8)_{16}$$

Hexadecimal to binary:

Since the base of hexadecimal number is 16, i.e., the fourth power of 2, each hexadecimal number is converted into its equivalent binary digit of length four.

E.g.:

$$(5D.2A)_{16} = X_2$$

$$5\quad D\quad .\quad 2\quad A$$

$$0101\ 1101.0010\ 1010$$

$$(5D.2A)_{16} = (01011101.00101010)_2$$

Decimal to hexadecimal:

The decimal number is divided by 16 and carries are taken after each division and then written in the reverse order. The fractional part is multiplied by 16 and carry is taken in the forward order.

E.g.:

$$(2479.859)_{10} = X_{16}$$

$$\begin{array}{r} 16 \overline{) 2479} \\ \underline{16} \\ 154 \\ \underline{154} \\ 9 \\ \underline{9} \\ 0 \end{array}$$

- 15(F)
↑
- 10(A)

$$16 \times 0.859 = 13.744 \Rightarrow 13 (D)$$

$$16 \times 0.744 = 11.904 \Rightarrow 11 \text{ (B)}$$

$$16 \times 0.904 = 14.464 \Rightarrow 14 \text{ (E)}$$

$$16 \times 0.464 = 7.424 \Rightarrow 7$$

$$16 \times 0.424 = 6.784 \Rightarrow 6$$

$$(2479.859)_{10} = (9AF.DBE76)_{16}$$

Hexadecimal to decimal:

Each digit of the hexadecimal number is multiplied by its weight and then added.

E.g.:

$$\begin{aligned} (81.21)_{16} &= X_{10} \\ &= 8 \times 16^1 + 1 \times 16^0 + 2 \times 16^{-1} + 1 \times 16^{-2} \\ &= 8 \times 16 + 1 \times 1 + 2/16 + 1/16^2 \\ &= (129.1289)_{10} \end{aligned}$$

$$(81.21)_{16} = (129.1289)_{10}$$

Codes

In the coding, when numbers or letters are represented by a specific group of symbols, it is said to be that number or letter is being encoded. The group of symbols is called as **code**. The digital data is represented, stored and transmitted as group of bits. This group of bits is also called as **binary code**.

Binary codes can be classified into two types.

- Weighted codes
- Unweighted codes

If the code has positional weights, then it is said to be **weighted code**. Otherwise, it is an unweighted code. Weighted codes can be further classified as positively weighted codes and negatively weighted codes.

Binary Codes for Decimal digits

The following table shows the various binary codes for decimal digits 0 to 9.

Decimal Digit	8421 Code	2421 Code	84-2-1 Code	Excess 3 Code
0	0000	0000	0000	0011
1	0001	0001	0111	0100
2	0010	0010	0110	0101
3	0011	0011	0101	0110
4	0100	0100	0100	0111
5	0101	1011	1011	1000
6	0110	1100	1010	1001
7	0111	1101	1001	1010
8	1000	1110	1000	1011
9	1001	1111	1111	1100

We have 10 digits in decimal number system. To represent these 10 digits in binary, we require minimum of 4 bits. But, with 4 bits there will be 16 unique combinations of zeros and ones. Since, we have only 10 decimal digits, the other 6 combinations of zeros and ones are not require

Commonly used Binary Codes

Before going into the details of individual binary codes, let us quickly take a look at some of the commonly used Binary Codes. The following is the list:

8421 Codes

2421 Codes

5211 Codes

Excess-3 Codes

Gray Codes

In the above list, the first three i.e., 8421, 2421 and 5211 are Weighted Binary Codes while the other two are Non-Weighted Binary Codes.

Weighted Binary Systems

The values assigned to consecutive places in the decimal system which is a place value system are 10^4 , 10^3 , 10^2 , 10^1 , 10^0 , 10^{-1} , 10^{-2} , 10^{-3} ... and so on from left to right. It is easily can be understood that the weight of digit of the decimal system is '10'.

For example:

$$(3546.25)_{10} = 3 \times 10^3 + 5 \times 10^2 + 4 \times 10^1 + 6 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$$

In the same way, the values assigned to consecutive places in the binary system, which is also a place value system, are called as weighted binary system.

The weights in a binary system are $2^4, 2^3, 2^2, 2^1, 2^0, 2^{-1}, 2^{-2}, 2^{-3} \dots$ from left to right. It is easily can be understood that the weight of digit of the binary system is '2'.

For example:

$$(1110110)_2 = 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$

$$= 64 + 32 + 16 + 0 + 4 + 2 + 0 = (118)_{10}$$

8421 Code or BCD Code

The decimal numbers 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 can be expressed in Binary numbers as shown in the following table. All these binary numbers again expressed in the last column by expanding into 4 bits. As per the weighted binary digits, the 4 Bit binary numbers can be expressed according to their place value from left to right as 8421 ($2^3 2^2 2^1 2^0 = 8421$).

DECIMAL NUMBER	BINARY NUMBER	4 BIT EXPRESSION(8421)
0	0	0000
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	0101
6	110	0110
7	111	0111
8	1000	1000
9	1001	1001

There are other forms of codes which are not so popular but rather confusing. They are 2421 code, 5211 code, reflective code, sequential code, non-weighted coded, excess-3 code and Grey code. They are having their own importance for some of the exclusive applications and may be useful for some of the special applications.

2421 Code

This code also a 4 bit application code where the binary weights carry 2, 4, 2, 1 from left to right.

DECIMAL NUMBER	BINARY NUMBER	2421 CODE
0	0	0000
1	1	0001
2	10	0010
3	11	0011
4	100	0100
5	101	1011
6	110	1100
7	111	1101
8	1000	1110
9	1001	1111

5211 Code

This code is also a 4 bit application code where the binary weights carry 5, 2, 1, 1 from left to right.

DECIMAL NUMBER	BINARY NUMBER	5211 CODE
0	0	0000
1	1	0001
2	10	0011
3	11	0101
4	100	0111
5	101	1000
6	110	1010
7	111	1100

DECIMAL NUMBER	BINARY NUMBER	5211 CODE
8	1000	1110
9	1001	1111

Reflective Code

It can be observed that in the 2421 and 5211 codes, the code for decimal 9 is the complement of the code for decimal 0, the code for decimal 8 is the complement of the code for decimal 1, the code for decimal 7 is the complement of the code for decimal 2, the code for decimal 6 is the complement of the code for decimal 3, the code for decimal 5 is the complement of the code for decimal 4. These codes are called as Reflective Codes. The same can be observed in the following table

DECIMAL NUMBER	BINARY NUMBER	2421 CODE	5211 CODE
0	0	0000	0000
1	1	0001	0001
2	10	0010	0011
3	11	0011	0101
4	100	0100	0111
5	101	1011	1000
6	110	1100	1010
7	111	1101	1100
8	1000	1110	1110
9	1001	1111	

Sequential Codes

Sequential codes are the codes in which 2 subsequent numbers in binary representation differ by only one digit. The 8421 and Excess-3 codes are examples of sequential codes. 2421 and 5211 codes do not come under sequential codes.

DECIMAL NUMBER	BINARY NUMBER	8421 CODE	EXCESS-3
0	0	0000	0011
1	1	0001	0100
2	10	0010	0101
3	11	0011	0110
4	100	0100	0111
5	101	0101	1000
6	110	0110	1001
7	111	0111	1010
8	1000	1000	1011
9	1001	1001	110

Non-Weighted Codes

Some of the codes will not follow the weights of the sequence binary numbers these are called as non-weighted codes. ASCII code and Grey code are some of the examples where they are coded for some special purpose applications and they do not follow the weighted binary number calculations.

Excess-3 Code

The excess-3 code is also treated as **XS-3 code**. The excess-3 code is a non-weighted and self-complementary BCD code used to represent the decimal numbers. This code has a biased representation. This code plays an important role in arithmetic operations because it resolves deficiencies encountered when we use the 8421 BCD code for adding two decimal digits whose sum is greater than 9. The Excess-3 code uses a special type of algorithm, which differs from the binary positional number system or normal non-biased BCD.

We can easily get an excess-3 code of a decimal number by simply adding 3 to each decimal digit. And then we write the 4-bit binary number for each digit of the decimal number. We can find the excess-3 code of the given binary number by using the following steps:

We find the decimal number of the given binary number.

Then we add 3 in each digit of the decimal number.

Now, we find the binary code of each digit of the newly generated decimal number.

We can also add 0011 in each 4-bit BCD code of the decimal number for getting excess-3 code

The Excess-3 code for the decimal number is as follows:

Decimal Digit	BCD Code	Excess-3 Code
0	0000	0011
1	0001	0100
2	0010	0101
3	0011	0110
4	0100	0111
5	0101	1000
6	0110	1001
7	0111	1010
8	1000	1011
9	1001	1100

In excess-3 code, the codes 1111 and 0000 are never used for any decimal digit. Let's take some examples of Excess-3 code.

Example 1: Decimal number 31

1. We find the BCD code of each digit of the decimal number.

Digit	BCD
3	0011
1	0001

2) Then, we add 0011 in both of the BCD code.

Decimal	BCD	Excess-3
3	0011+0011	0110
1	0001+0011	0100

3. So, the excess-3 code of the decimal number 31 is **0110 0100**

Example 2: Decimal number 81.61

1. We find the BCD code of each digit of the decimal number.

Digit	BCD
8	1000
1	0001
6	0110
1	0001

2) Then, we add 0011 in both of the BCD code.

Decimal	BCD	Excess-3
8	1000+0011	1011
1	0001+0011	0100
6	0110+0011	1001

3) So, the excess-3 code of the decimal number 81.61 is **1011 0100.1001 0100**

Self-complementary property

A self-complementary binary code is a code which is always complimented in itself. By replacing the bit 0 to 1 and 1 to 0 of a number, we find the 1's complement of the number. The sum of the 1's complement and the binary number of a decimal is equal to the binary number of decimal 9.

Note: if we perform the 1's complement of excess-3 of a decimal number, it will be equal to the excess-3 code of the 9's complement of that decimal number.

For example: If we perform 1's complement of the excess-3 code 1000(decimal 5), complement value will be 0111, which is the excess-3 code of 9's complement of 5, i.e., 4(0111).

Why use Excess-3 code?

There are the following advantages of excess-3 code which make it required to use:

These codes are self-complementary.

These codes use biased representation.

The excess-3 code has no limitation, so that it considerably simplifies arithmetic operations.

The codes 0000 and 1111 can cause a fault in the transmission line. The excess-3 code doesn't use these codes and gives an advantage for memory organization.

These codes are usually unweighted binary decimal codes.

This code has a vital role in arithmetic operations. It is because it resolves deficiencies which are encountered when we use the 8421 BCD code for adding two decimal digits whose sum is greater than 9.

GRAY CODE

The **Gray Code** is a sequence of binary number systems, which is also known as **reflected binary code**. The reason for calling this code as reflected binary code is the first $N/2$ values compared with those of the last $N/2$ values in reverse order. In this code, two consecutive values are differed by one bit of binary digits. Gray codes are used in the general sequence of hardware-generated binary numbers. These numbers cause ambiguities or errors when the transition from one number to its successive is done. This code simply solves this problem by changing only one bit when the transition is between numbers is done.

The gray code is a very light weighted code because it doesn't depend on the value of the digit specified by the position. This code is also called a cyclic variable code as the transition of one value to its successive value carries a change of one bit only.

How to generate Gray code?

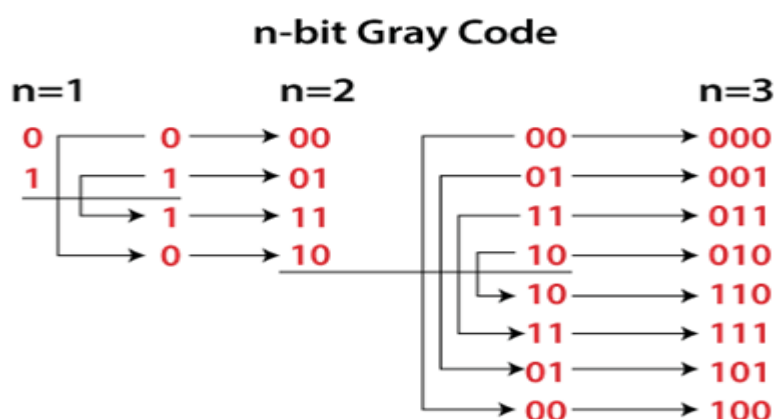
The prefix and reflect method are recursively used to generate the Gray code of a number. For generating gray code:

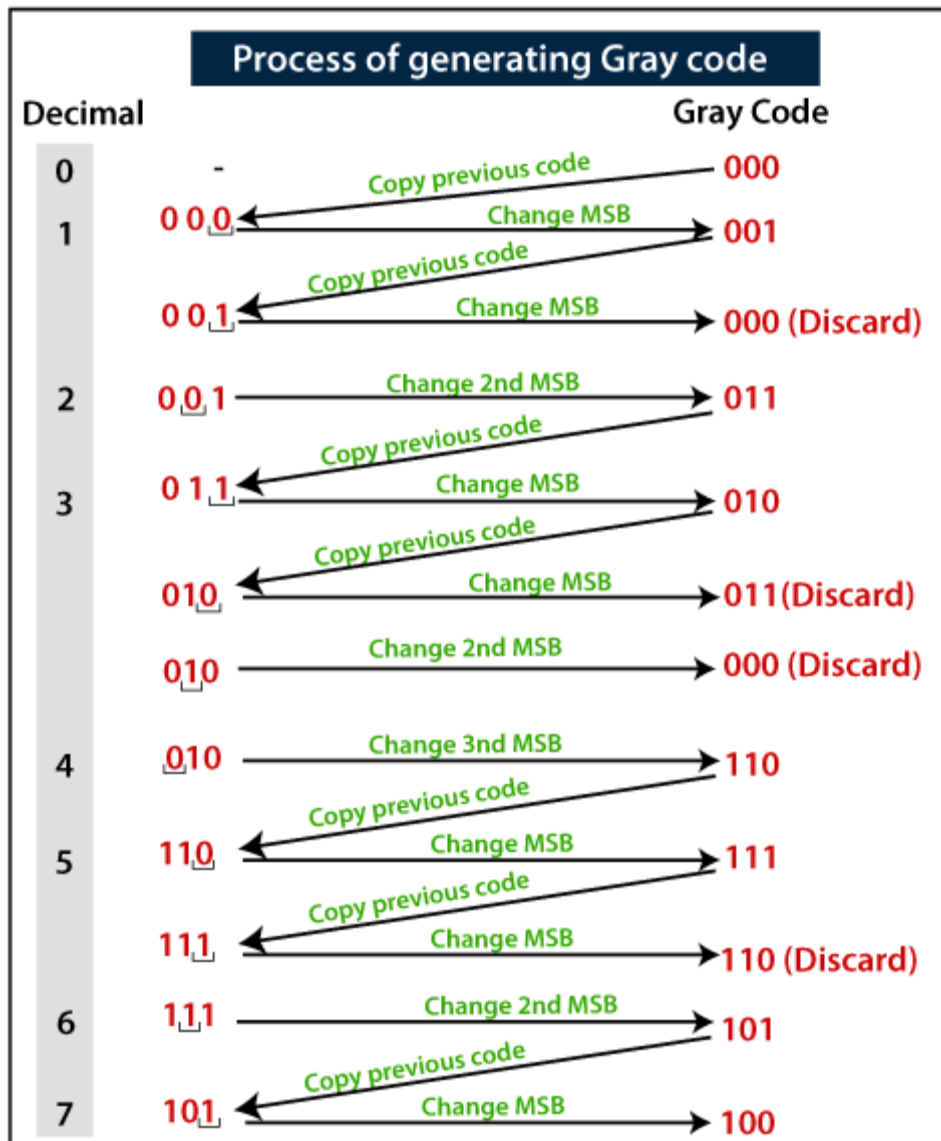
We find the number of bits required to represent a number.

Next, we find the code for 0, i.e., 0000, which is the same as binary.

Now, we take the previous code, i.e., 0000, and change the most significant bit of it.

We perform this process recursively until all the codes are not uniquely identified. If by changing the most significant bit, we find the same code obtained previously, then the second most significant bit will be changed, and so on.





Gray Code Table

Decimal Number	Binary Number	Gray Code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100

8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Binary to Gray Code Conversion

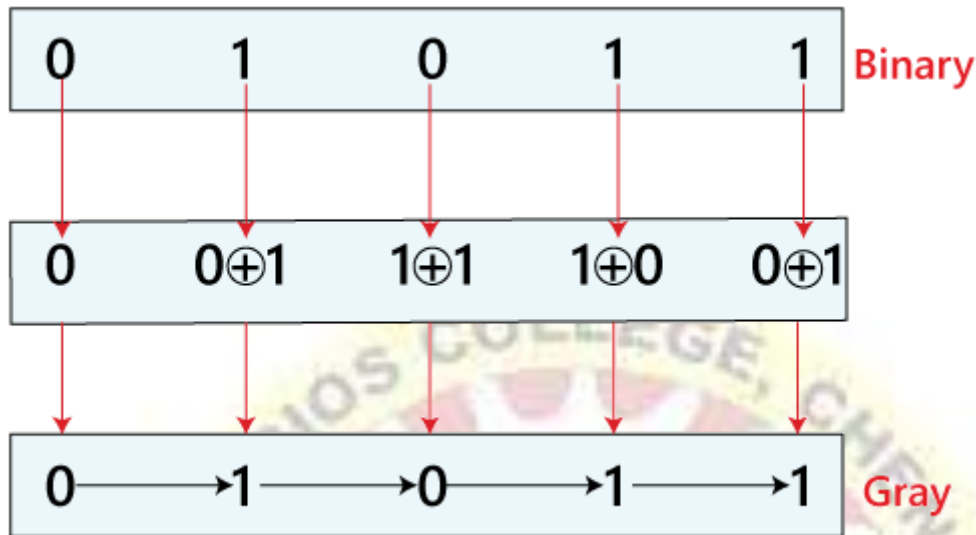
- In the Gray code, the MSB will always be the same as the 1st bit of the given binary number.
- In order to perform the 2nd bit of the gray code, we perform the exclusive-or (XOR) of the 1st and 2nd bit of the binary number. It means that if both the bits are different, the result will be one else the result will be 0.
- In order to get the 3rd bit of the gray code, we need to perform the exclusive-or (XOR) of the 2nd and 3rd bit of the binary number. The process remains the same for the 4th bit of the Gray code. Let's take an example to understand these steps.

Example

Suppose we have a binary number 01101, which we want to convert into Gray code. There are the following steps which need to perform this conversion:

- As we know that the 1st bit of the Gray code is the same as the MSB of the binary number. In our example, the MSB is 0, so the MSB or 1st bit of the gray code is 0.
- Next, we perform the XOR operation of the 1st and the second binary number. The 1st bit is 0, and the 2nd bit is 1. Both the bits are different, so the 2nd bit of the Gray code is 1.
- Now, we perform the XOR of the 2nd bit and 3rd bit of the binary number. The 2nd bit is 1, and the 3rd bit is also 1. These bits are the same, so the 3rd bit of the Gray code is 0.
- Again perform the XOR operation of the 3rd and 4th bit of binary number. The 3rd bit is 1, and the 4th bit is 0. As these are different, the 4th bit of the Gray code is 1.

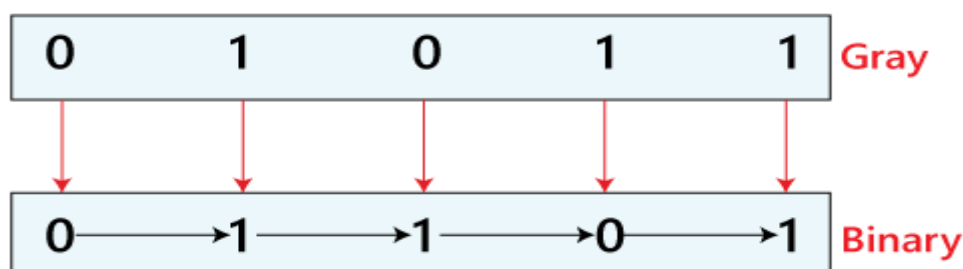
- Lastly, perform the XOR of the 4th bit and 5th bit of the binary number. The 4th bit is 0, and the 5th bit is 1. Both the bits are different, so that the 5th bit of the Gray code is 1.
- The gray code of the binary number 01101 is 01011.



Gray to Binary Code Conversion

Just like Binary to Gray code conversion; it is also a very simple process. There are the following steps used to convert the Gray code into binary.

- Just like binary to gray, in gray to binary, the 1st bit of the binary number is similar to the MSB of the Gray code.
- The 2nd bit of the binary number is the same as the 1st bit of the binary number when the 2nd bit of the Gray code is 0; otherwise, the 2nd bit is altered bit of the 1st bit of binary number. It means if the 1st bit of the binary is 1, then the 2nd bit is 0, and if it is 0, then the 2nd bit be 1.
- The 2nd step continues for all the bits of the binary number.



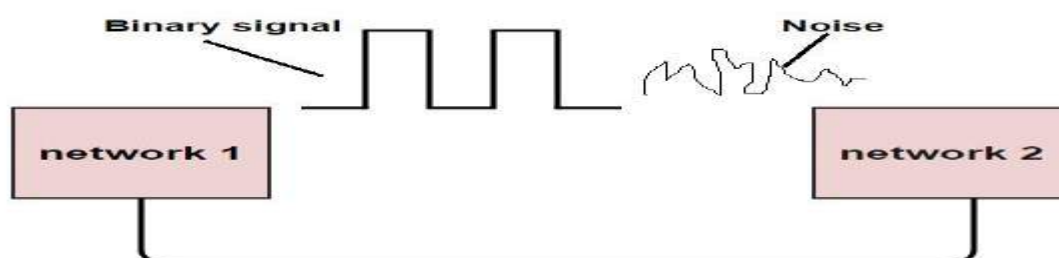
Gray Code to Binary Conversion Example

Suppose we have the Gray code 01011, which we want to convert into a binary number. There are the following steps which we need to perform for the conversion:

- The 1st bit of the binary number is the same as the MSB of the Gray code. The MSB of the Gray code is 0, so the MSB of the binary number is 0.
- Now, for the 2nd bit, we check the 2nd bit of the Gray code. The 2nd bit of the Gray code is 1, so the 2nd bit of the binary number is one that is altered number of 1st
- The next bit of the Gray code is 0; the 3rd bit is the same as the 2nd bit of the Gray code, i.e., 1.
- The 4th bit of the Gray code is 1; the 4th bit of the binary number is 0 that is the altered number of the 3rd
- The 5th bit of the Gray code is 1; the 5th bit of the binary number is 1; that is the altered number of the 4th bit of the binary number.
- So, the binary number of the Gray code 01011 is 01101.

ERROR DETECTION & CORRECTION CODES

We know that the bits 0 and 1 corresponding to two different range of analog voltages. So, during transmission of binary data from one system to the other, the noise may also be added. Due to this, there may be errors in the received data at other system.



That means a bit 0 may change to 1 or a bit 1 may change to 0. We can't avoid the interference of noise. But, we can get back the original data first by detecting whether any errors present and then correcting those errors. For this purpose, we can use the following codes.

- Error detection codes
- Error correction codes

Types Of Errors

In a data sequence, if 1 is changed to zero or 0 is changed to 1, it is called “Bit error”.

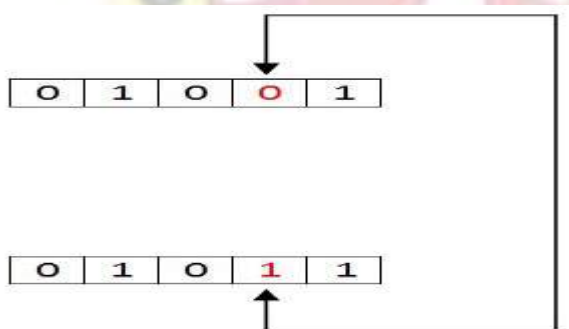
There are generally 3 types of errors occur in data transmission from transmitter to receiver.

They are

- Single bit errors
- Multiple bit errors
- Burst errors

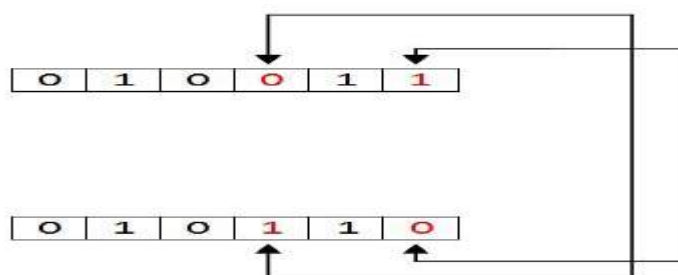
Single Bit Data Errors

The change in one bit in the whole data sequence, is called “Single bit error”. Occurrence of single bit error is very rare in serial communication system. This type of error occurs only in parallel communication system, as data is transferred bit wise in single line, there is chance that single line to be noisy.



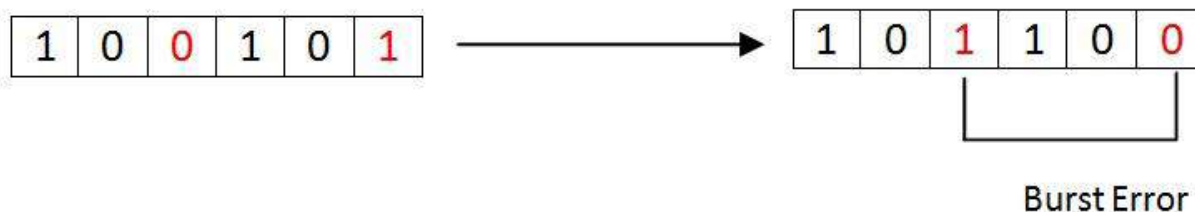
Multiple Bit Data Errors

If there is change in two or more bits of data sequence of transmitter to receiver, it is called “Multiple bit error”. This type of error occurs in both serial type and parallel type data communication networks.



Burst Errors

The change of set of bits in data sequence is called “Burst error”. The burst error is calculated in from the first bit change to last bit change.

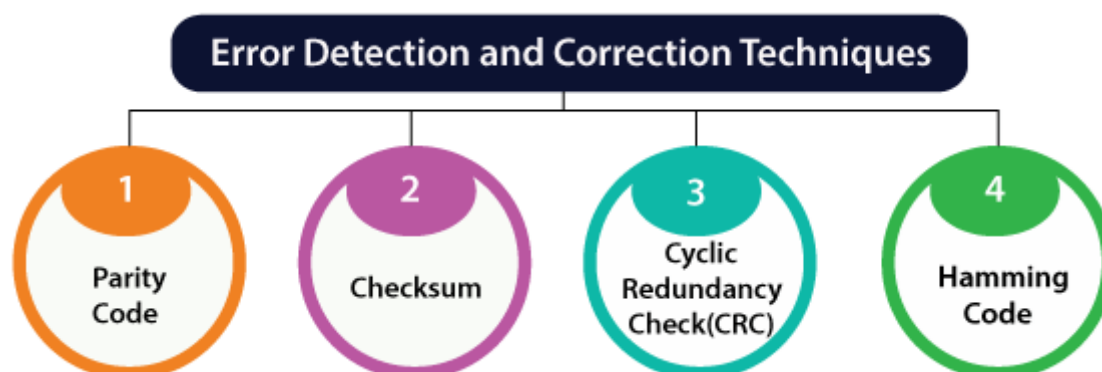


Here we identify the error from fourth bit to 6th bit. The numbers between 4th and 6th bits are also considered as error. These set of bits are called “Burst error”. These burst bits changes from transmitter to receiver, which may cause a major error in data sequence. This type of errors occurs in serial communication and they are difficult to solve.

Error Detecting Codes

In digital communication system errors are transferred from one communication system to another, along with the data. If these errors are not detected and corrected, data will be lost . For effective communication, data should be transferred with high accuracy .This can be achieved by first detecting the errors and then correcting them.

Error detection is the process of detecting the errors that are present in the data transmitted from transmitter to receiver, in a communication system. We use some redundancy codes to detect these errors, by adding to the data while it is transmitted from source (transmitter). These codes are called “Error d



etecting codes”.

Parity Checking

Parity bit means nothing but an additional bit added to the data at the transmitter before transmitting the data. Before adding the parity bit, number of 1's or zeros is calculated in the data. Based on this calculation of data an extra bit is added to the actual information / data. The addition of parity bit to the data will result in the change of data string size.

This means if we have an 8 bit data, then after adding a parity bit to the data binary string it will become a 9 bit binary data string.

Parity check is also called as "Vertical Redundancy Check (VRC)".

There is two types of parity bits in error detection, they are

- Even parity
- Odd parity

Even Parity

If the data has even number of 1's, the parity bit is 0. Ex: data is 10000001 -> parity bit 0

Odd number of 1's, the parity bit is 1. Ex: data is 10010001 -> parity bit 1

Odd Parity

If the data has odd number of 1's, the parity bit is 0. Ex: data is 10011101 -> parity bit 0

Even number of 1's, the parity bit is 1. Ex: data is 10010101 -> parity bit 1

NOTE: The counting of data bits will include the parity bit also.

The circuit which adds a parity bit to the data at transmitter is called "Parity generator". The parity bits are transmitted and they are checked at the receiver. If the parity bits sent at the transmitter and the parity bits received at receiver are not equal then an error is detected. The circuit which checks the parity at receiver is called "Parity checker".

Messages with even parity and odd parity

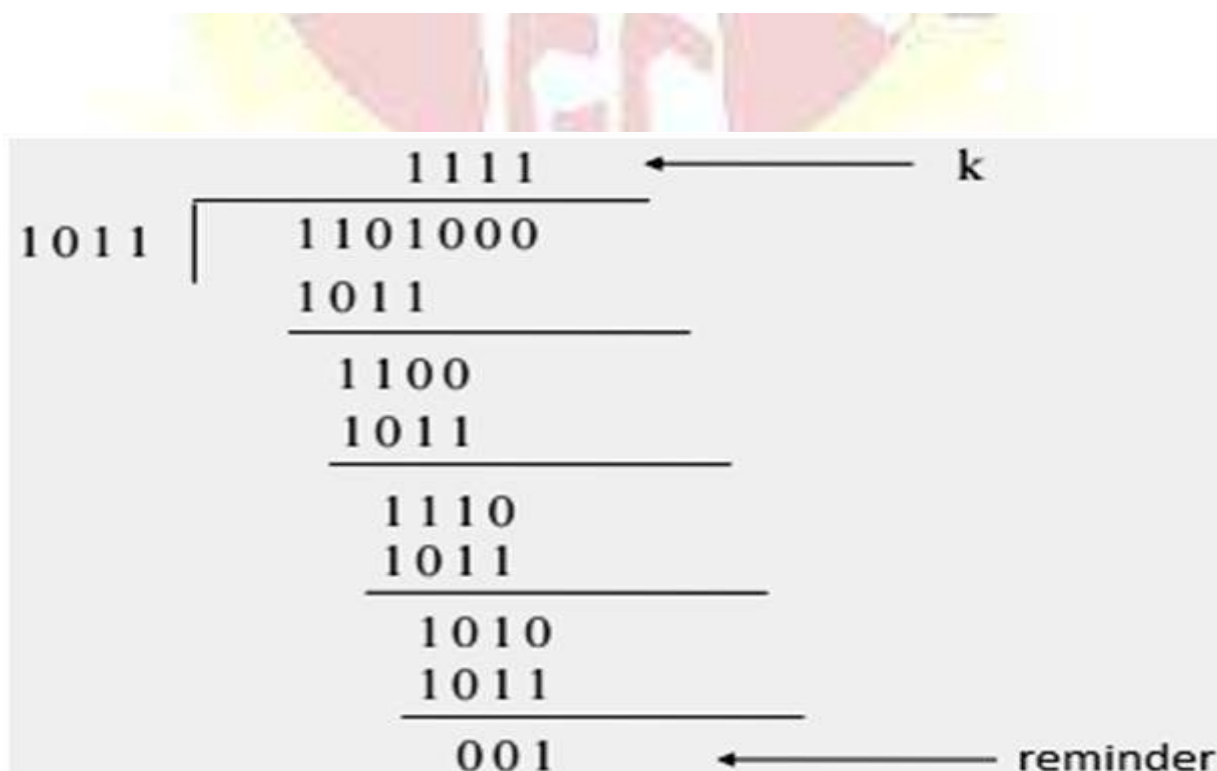
3 bit data			Message with even parity		Message with odd parity	
A	B	C	Message	Parity	Message	Parity
0	0	0	000	0	000	1
0	0	1	001	1	001	0
0	1	0	010	1	010	0
0	1	1	011	0	011	1
1	0	0	100	1	100	0
1	0	1	101	0	101	1
1	1	0	110	0	110	1
1	1	1	111	1	111	0

Cyclic Redundancy Check (CRC)

A cyclic code is a linear (n, k) block code with the property that every cyclic shift of a codeword results in another code word. Here k indicates the length of the message at transmitter (the number of information bits). n is the total length of the message after adding check bits. (actual data and the check bits). n, k is the number of check bits. The codes used for cyclic redundancy check there by error detection are known as CRC codes (Cyclic redundancy check codes). Cyclic redundancy-check codes are shortened cyclic codes. These types of codes are used for error detection and encoding. They are easily implemented using shift-registers with feedback connections. That is why they are widely used for error detection on digital communication. CRC codes will provide effective and high level of protection.

CRC Code Generation

Based on the desired number of bit checks, we will add some zeros (0) to the actual data. This new binary data sequence is divided by a new word of length $n + 1$, where n is the number of check bits to be added . The remainder obtained as a result of this modulo 2- division is added to the dividend bit sequence to form the cyclic code. The generated code word is completely divisible by the divisor that is used in generation of code. This is transmitted through the transmitter.



At the receiver side, we divide the received code word with the same divisor to get the actual code word. For an error free reception of data, the remainder is 0. If the remainder is a non-zero, that means there is an error in the received code / data sequence. The probability of error detection depends upon the number of check bits (n) used to construct the cyclic code. For single bit and two bit errors, the probability is 100 % .

For a burst error of length $n - 1$, the probability of error detecting is 100 % .

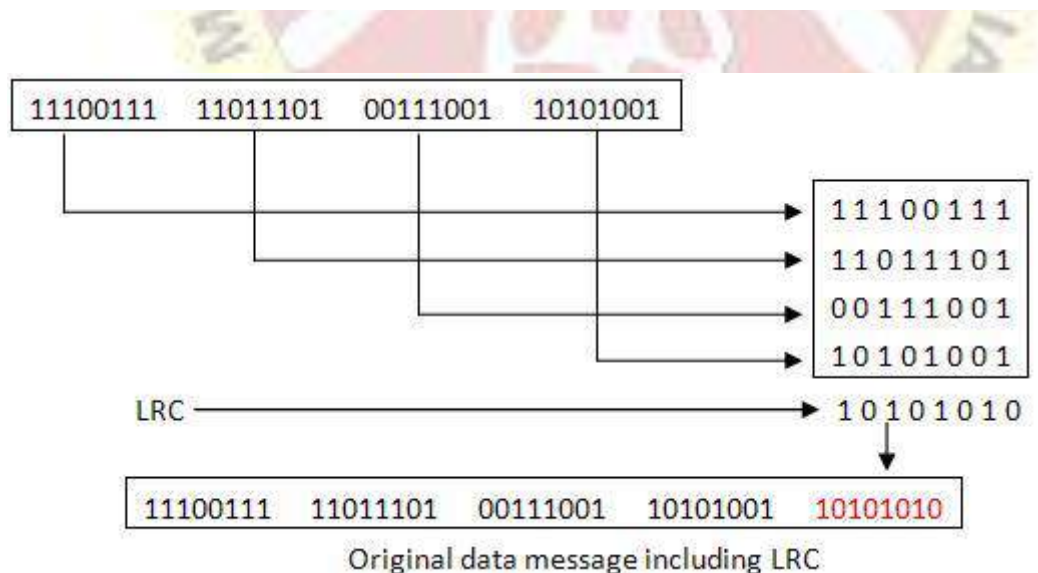
A burst error of length equal to $n + 1$, the probability of error detecting reduces to $1 - (1/2)^{n-1}$. A burst error of length greater than $n - 1$, the probability of error detecting is $1 - (1/2)^n$.

Longitudinal Redundancy Check

In longitudinal redundancy method, a BLOCK of bits are arranged in a table format (in rows and columns) and we will calculate the parity bit for each column separately. The set of these parity bits are also sent along with our original data bits.

Longitudinal redundancy check is a bit by bit parity computation, as we calculate the parity of each column individually.

This method can easily detect burst errors and single bit errors and it fails to detect the 2 bit errors occurred in same vertical slice.



Check Sum

Checksums are similar to parity bits except, the number of bits in the sums is larger than parity and the result is always constrained to be zero. That means if the checksum is zero,

error is detected. A checksum of a message is an arithmetic sum of code words of certain length. The sum is stated by means of 1's complement and stored or transferred as a code extension of actual code word. At receiver a new checksum is calculated by receiving the bit sequence from transmitter.

The checksum method includes parity bits, check digits and longitudinal redundancy check (LRC). For example, if we have to transfer and detect errors for a long data sequence (also called as Data string) then we divide that into shorter words and we can store the data with a word of same width. For each another incoming bit we will add them to the already stored data. At every instance, the newly added word is called "Checksum".

At receiver side, the received bits checksum is same as that of transmitter's, there is no error found.

Checksums are similar to parity bits except, the number of bits in the sums is larger than parity and the result is always constrained to be zero. That means if the checksum is zero, error is detected. A checksum of a message is an arithmetic sum of code words of certain length. The sum is stated by means of 1's complement and stored or transferred as a code extension of actual code word. At receiver a new checksum is calculated by receiving the bit sequence from transmitter.

The checksum method includes parity bits, check digits and longitudinal redundancy check (LRC). For example, if we have to transfer and detect errors for a long data sequence (also called as Data string) then we divide that into shorter words and we can store the data with a word of same width. For each another incoming bit we will add them to the already stored data. At every instance, the newly added word is called "Checksum".

At receiver side, the received bits checksum is same as that of transmitter's, there is no error found.

We can also find the checksum by adding all data bits. For example, if we have 4 bytes of data as 25h, 62h, 3fh, 52h.

Then, adding all bytes we get 118H

Dropping the carry Nibble, we get 18H

Find the 2's complement of the nibble, i.e. E8H

This is the checksum of the transmitted 4 bits of data.

At receiver side, to check whether the data is received without error or not, just add the checksum to the actual data bits (we will get 200H). By dropping the carry nibble we get 00H. This means the checksum is constrained to zero. So there is no error in the data.

Error correction code

Error correction codes are generated by using the specific algorithm used for removing and detecting errors from the message transmitted over the noisy channels. The error-correcting codes find the correct number of corrupted bits and their positions in the message. There are two types of ECCs (Error Correction Codes), which are as follows.

Block codes

In block codes, in fixed-size blocks of bits, the message is contained. In this, the redundant bits are added for correcting and detecting errors.

Convolutional codes

The message consists of data streams of random length, and parity symbols are generated by the sliding application of the Boolean function to the data stream.

The hamming code technique is used for error correction.

Hamming Code

Hamming code is an example of a block code. The two simultaneous bit errors are detected, and single-bit errors are corrected by this code. In the hamming coding mechanism, the sender encodes the message by adding the unessential bits in the data. These bits are added to the specific position in the message because they are the extra bits for correction.

ASCII Code

The ASCII stands for American Standard Code for Information Interchange. The ASCII code is an alphanumeric code used for data communication in digital computers. The ASCII is a 7-bit code capable of representing 2^7 or 128 number of different characters. The ASCII code is made up of a three-bit group, which is followed by a four-bit code.

The ASCII Code is a 7 or 8-bit alphanumeric code.

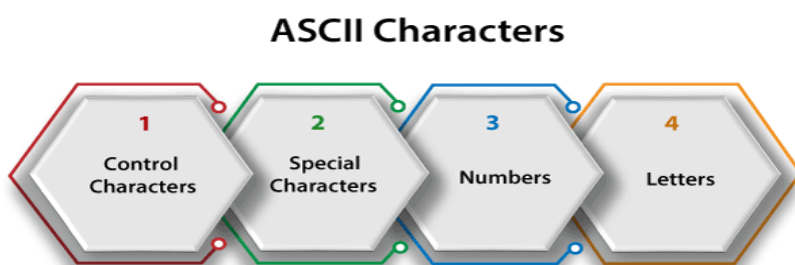
This code can represent 127 unique characters.

The ASCII code starts from 00h to 7Fh. In this, the code from 00h to 1Fh is used for control characters, and the code from 20h to 7Fh is used for graphic symbols.

The 8-bit code holds ASCII, which supports 256 symbols where math and graphic symbols are added.

The range of the extended ASCII is 80h to FFh.

The ASCII characters are classified into the following groups:



Symbol	Binary value	Symbol	Binary value	Symbol	Binary value
(Space)	0100000	@	1000000	`	1100000
!	0100001	A	1000001	a	1100001
"	0100010	B	1000010	b	1100010
#	0100011	C	1000011	c	1100011
\$	0100100	D	1000100	d	1100100
%	0100101	E	1000101	e	1100101
&	0100110	F	1000110	f	1100110
'	0100111	G	1000111	g	1100111
(0101000	H	1001000	h	1101000
)	0101001	I	1001001	i	1101001
*	0101010	J	1001010	j	1101010
+	0101011	K	1001011	k	1101011
,	0101100	L	1001100	l	1101100
-	0101101	M	1001101	m	1101101
.	0101110	N	1001110	n	1101110
/	0101111	O	1001111	o	1101111
0	0110000	P	1010000	p	1110000
1	0110001	Q	1010001	q	1110001
2	0110010	R	1010010	r	1110010
3	0110011	S	1010011	s	1110011
4	0110100	T	1010100	t	1110100
5	0110101	U	1010101	u	1110101
6	0110110	V	1010110	v	1110110
7	0110111	W	1010111	w	1110111
8	0111000	X	1011000	x	1111000
9	0111001	Y	1011001	y	1111001
:	0111010	Z	1011010	z	1111010
;	0111011	[1011011	{	1111011
<	0111100	\	1011100		1111100
=	0111101]	1011101	}	1111101
>	0111110	^	1011110	~	1111110
?	0111111	_	1011111	(Delete)	1111111

EBCDI CODE

EBCDI stands for Extended Binary Coded Decimal Interchange code. This code is developed by IBM Inc Company. It is an 8 bit code, so we can represent $2^8 = 256$ characters by using EBCDI code. This include all the letters and symbols like 26 lower case letters (a – z), 26 upper case letters (A – Z), 33 special characters and symbols (like ! @ # \$ etc), 33 control characters (* – + / and % etc) and 10 digits (0 – 9).

In the EBCDI code, the 8 bit code the numbers are represented by 8421 BCD code preceded by 1111.

Special characters	EBCDIC	Alphabetic	EBCDIC
<	01001011	A	11000001
(01001100	B	11000010
+	01001101	C	11000011
/	01001110	D	11000100
&	01010000	E	11000101
:	01111011	F	11000110
#	01111011	G	11000111
@	01111100	H	11001000
'	01111101	I	11001001
=	01111110	J	11010001
"	01111111	K	11010010
,	01101011	L	11010011
%	01101100	M	11010100
-	01101101	N	11010101
>	01101110	O	11010110
		P	11010111

UNIT II

BOOLEAN ALGEBRA AND THEOREMS

Logic Gates

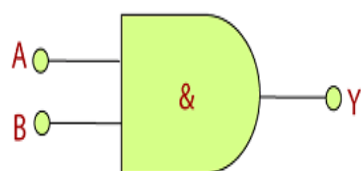
Logic gates play an important role in circuit design and digital systems. It is a building block of a digital system and an electronic circuit that always has only one output. These gates can have one input or more than one input, but most of the gates have two inputs. On the basis of

the relationship between the input and the output, these gates are named as AND gate, OR gate, NOT gate, etc.

AND Gate

The AND gate is a circuit that performs the AND operation of the inputs. This gate has a minimum of 2 input values and an output value.

SYMBOL

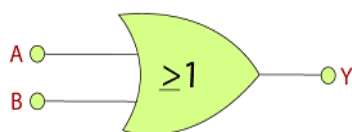


2- Input AND Gate

Inputs		Output
A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate

The OR gate is a circuit which performs the OR operation of the inputs. This gate also has a minimum of 2 input values and an output value.

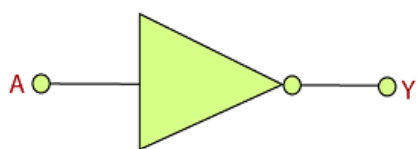


2- Input OR Gate

Inputs		Output
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate

The NOT gate is also called an inverter. This gate gives the inverse value of the input value as a result. This gate has only one input and one output value.

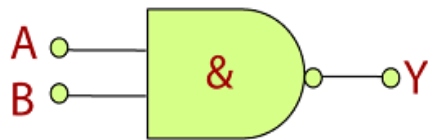


NOT Gate

Input	Output
A	B
0	1
1	0

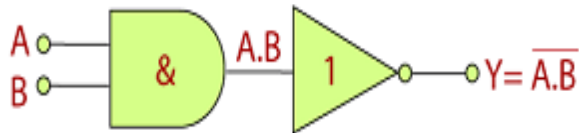
NAND Gate

The NAND gate is the combination of AND gate and NOT gate. This gate gives the same result as a NOT-AND operation. This gate can have two or more than two input values and only one output value.



2- Input NAND Gate

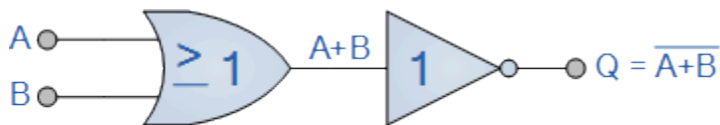
Inputs		Output
A	B	$(AB)'$
0	0	1
0	1	1
1	0	1
1	1	0



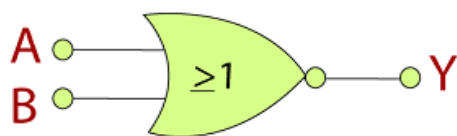
2- Input "AND" gate plus a "NOT" gate

NOR Gate

The NOR gate is the combination of an OR gate and NOT gate. This gate gives the same result as the NOT-OR operation. This gate can have two or more than two input values and only one output value.



2-input "OR" gate plus a "NOT" gate

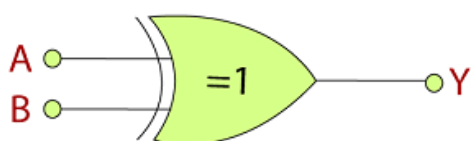


2- Input NOR Gate

Inputs		Output
A	B	$(AB)'$
0	0	1
0	1	0
1	0	0
1	1	0

XOR Gate

The XOR gate is also known as the Ex-OR gate. The XOR gate is used in half and full adder and subtractor. The exclusive-OR gate is sometimes called as EX-OR and X-OR gate. This gate can have two or more than two input values and only one output value.

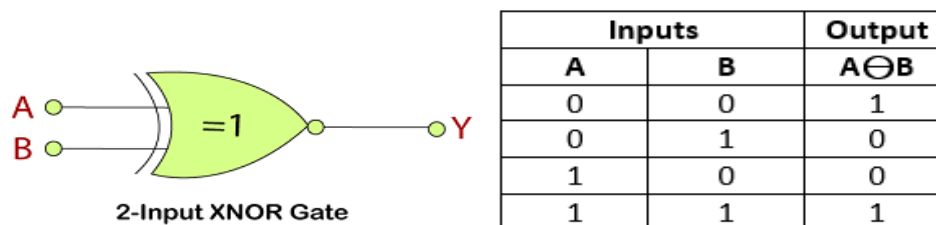


2-Input XOR Gate

Inputs		Output
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

XNOR Gate

The XNOR gate is also known as the Ex-NOR gate. The XNOR gate is used in half and full adder and subtractor. The exclusive-NOR gate is sometimes called as EX-NOR and X-NOR gate. This gate can have two or more than two input values and only one output value.



Boolean Algebra

The logical symbol 0 and 1 are used for representing the digital input or output. The symbols "1" and "0" can also be used for a permanently open and closed digital circuit. The digital circuit can be made up of several logic gates. To perform the logical operation with minimum logic gates, a set of rules were invented, known as the **Laws of Boolean Algebra**. These rules are used to reduce the number of logic gates for performing logic operations.

The Boolean algebra is mainly used for simplifying and analyzing the complex Boolean expression. It is also known as **Binary algebra** because we only use binary numbers in this.

Rules in Boolean algebra

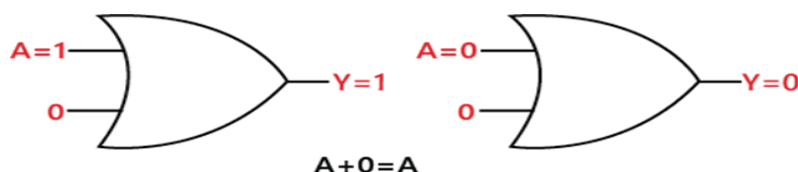
1. Only two values(1 for high and 0 for low) are possible for the variable used in Boolean algebra.
2. The overbar(-) is used for representing the complement variable. So, the complement of variable C is represented as \bar{C} .
3. The plus(+) operator is used to represent the ORing of the variables.
4. The dot(.) operator is used to represent the ANDing of the variables.

Rules of Boolean algebra

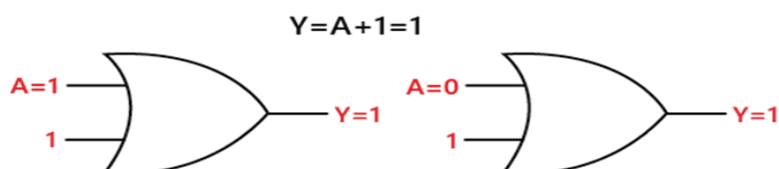
1.	$A+0=A$	7.	$A.A=A$
2.	$A+1=1$	8.	$A.A'=0$
3.	$A.0=0$	9.	$A''=A$
4.	$A.1=A$	10.	$A+AB=A$
5.	$A+A=A$	11.	$A+A'B=A+B$
6.	$A+A'=1$	12.	$(A+B)(A+C)=A+BC$

Rule 1: $A + 0 = A$

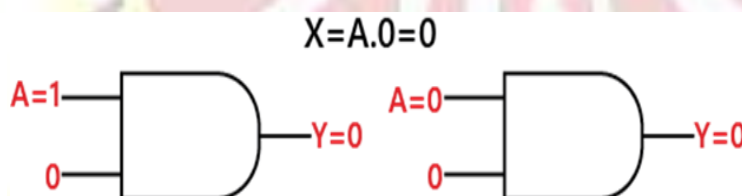
We have an input variable A whose value is either 0 or 1. When we perform OR operation with 0, the result will be the same as the input variable. So, if the variable value is 1, then the result will be 1, and if the variable value is 0, then the result will be 0.

**Rule 2: $(A + 1) = 1$**

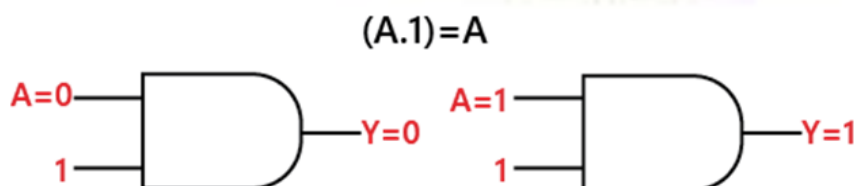
We have an input variable A whose value is either 0 or 1. When we perform OR operation with 1, the result will always be 1. So, if the variable value is either 1 or 0, then the result will always be 1.

**Rule 3: $(A \cdot 0) = 0$**

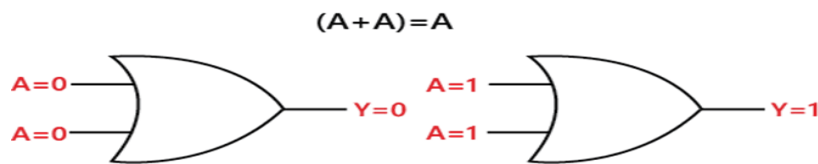
We have an input variable A whose value is either 0 or 1. When we perform the AND operation with 0, the result will always be 0. This rule states that an input variable ANDed with 0 is equal to 0 always.

**Rule 4: $(A \cdot 1) = A$**

We have an input variable A whose value is either 0 or 1. When we perform the AND operation with 1, the result will always be equal to the input variable. This rule states that an input variable ANDed with 1 is equal to the input variable always.

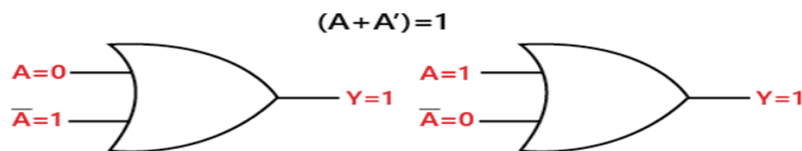
**Rule 5: $(A + A) = A$**

We have an input variable A whose value is either 0 or 1. When we perform the OR operation with the same variable, the result will always be equal to the input variable. This rule states an input variable ORed with itself is equal to the input variable always.



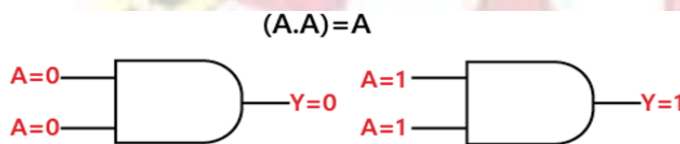
Rule 6: $(A + A') = 1$

We have an input variable A whose value is either 0 or 1. When we perform the OR operation with the complement of that variable, the result will always be equal to 1. This rule states that a variable ORed with its complement is equal to 1 always.



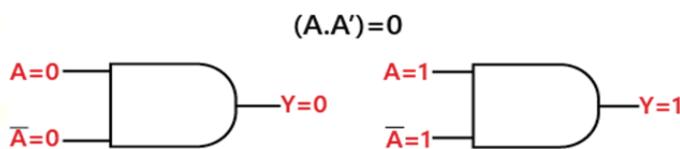
Rule 7: $(A.A) = A$

We have an input variable A whose value is either 0 or 1. When we perform the AND operation with the same variable, the result will always be equal to that variable only. This rule states that a variable ANDed with itself is equal to the input variable always.



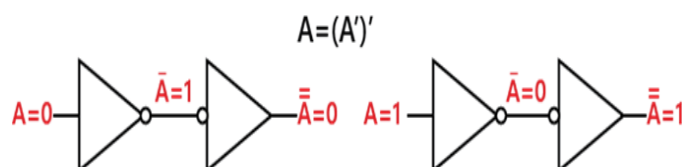
Rule 8: $(A.A') = 0$

We have an input variable A whose value is either 0 or 1. When we perform the AND operation with the complement of that variable, the result will always be equal to 0. This rule states that a variable ANDed with its complement is equal to 0 always.



Rule 9: $A = (A')'$

This rule states that if we perform the double complement of the variable, the result will be the same as the original variable. So, when we perform the complement of variable A, then the result will be A'. Further if we again perform the complement of A', we will get A.



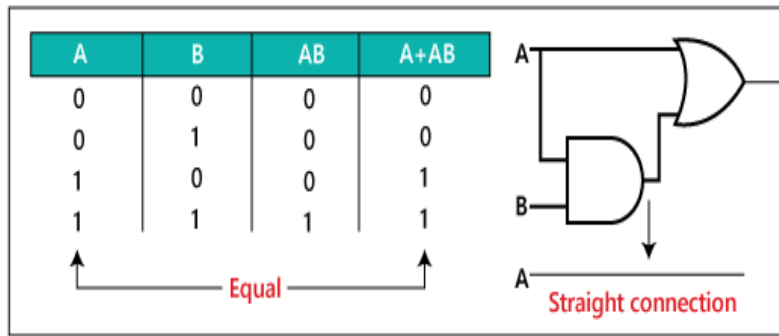
Rule 10: $(A + AB) = A$

We can prove this rule by using the rule 2, rule 4, and the distributive law as:

$$A + AB = A(1 + B) \quad \text{Factoring (distributive law)}$$

$$A + AB = A.1 \quad \text{Rule 2: } (1 + B) = 1$$

$$A + AB = A \quad \text{Rule 4: } A.1 = A$$



Rule 11: $A + AB = A + B$

$A + AB = (A + AB) + AB$

$A + AB = (AA + AB) + AB$

$A + AB = AA + AB + AA + AB$

$A + AB = (A + A)(A + B)$

$A + AB = 1.(A + B)$

$A + AB = A + B$

Rule 10: $A = A + AB$

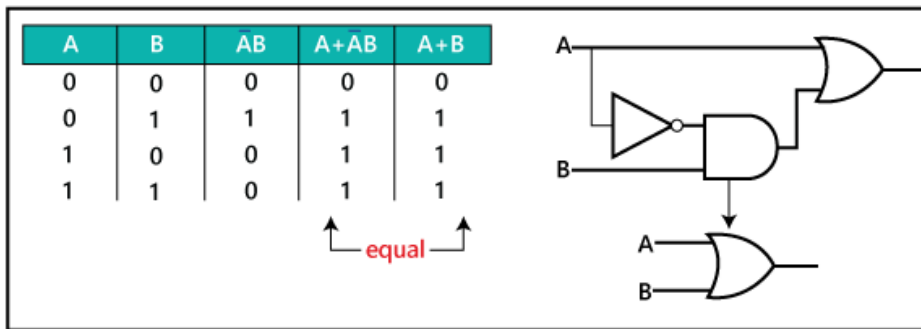
Rule 7: $A = AA$

Rule 8: adding $AA = 0$

Factoring

Rule 6: $A + A = 1$

Rule 4: drop the



Rule 12: $(A + B)(A + C) = A + BC$

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

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

$(A + B)(A + C) = A(1 + C) + AB + BC$

$(A + B)(A + C) = A.1 + AB + BC$

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

$(A + B)(A + C) = A.1 + BC$

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

Distributive law

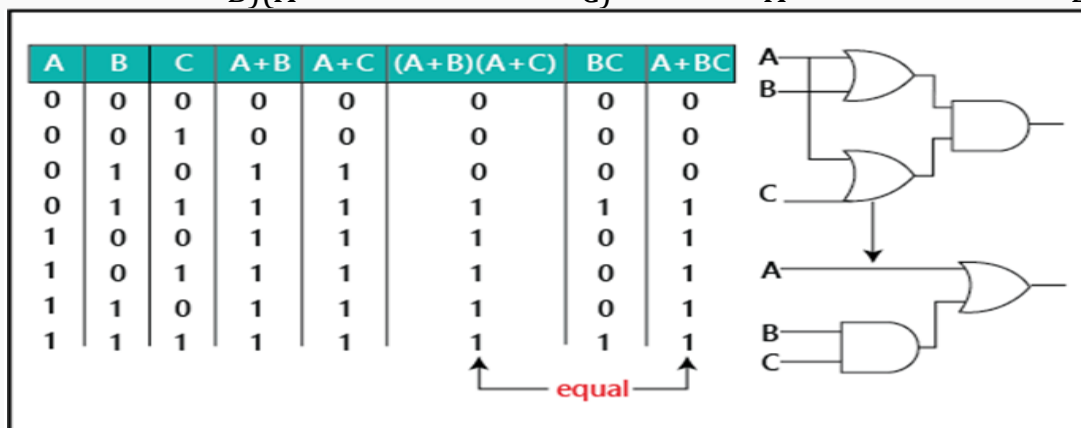
Rule 7: $AA = A$

Rule 2: $1 + C = 1$

Factoring (distributive law)

Rule 2: $1 + B = 1$

Rule 4: $A .1 = A$



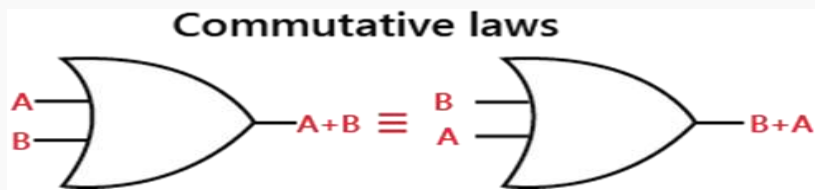
Laws of Boolean algebra

Commutative Law

This law states that no matter in which order we use the variables. It means that the order of variables doesn't matter. In Boolean algebra, the OR and the addition operations are similar.

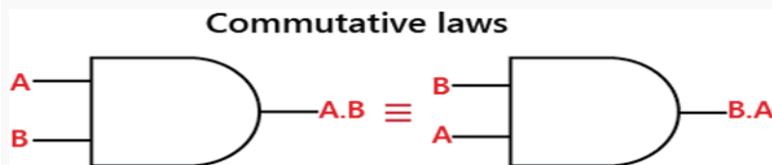
For two variables, the commutative law of addition is written as:

$$A+B = B+A$$



For two variables, the commutative law of multiplication is written as:

$$A.B = B.A$$

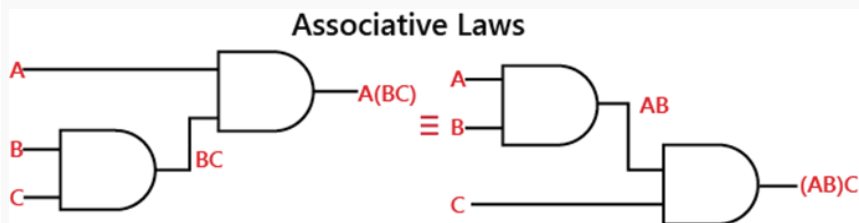


Associative Law

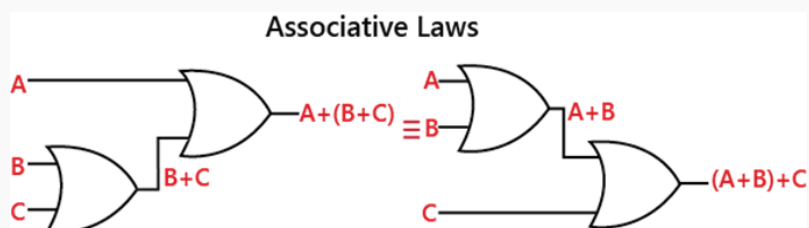
This law states that the operation can be performed in any order when the variables priority is same. As '*' and '/' have same priority. In the below diagram, the associative law is applied to the 2-input OR gate.

For three variables, the associative law of addition is written as:

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



$$A(BC) = (AB)C$$

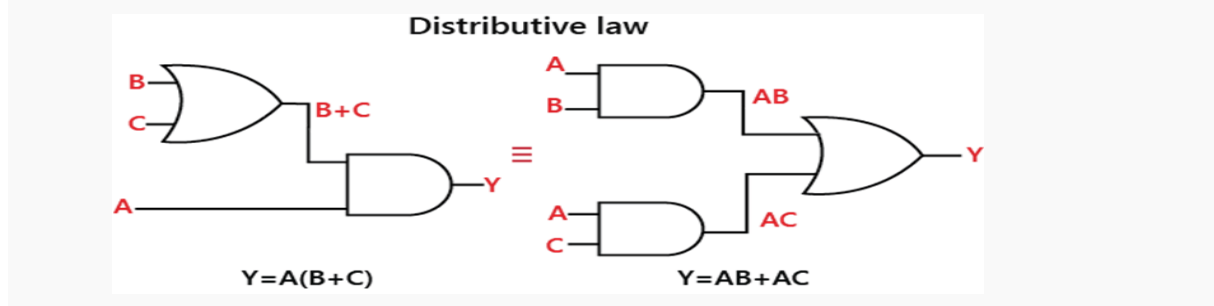


Distributive Law:

According to this law, if we perform the OR operation of two or more variables and then perform the AND operation of the result with a single variable, then the result will be similar to performing the AND operation of that single variable with each two or more variable and then perform the OR operation of that product. This law explains the process of factoring.

For three variables, the distributive law is written as:

$$A(B + C) = AB + AC$$

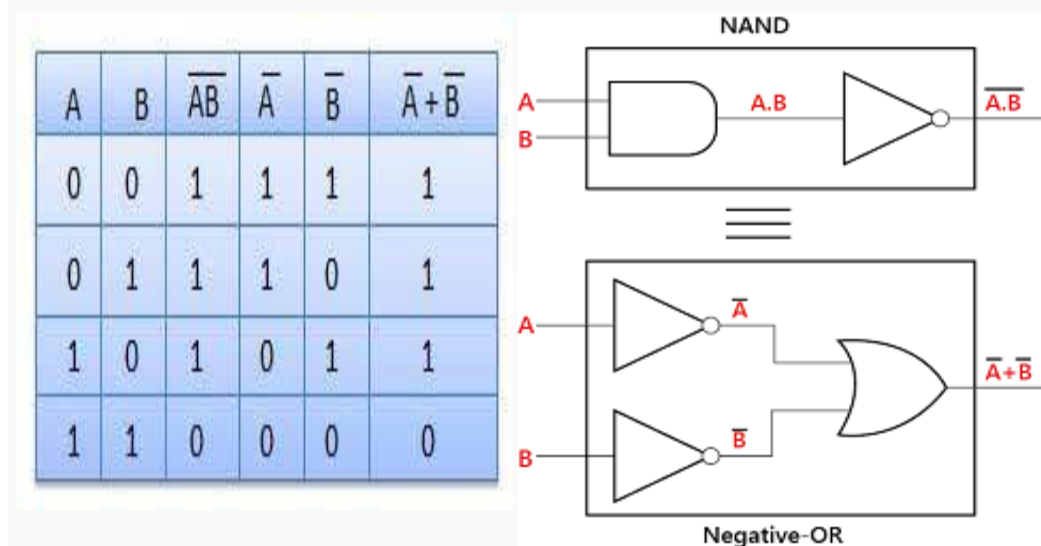


De-Morgan's Theorem

A famous mathematician **DeMorgan** invented the two most important theorems of boolean algebra. The DeMorgan's theorems are used for mathematical verification of the equivalency of the NOR and negative-AND gates and the negative-OR and NAND gates. These theorems play an important role in solving various boolean algebra expressions.

Theorem 1

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



Theorem 2

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

SUM OF PRODUCT AND PRODUCT OF SUM

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

Minterm

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

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.

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 $\sim A$ 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

Maxterm

The sum of all literals, either with complement or without complement, is known as **maxterm**.

Maxterm from values

Using the given variable values, we can write the maxterm as:

1. If the variable value is 1, then we will take the variable without a complement.
2. If the variable value is 0, take the complement of the variable.

The steps for the maxterm are same as 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 $\sim A$ 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 Here, **M** denotes maxterm.

X	Y	Z	Minterms Product Terms	Maxterms Sum Terms
0	0	0	$m_0 = \bar{X} \cdot \bar{Y} \cdot \bar{Z} = \min\{\bar{X}, \bar{Y}, \bar{Z}\}$	$M_0 = X + Y + Z = \max\{X, Y, Z\}$
0	0	1	$m_1 = \bar{X} \cdot \bar{Y} \cdot Z = \min\{\bar{X}, \bar{Y}, Z\}$	$M_1 = X + Y + \bar{Z} = \max\{X, Y, \bar{Z}\}$
0	1	0	$m_2 = \bar{X} \cdot Y \cdot \bar{Z} = \min\{\bar{X}, Y, \bar{Z}\}$	$M_2 = X + \bar{Y} + Z = \max\{X, \bar{Y}, Z\}$
0	1	1	$m_3 = \bar{X} \cdot Y \cdot Z = \min\{\bar{X}, Y, Z\}$	$M_3 = X + \bar{Y} + \bar{Z} = \max\{X, \bar{Y}, \bar{Z}\}$
1	0	0	$m_4 = X \cdot \bar{Y} \cdot \bar{Z} = \min\{X, \bar{Y}, \bar{Z}\}$	$M_4 = \bar{X} + Y + Z = \max\{\bar{X}, Y, Z\}$
1	0	1	$m_5 = X \cdot \bar{Y} \cdot Z = \min\{X, \bar{Y}, Z\}$	$M_5 = \bar{X} + Y + \bar{Z} = \max\{\bar{X}, Y, \bar{Z}\}$
1	1	0	$m_6 = X \cdot Y \cdot \bar{Z} = \min\{X, Y, \bar{Z}\}$	$M_6 = \bar{X} + \bar{Y} + Z = \max\{\bar{X}, \bar{Y}, Z\}$
1	1	1	$m_7 = X \cdot Y \cdot Z = \min\{X, Y, Z\}$	$M_7 = \bar{X} + \bar{Y} + \bar{Z} = \max\{\bar{X}, \bar{Y}, \bar{Z}\}$

Simplification of Boolean Expression

$$AB + A(B+C) + B(B+C)$$

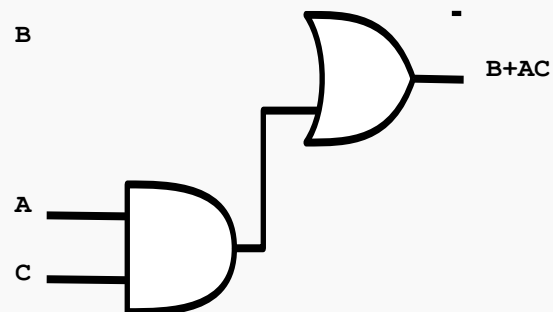
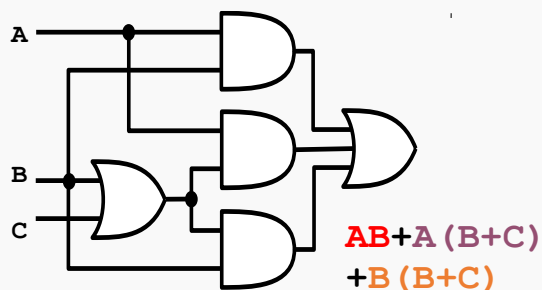
$$AB + AB + AC + BB + BC$$

$$AB + AB + AC + B + BC$$

$$AB + AC + B + BC$$

$$AB + AC + B$$

$$B + AC$$



2.

$$\begin{aligned}
 (A + B)(A + C) &= AA + AC + AB + BC \\
 &= A + AC + AB + BC \\
 &= A(1 + C + B) + BC \\
 &= A \cdot 1 + BC \\
 &= A + BC
 \end{aligned}$$

3.

$$\begin{aligned}
 (\bar{A} + B)(A + B) &= B(\bar{A} + A) \\
 &= B(1) \\
 &= B
 \end{aligned}$$

4.

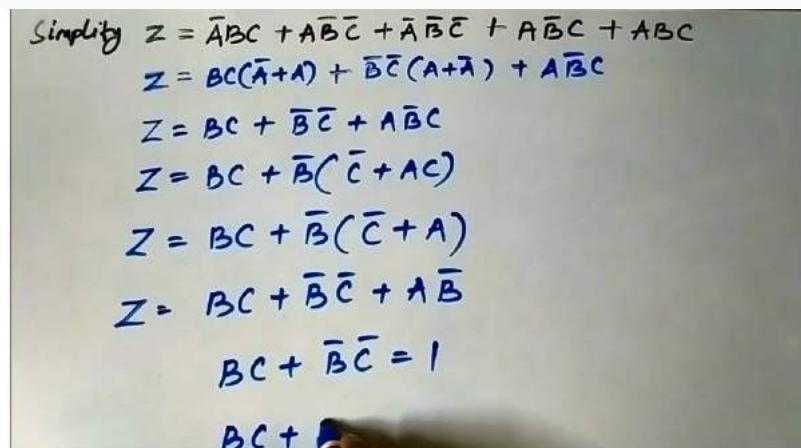
$$\begin{aligned} A\bar{C} + ABC\bar{C} &= A\bar{C} (1 + B) \\ &= A\bar{C} (1) \\ &= A\bar{C} \end{aligned}$$

5.

Simplification: Example

$$\begin{aligned} \overline{(\bar{A}B + \bar{A}\bar{B})}(A+B) &= \overline{\bar{A}B\bar{A}\bar{B}}(A+B) \\ &= (\bar{A}+B)(A+\bar{B})(A+B) \\ &= (\bar{A}+B)(AA+AB+\bar{B}A+\bar{B}B) \\ &= (\bar{A}+B)(A+AB+\bar{B}A+\bar{B}B) \\ &= (\bar{A}+B)(A(1+B+\bar{B})+\bar{B}B) \\ &= (\bar{A}+B)(A(1)+\bar{B}B) \\ &= (\bar{A}+B)A \\ &= A\bar{A} + AB \\ &= AB \end{aligned}$$

6.



Simplify $Z = \bar{A}BC + A\bar{B}C + \bar{A}\bar{B}C + \bar{A}\bar{B}\bar{C} + ABC$
 $Z = BC(\bar{A}+A) + \bar{B}\bar{C}(A+\bar{A}) + A\bar{B}C$
 $Z = BC + \bar{B}\bar{C} + A\bar{B}C$
 $Z = BC + \bar{B}(\bar{C} + AC)$
 $Z = BC + \bar{B}(\bar{C} + A)$
 $Z = BC + \bar{B}\bar{C} + A\bar{B}$
 $BC + \bar{B}\bar{C} = 1$
 $BC + A\bar{B}$



Simplification of Boolean Expression using Boolean Algebra Rules 2

Digital
Electronics

Karnaugh Map(K-Map) method

The **K-map** is a systematic way of simplifying Boolean expressions. With the help of the K-map method, we can find the simplest POS and SOP expression, which is

known as the minimum expression. The K-map provides a cookbook for simplification.

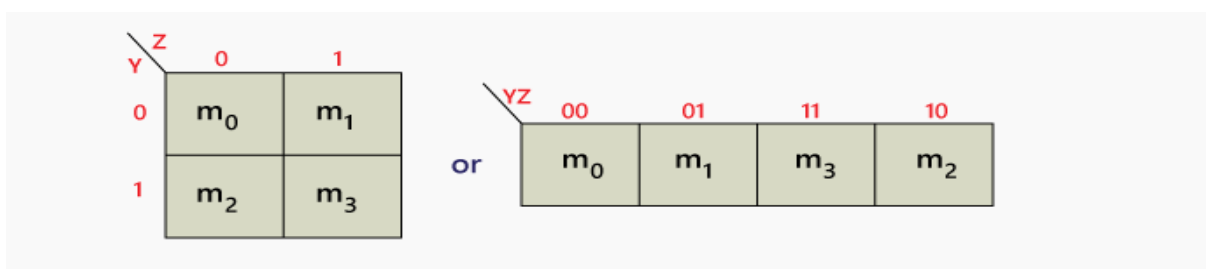
A K-map contains all the possible values of input variables and their corresponding output values. The values are stored in cells of the array. In each cell, a binary value of each input variable is stored.

The K-map method is used for expressions containing 2, 3, 4, and 5 variables. In K-map, the number of cells is similar to the total number of variable input combinations. For example, if the number of variables is three, the number of cells is $2^3=8$, and if the number of variables is four, the number of cells is 2^4 . The K-map takes the SOP and POS forms. The K-map grid is filled using 0's and 1's. The K-map is solved by making groups. There are the following steps used to solve the expressions using K-map:

1. First, we find the K-map as per the number of variables.
2. Find the maxterm and minterm in the given expression.
3. Fill cells of K-map for SOP with 1 respective to the minterms.
4. Fill cells of the block for POS with 0 respective to the maxterm.
5. Next, we create rectangular groups that contain total terms in the power of two like 2, 4, 8, ... and try to cover as many elements as we can in one group.
6. With the help of these groups, we find the product terms and sum them up for the SOP form.

2 Variable K-map

There is a total of 4 variables in a 2-variable K-map. There are two variables in the 2-variable K-map. The following figure shows the structure of the 2-variable K-map:



The possible combinations of grouping 2 adjacent minterms are $\{(m_0, m_1), (m_2, m_3), (m_0, m_2)$ and $(m_1, m_3)\}$.

3-variable K-map

The 3-variable K-map is represented as an array of eight cells. In this case, we used A, B, and C for the variable. We can use any letter for the names of the variables. The binary values of variables A and B are along the left side, and the values of C are across the top. The value of the given cell is the binary values of A and B at left side in the same row combined with the value of C at the top in the same column.

		C	
		0	1
AB	00		
	01		
	11		
	10		

		C	
		0	1
AB	00	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$
	01	$\bar{A}\bar{B}\bar{C}$	$\bar{A}\bar{B}C$
	11	$A\bar{B}\bar{C}$	ABC
	10	$A\bar{B}\bar{C}$	$A\bar{B}C$

4-Variable Karnaugh Map

The 4-variable K-map is represented as an array of 16 cells. Binary values of A and B are along the left side, and the values of C and D are across the top. The value of the given cell is the binary values of A and B at left side in the same row combined with the binary values of C and D at the top in the same column.

		CD			
		00	01	11	10
AB	00				
	01				
	11				
	10				

		CD			
		00	01	11	10
AB	00	$\bar{A}\bar{B}\bar{C}\bar{D}$	$\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}C\bar{D}$	$\bar{A}\bar{B}CD$
	01	$\bar{A}\bar{B}\bar{C}\bar{D}$	$\bar{A}\bar{B}\bar{C}D$	$\bar{A}\bar{B}C\bar{D}$	$\bar{A}\bar{B}CD$
	11	$A\bar{B}\bar{C}\bar{D}$	$A\bar{B}\bar{C}D$	$ABC\bar{D}$	$ABCD$
	10	$A\bar{B}\bar{C}\bar{D}$	$A\bar{B}\bar{C}D$	$A\bar{B}C\bar{D}$	$A\bar{B}CD$

Simplification of boolean expressions using Karnaugh Map

As we know that K-map takes both SOP and POS forms. So, there are two possible solutions for K-map, i.e., minterm and maxterm solution. Let's start and learn about how we can find the minterm and maxterm solution of K-map.

Minterm Solution of K Map

There are the following steps to find the minterm solution or K-map:

Step 1:

Firstly, we define the given expression in its canonical form.

Step 2:

Next, we create the K-map by entering 1 to each product-term into the K-map cell and fill the remaining cells with zeros.

Step 3:

Next, we form the groups by considering each one in the K-map.

Example 1: $Y = A'B' + A'B + AB$

		B	
		0	1
A	0	1	1
	1	1	1

0 1 2 3

Simplified expression: $Y=A+B$

Example 2: $Y=A'B'C'+A'BC'+AB'C'+AB'C+ABC'+ABC$

		BC			
		00	01	11	10
A	0	0	0	1	1
	1	1	1	1	1

0 1 3 2 4 5 7 6

Simplified expression: $Y=A+C'$

Example 3: $Y=A'B'C'D'+A'B'CD'+A'BCD'+A'BCD+AB'C'D'+ABCD'+ABCD$

		CD			
		00	01	11	10
AB	00	1	0	0	1
	01	0	1	1	0
	11	0	1	1	0
	10	1	0	0	0

0 1 3 2 4 5 7 6 12 13 15 14 8 9 11 10

Simplified expression: $Y=BD+B'D'$

UNIT III

COMBINATIONAL DIGITAL CIRCUITS

The combinational logic circuits are the circuits that contain different types of logic gates. Simply, a circuit in which different types of logic gates are combined is known as a **combinational logic circuit**. The output of the combinational circuit is determined from the present combination of inputs, regardless of the previous input. The input variables, logic gates, and output variables are the basic components of the combinational logic circuit. There are different types of combinational logic circuits, such as Adder, Subtractor, Decoder, Encoder, Multiplexer, and De-multiplexer.

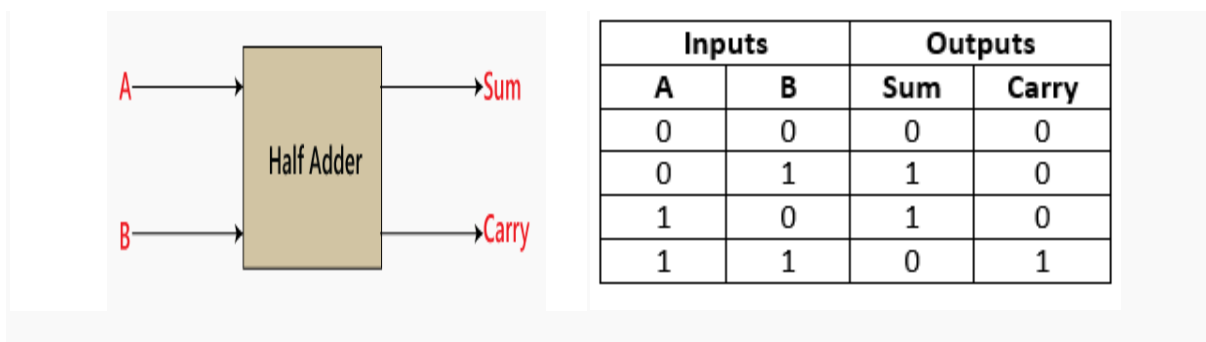
There are the following characteristics of the combinational logic circuit:

- At any instant of time, the output of the combinational circuits depends only on the present input terminals.
- The combinational circuit doesn't have any backup or previous memory. The present state of the circuit is not affected by the previous state of the input.
- The n number of inputs and m number of outputs are possible in combinational logic circuits.

Half Adder

The Half-Adder is a basic building block of adding two numbers as two inputs and produce out two outputs. The adder is used to perform OR operation of two single bit binary numbers. The **augend** and **addend** bits are two input states, and '**carry**' and '**sum**' are two output states of the half adder.

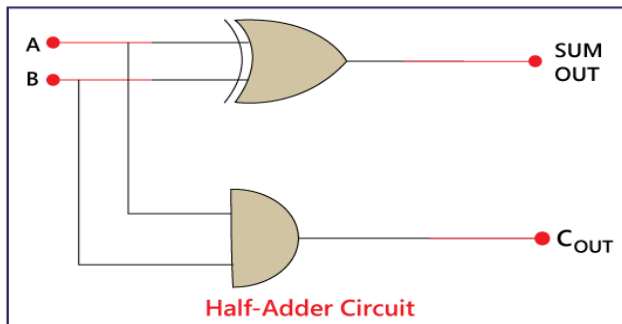
Block diagram



In the above table,

1. 'A' and 'B' are the input states, and 'sum' and 'carry' are the output states.
2. The carry output is 0 in case where both the inputs are not 1.
3. The least significant bit of the sum is defined by the 'sum' bit.

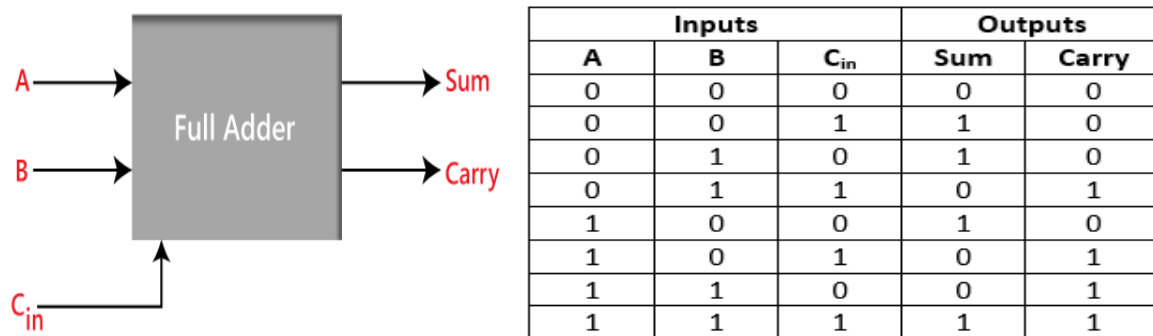
4. Sum = $x'y + xy'$
 Carry = xy



Full Adder

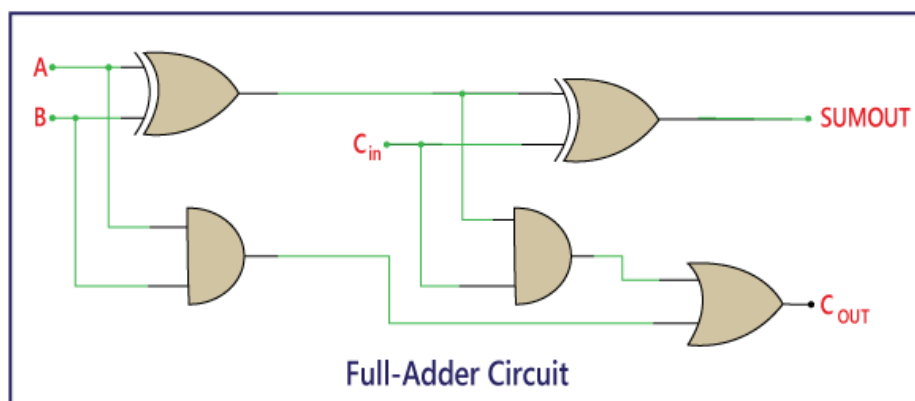
The half adder is used to add only two numbers. To overcome this problem, the full adder was developed. The full adder is used to add three 1-bit binary numbers A, B, and carry C. The full adder has three input states and two output states i.e., sum and carry.

Block diagram



In the above table,

1. 'A' and 'B' are the input variables. These variables represent the two significant bits which are going to be added
2. 'C_{in}' is the third input which represents the carry. From the previous lower significant position, the carry bit is fetched.
3. The 'Sum' and 'Carry' are the output variables that define the output values.
4. The eight rows under the input variable designate all possible combinations of 0 and 1 that can occur in these variables.



There are two half adder circuits that are combined using the OR gate. The first half adder has two single-bit binary inputs A and B. As we know that, the half adder produces two outputs, i.e., Sum and Carry. The 'Sum' output of the first adder will be the first input of the second half adder, and the 'Carry' output of the first adder will be the second input of the second half adder. The second half adder will again provide 'Sum' and 'Carry'. The final outcome of the Full adder circuit is the 'Sum' bit. In order to find the final output of the 'Carry', we provide the 'Carry' output of the first and the second adder into the OR gate. The outcome of the OR gate will be the final carry out of the full adder circuit.

Half Subtractor

The half subtractor is also a building block for subtracting two binary numbers. It has two inputs and two outputs. This circuit is used to subtract two single bit binary numbers A and B. The '**diff**' and '**borrow**' are two output states of the half subtractor.

Block diagram



The SOP form of the **Diff** and **Borrow** is as follows:

Diff=

$A'B + AB'$

Borrow = $A'B$

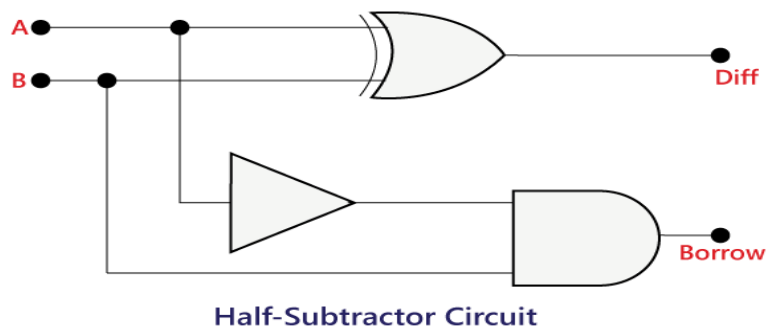
In the above table,

- 'A' and 'B' are the input variables whose values are going to be subtracted.
- The 'Diff' and 'Borrow' are the variables whose values define the subtraction result, i.e., difference and borrow.

- The first two rows and the last row, the difference is 1, but the 'Borrow' variable is 0.
- The third row is different from the remaining one. When we subtract the bit 1 from the bit 0, the borrow bit is produced.

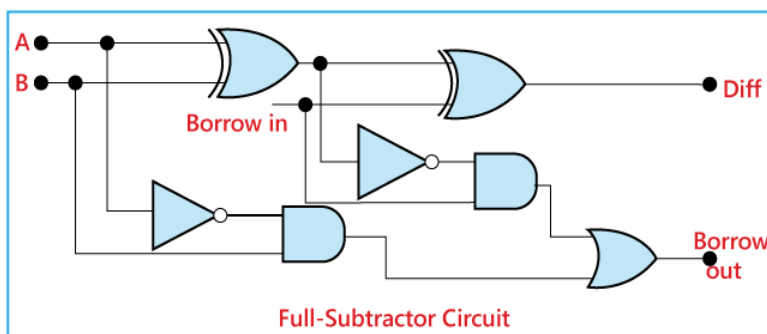
Half-Subtractor logical circuit

So, the Half Subtractor is designed by combining the 'XOR', 'AND', and 'NOT' gates and provide the Diff and Borrow.



Full Subtractor

To overcome this problem, a full subtractor was designed. The full subtractor is used to subtract three 1-bit numbers A, B, and C, which are minuend, subtrahend, and borrow, respectively. The full subtractor has three input states and two output states i.e., diff and borrow.



The full subtractor logic circuit can be constructed using the 'AND', 'XOR', and NOT gate with an OR gate. There are two half subtractor circuits that are combined using the OR

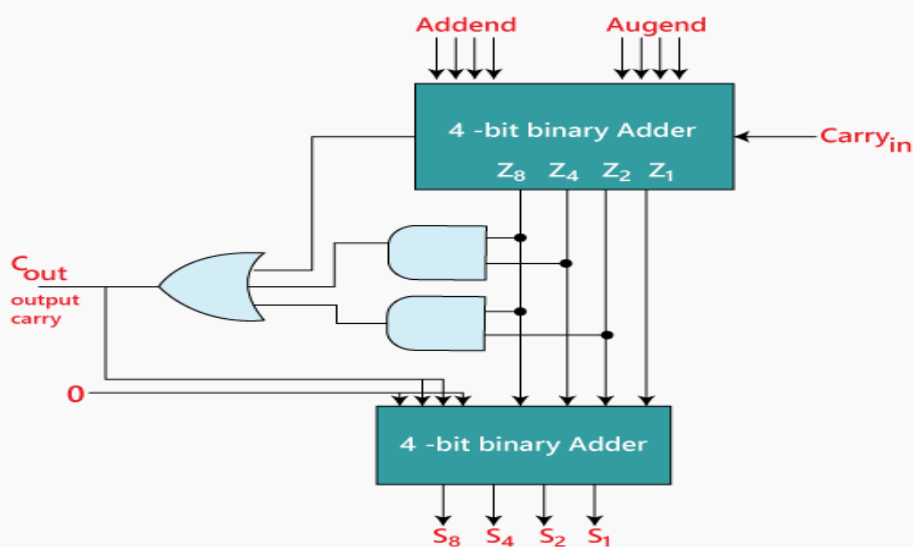
gate. The first half subtractor has two single-bit binary inputs A and B. The half subtractor produces two outputs, i.e., 'Diff' and 'Borrow'. The 'Diff' output of the first subtractor will be the first input of the second half subtractor, and the 'Borrow' output of the first subtractor will be the second input of the second half subtractor. The second half subtractor will again provide 'Diff' and 'Borrow'. The final outcome of the Full subtractor circuit is the 'Diff' bit. In order to find the final output of the 'Borrow', we provide the 'Borrow' of the first and the second subtractor into the OR gate. The outcome of the OR gate will be the final carry 'Borrow' of full subtractor circuit.

BCD Adder

The BCD-Adder is used in the computers and the calculators that perform arithmetic operation directly in the decimal number system. The BCD-Adder accepts the binary-coded form of decimal numbers. The Decimal-Adder requires a minimum of nine inputs and five outputs.

sum is between 1 to 9, the Binary and the BCD code is the same. But for 10 to 19 decimal numbers, both the codes are different.

The binary sum combinations from 10 to 19 give invalid BCD. Once the circuit found the invalid BCD, the circuit adds the binary number of 6 into the invalid BCD code to make it valid.



1. We take a 4-bit Binary-Adder, which takes addend and augend bits as an input with an input carry '**Carry in**'.
2. The Binary-Adder produces five outputs, i.e., Z_8 , Z_4 , Z_2 , Z_1 , and an output carry K.
3. With the help of the output carry K and Z_8 , Z_4 , Z_2 , Z_1 outputs, the logical circuit is designed to identify the C_{out}

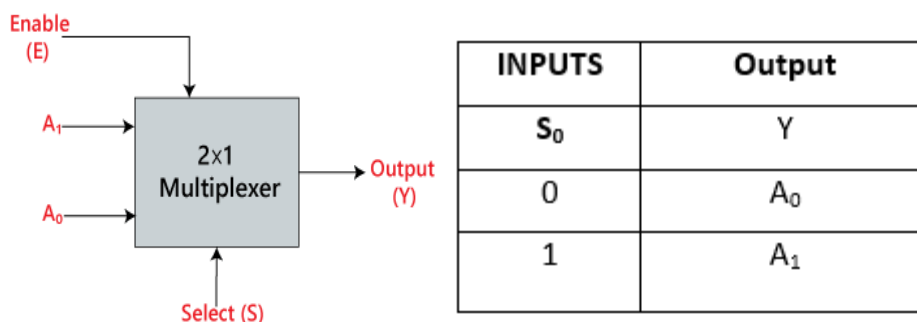
4. The Z8, Z4, Z2, and Z1 outputs of the binary adder are passed into the 2nd 4-bit binary adder as an Augend.
5. The addend bit of the 2nd 4-bit binary adder is designed in such a way that the 1st and the 4th bit of the addend number are 0 and the 2nd and the 3rd bit are the same as C_{out}. When the value of C_{out} is 0, the addend number will be 0000, which produce the same result as the 1st 4-bit binary number. But when the value of the C_{out} is 1, the addend bit will be 0110, i.e., 6, which adds with the augent to get the valid BCD number.

Multiplexer

A multiplexer is a combinational circuit that has 2^n input lines and a single output line. Simply, the multiplexer is a multi-input and single-output combinational circuit. The binary information is received from the input lines and directed to the output line. On the basis of the values of the selection lines, one of these data inputs will be connected to the output.

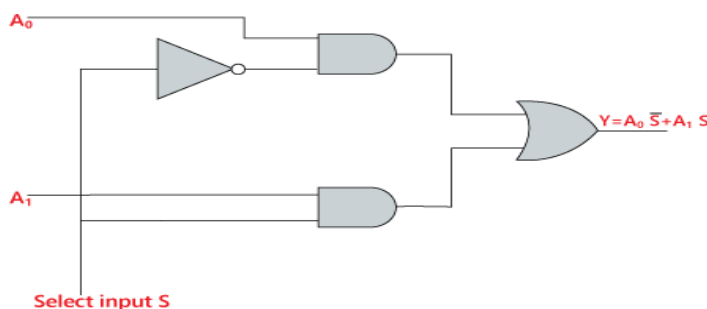
2×1 Multiplexer:

In 2×1 multiplexer, there are only two inputs, i.e., A_0 and A_1 , 1 selection line, i.e., S_0 and single outputs, i.e., Y . On the basis of the combination of inputs which are present at the selection line S^0 , one of these 2 inputs will be connected to the output.



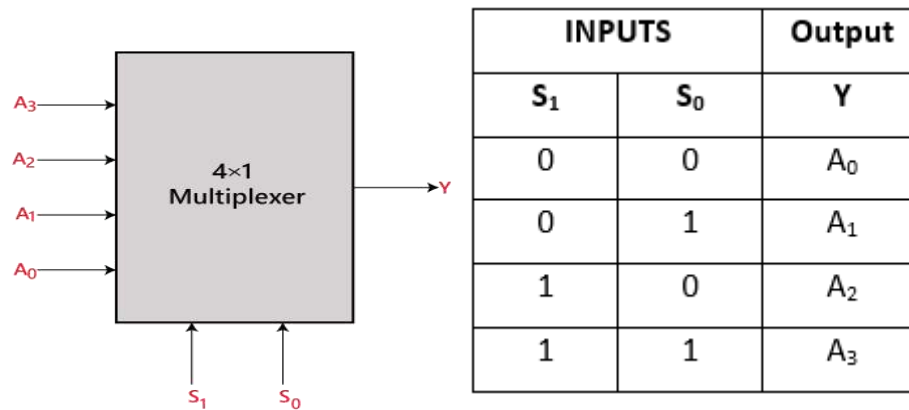
The logical expression of the term Y is as follows:

$$Y = S_0' \cdot A_0 + S_0 \cdot A_1$$



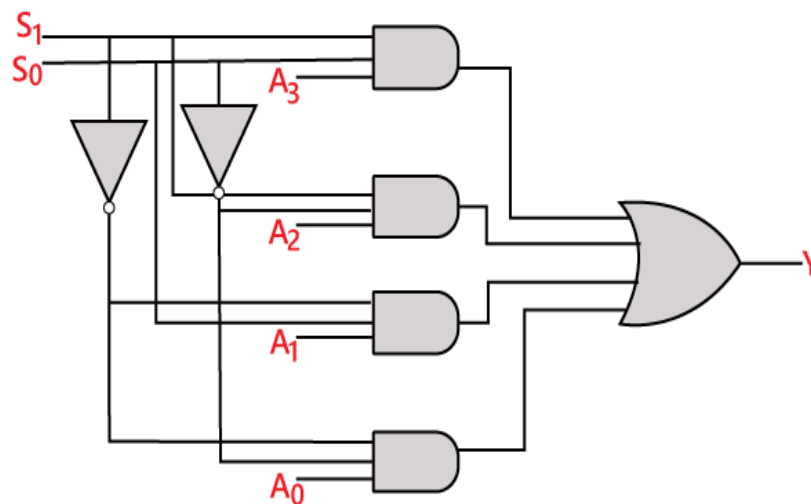
4×1 Multiplexer:

In the 4×1 multiplexer, there is a total of four inputs, i.e., A_0 , A_1 , A_2 , and A_3 , 2 selection lines, i.e., S_0 and S_1 and single output, i.e., Y . On the basis of the combination of inputs that are present at the selection lines S^0 and S_1 , one of these 4 inputs are connected to the output.



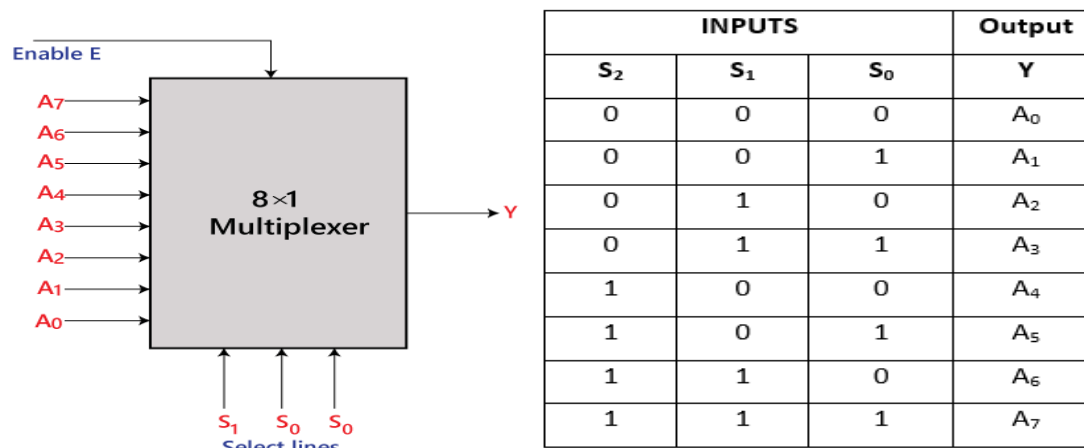
The logical expression of the term Y is as follows:

$$Y = S_1' S_0' A_0 + S_1' S_0 A_1 + S_1 S_0' A_2 + S_1 S_0 A_3$$



8 to 1 Multiplexer

In the 8 to 1 multiplexer, there are total eight inputs, i.e., $A_0, A_1, A_2, A_3, A_4, A_5, A_6,$ and A_7 , 3 selection lines, i.e., S_0, S_1 and S_2 and single output, i.e., Y . On the basis of the combination of inputs that are present at the selection lines $S^0, S^1,$ and S_2 , one of these 8 inputs are connected to the output.



The logical expression of the term Y is as follows:

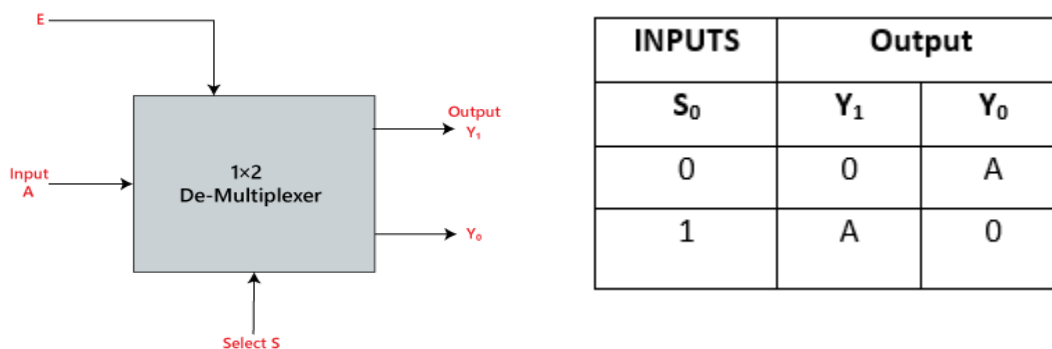
$$Y = S_0' \cdot S_1' \cdot S_2' \cdot A_0 + S_0 \cdot S_1' \cdot S_2' \cdot A_1 + S_0' \cdot S_1 \cdot S_2' \cdot A_2 + S_0 \cdot S_1 \cdot S_2' \cdot A_3 + S_0' \cdot S_1' \cdot S_2 \cdot A_4 + S_0 \cdot S_1' \cdot S_2 \cdot A_5 + S_0' \cdot S_1 \cdot S_2 \cdot A_6 + S_0 \cdot S_1 \cdot S_2 \cdot A_7$$

De-multiplexer

A De-multiplexer is a combinational circuit that has only 1 input line and 2^N output lines. Simply, the multiplexer is a single-input and multi-output combinational circuit. The information is received from the single input lines and directed to the output line. On the basis of the values of the selection lines, the input will be connected to one of these outputs. De-multiplexer is opposite to the multiplexer.

1×2 De-multiplexer:

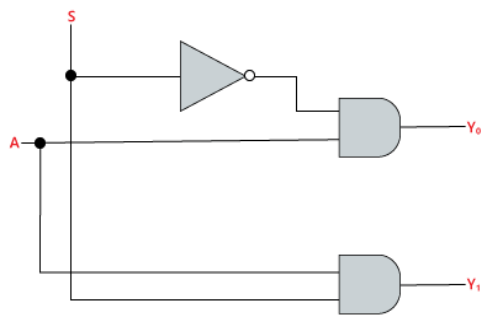
In the 1 to 2 De-multiplexer, there are only two outputs, i.e., Y_0 , and Y_1 , 1 selection lines, i.e., S_0 , and single input, i.e., A. On the basis of the selection value, the input will be connected to one of the outputs. The block diagram and the truth table of the 1×2 multiplexer are given below.



The logical expression of the term Y is as follows:

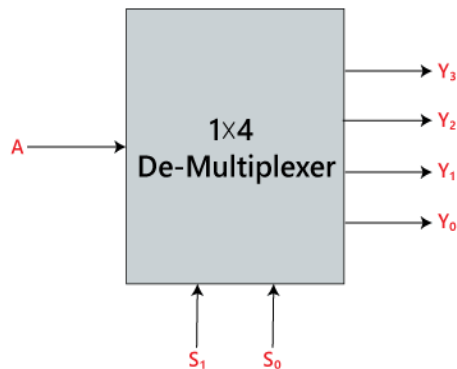
$$Y_0 = S_0' \cdot A$$

$$Y_1 = S_0 \cdot A$$



1×4 De-multiplexer:

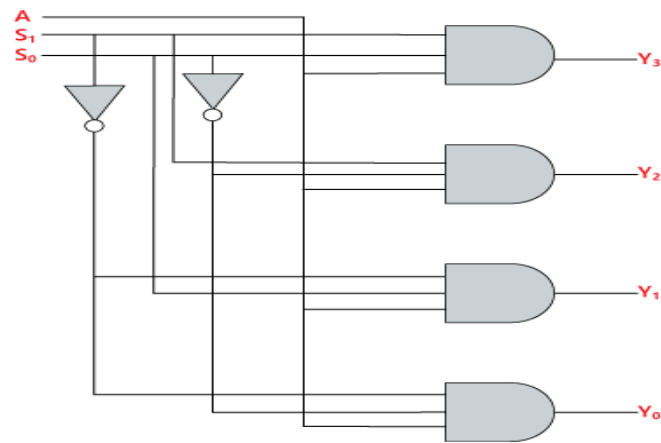
In 1 to 4 De-multiplexer, there are total of four outputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 , 2 selection lines, i.e., S_0 and S_1 and single input, i.e., A . On the basis of the combination of inputs which are present at the selection lines S_0 and S_1 , the input be connected to one of the outputs.



INPUTS		Output			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	A
0	1	0	0	A	0
1	0	0	A	0	0
1	1	A	0	0	0

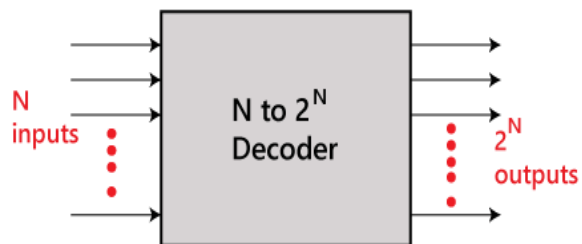
The logical expression of the term Y is as follows:

$$\begin{array}{llll}
 Y_0 = S_1' & & S_0' & A \\
 Y_1 = S_1' & & & S_0 A \\
 Y_2 = S_1 S_0' & & & A \\
 Y_3 = S_1 S_0 A & & &
 \end{array}$$



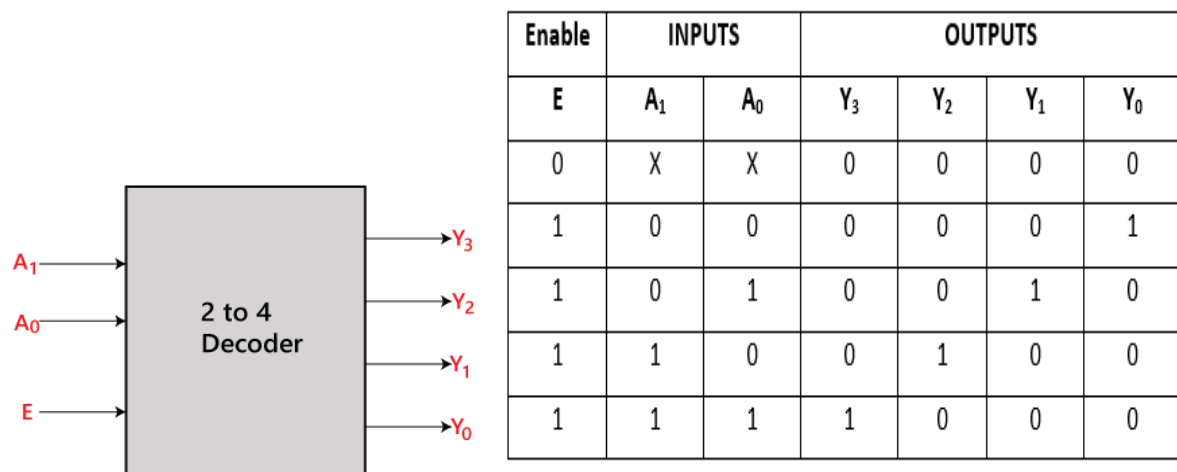
Decoder

The combinational circuit that change the binary information into 2^N output lines is known as **Decoders**. The binary information is passed in the form of N input lines. The output lines define the 2^N -bit code for the binary information. The produced 2^N -bit output code is equivalent to the binary information.



2 to 4 line decoder:

In the 2 to 4 line decoder, there is a total of three inputs, i.e., A_0 , and A_1 and E and four outputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 . For each combination of inputs, when the enable 'E' is set to 1, one of these four outputs will be 1.



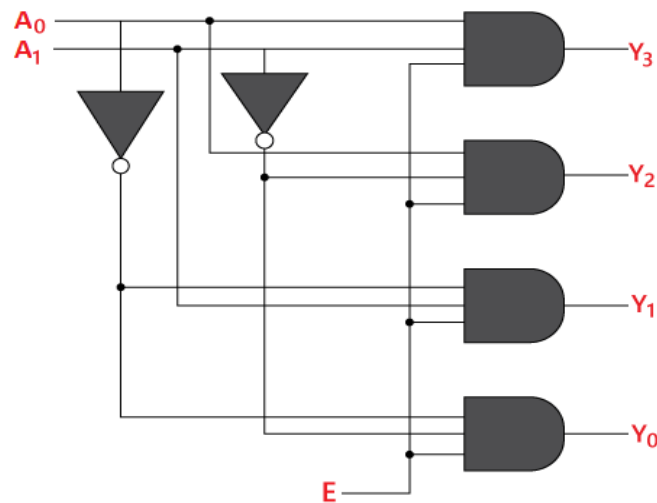
The logical expression of the term Y_0 , Y_0 , Y_2 , and Y_3 is as follows:

$$Y_3 = E \cdot A_1 \cdot A_0$$

$$Y_2 = E \cdot A_1 \cdot A_0'$$

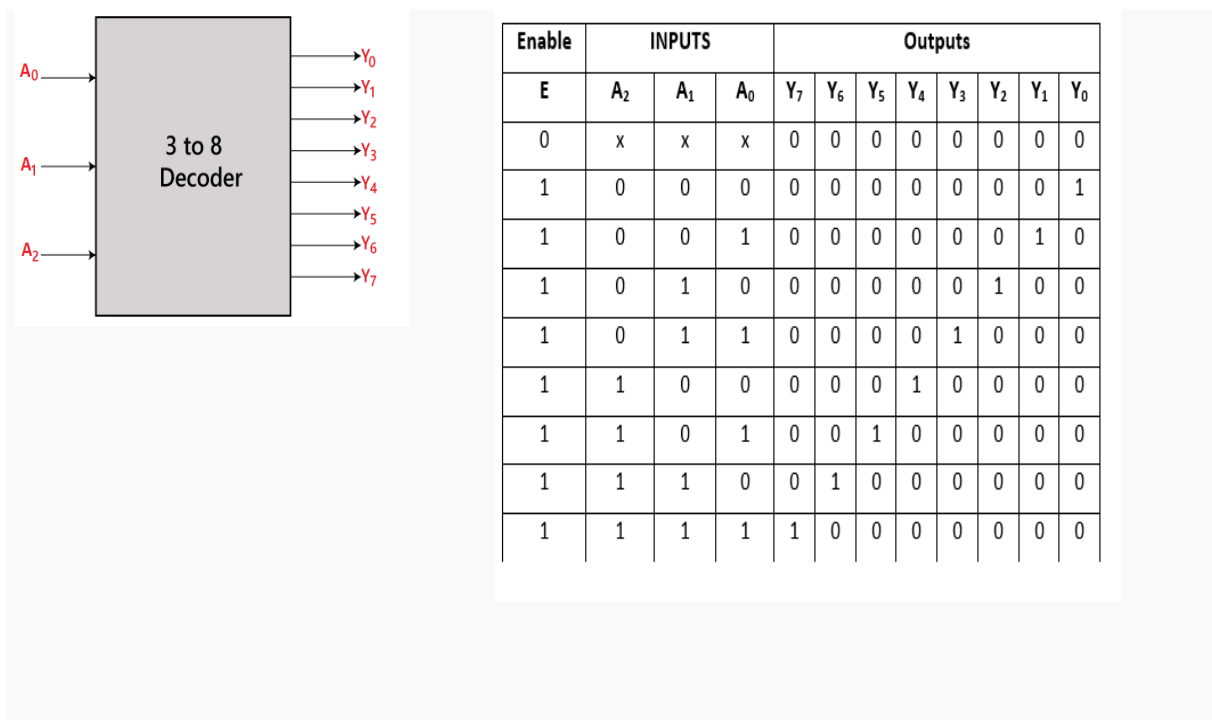
$$Y_1 = E \cdot A_1' \cdot A_0$$

$$Y_0 = E \cdot A_1' \cdot A_0'$$



3 to 8 line decoder:

The 3 to 8 line decoder is also known as **Binary to Octal Decoder**. In a 3 to 8 line decoder, there is a total of eight outputs, i.e., $Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6,$ and Y_7 and three outputs, i.e., $A_0, A_1,$ and A_2 . This circuit has an enable input 'E'. Just like 2 to 4 line decoder, when enable 'E' is set to 1, one of these four outputs will be 1.

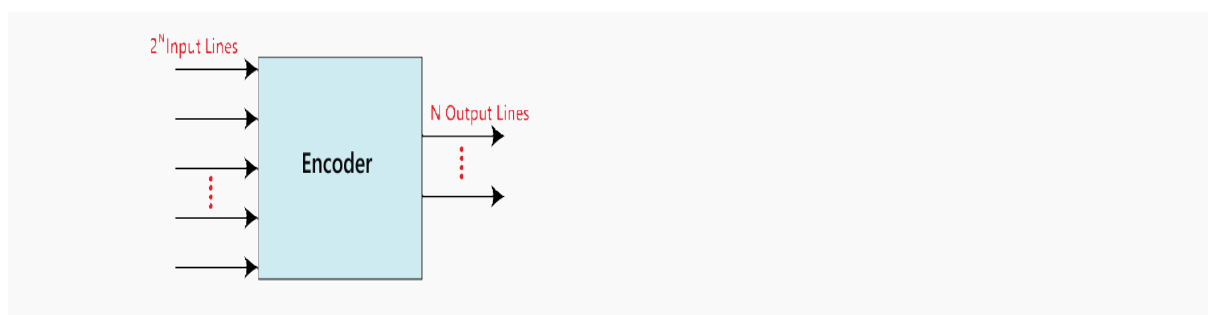


The logical expression of the term $Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6,$ and Y_7 is as follows:

$$\begin{aligned}
 Y_0 &= A_0' \cdot A_1' \cdot A_2' \\
 Y_1 &= A_0 \cdot A_1' \cdot A_2' \\
 Y_2 &= A_0' \cdot A_1 \cdot A_2' \\
 Y_3 &= A_0 \cdot A_1 \cdot A_2' \\
 Y_4 &= A_0' \cdot A_1' \cdot A_2 \\
 Y_5 &= A_0 \cdot A_1' \cdot A_2 \\
 Y_6 &= A_0' \cdot A_1 \cdot A_2 \\
 Y_7 &= A_0 \cdot A_1 \cdot A_2
 \end{aligned}$$

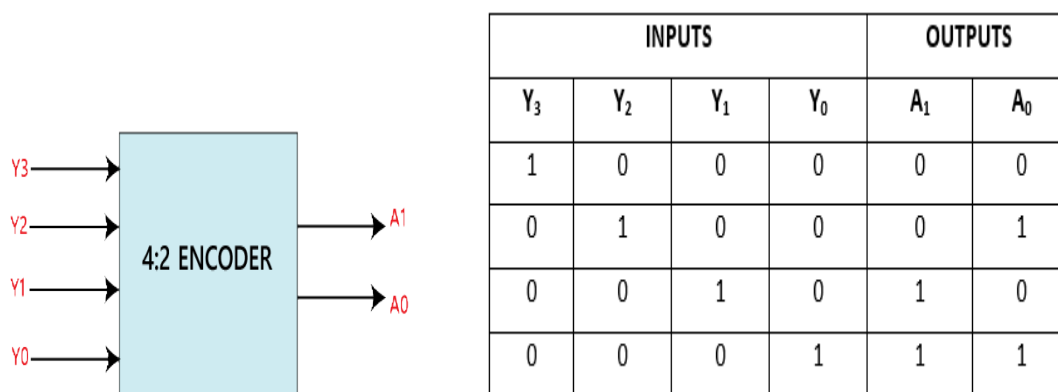
Encoders

The combinational circuits that change the binary information into N output lines are known as **Encoders**. The binary information is passed in the form of 2^N input lines. The output lines define the N-bit code for the binary information. In simple words, the **Encoder** performs the reverse operation of the **Decoder**. At a time, only one input line is activated for simplicity. The produced N-bit output code is equivalent to the binary information.



4 to 2 line Encoder:

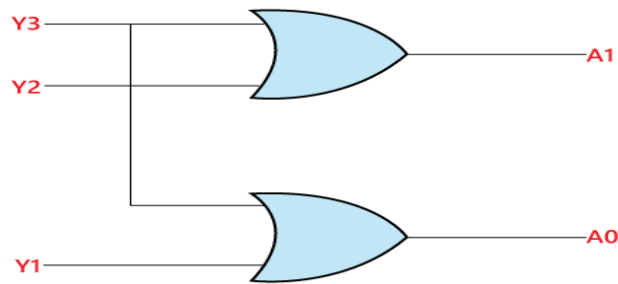
In 4 to 2 line encoder, there are total of four inputs, i.e., Y_0 , Y_1 , Y_2 , and Y_3 , and two outputs, i.e., A_0 and A_1 . In 4-input lines, one input-line is set to true at a time to get the respective binary code in the output side.



The logical expression of the term A_0 and A_1 is as follows:

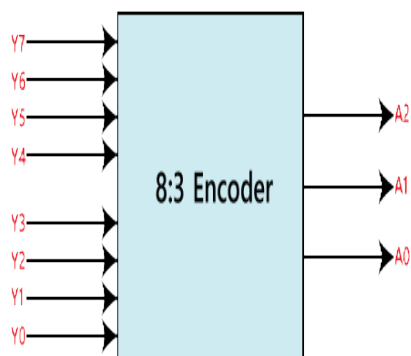
$$A_1 = Y_3 + Y_2$$

$$A_0 = Y_3 + Y_1$$



8 to 3 line Encoder:

The 8 to 3 line Encoder is also known as **Octal to Binary Encoder**. In 8 to 3 line encoder, there is a total of eight inputs, i.e., $Y_0, Y_1, Y_2, Y_3, Y_4, Y_5, Y_6,$ and Y_7 and three outputs, i.e., $A_0, A_1,$ and A_2 . In 8-input lines, one input-line is set to true at a time to get the respective binary code in the output side.



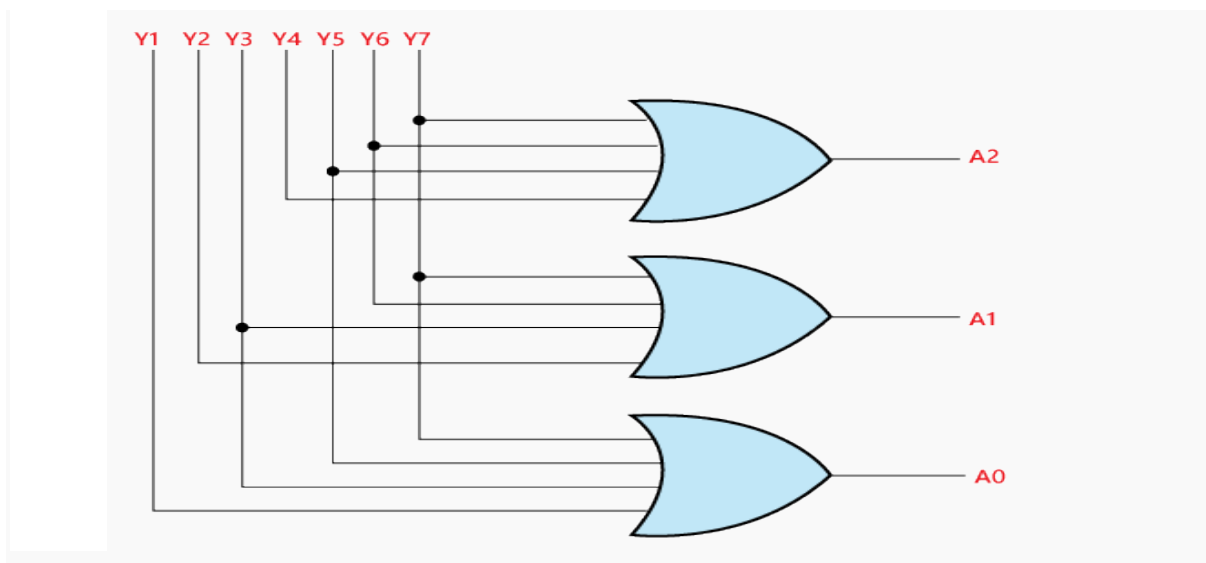
INPUTS								OUTPUTS		
Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0	A_2	A_1	A_0
0	0	0	0	0	0	0	1	0	0	0
0	0	0	0	0	0	1	0	0	0	1
0	0	0	0	0	1	0	0	0	1	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	0
0	0	1	0	0	0	0	0	1	0	1
0	1	0	0	0	0	0	0	1	1	0
1	0	0	0	0	0	0	0	1	1	1

The logical expression of the term $A_0, A_1,$ and A_2 are as follows:

$$A_2 = Y_4 + Y_5 + Y_6 + Y_7$$

$$A_1 = Y_2 + Y_3 + Y_6 + Y_7$$

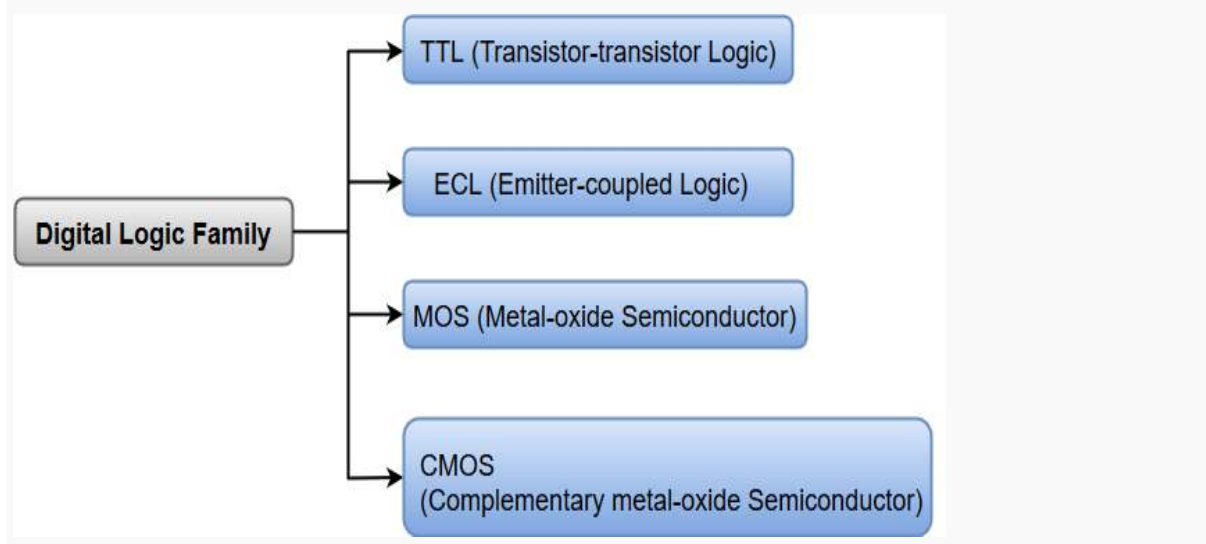
$$A_0 = Y_7 + Y_5 + Y_3 + Y_1$$



Digital Logic Families

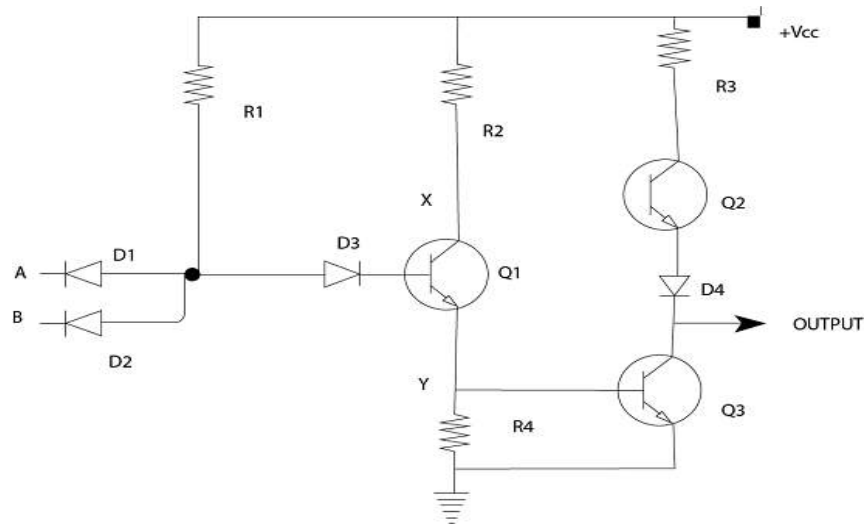
In Digital Designs, our primary aim is to create an Integrated Circuit (IC). A Circuit configuration or arrangement of the circuit elements in a special manner will result in a particular Logic Family.

The most popular among the digital logic families include:



TTL (Transistor-transistor Logic)

The TTL technology was an upgraded version of a previous technology called as DTL (Diode-Transistor Logic). The DTL technology used to have diodes and transistors for the basic NAND gate. TTL came in existence when these diodes are replaced with transistors to improve the circuit operation.



Features of TTL Family:

- The overall power supply voltage for TTL circuit is 5 volts, and the two logic levels are approximately 0 and 3.5 volts.
- A TTL circuit can support at most 10 gates at its output.
- The average propagation delay for a TTL circuit is about 9ns.

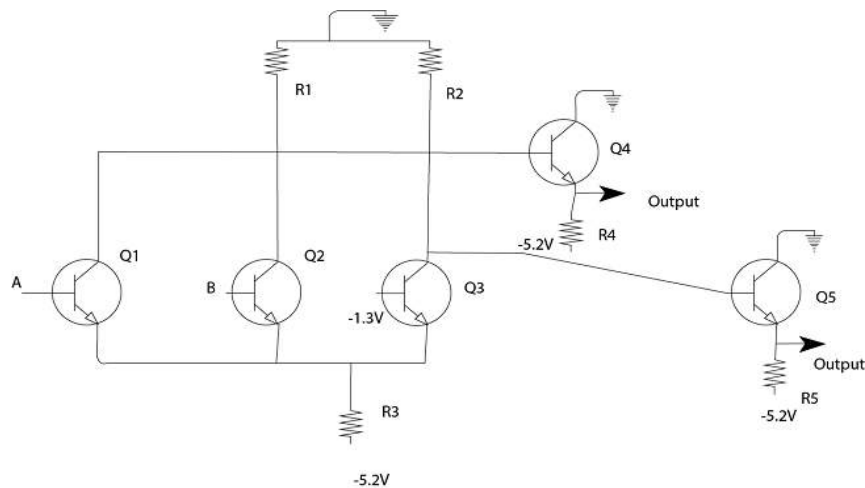
TTL Applications

- TTL is used as a switching device in driving lamps and relays.
- TTL is used in controller application for providing 0 to 5Vs.
- TTL families are mostly used in processors of minicomputers like DEC VAX.
- It is also used in printers and video display terminals.

ECL (Emitter-coupled Logic)

The ECL technology provides the highest-speed digital circuits in integrated form. An ECL circuit is used in supercomputers and signal processors where high speed is essential.

The transistors in ECL gates operate in a non-saturated state, a condition that allows the achievement of propagation delays of 1 to 2 nanoseconds.

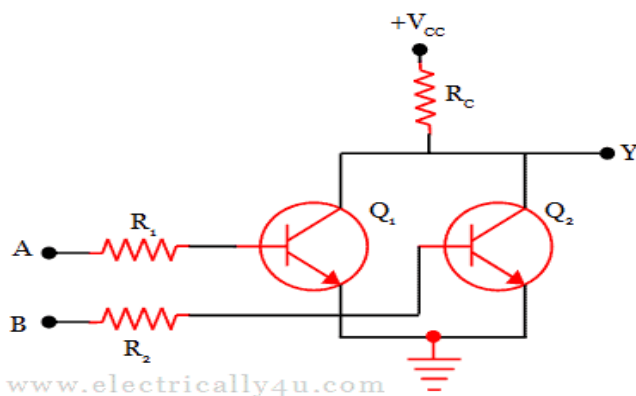


Features of ECL Family

- The logic gates continuously draw current even in the inactive state. Hence power consumption is more as compared to other logic families.
- ECL uses bipolar transistor logic where the transistors are not operated in the saturation region.
- The average propagation delay for an ECL gate is about 0.5 to 2ns.

Resistor Transistor Logic (RTL)

The RTL circuit consists of resistors at inputs and transistors at the output side. Transistors are used as the switching device. The emitter of the transistor is connected to the ground. The collector terminals are tied together and given to the supply through the resistor R_C . The collector resistor is known as a passive pull-up resistor.



The figure shows the circuit diagram of 2-input RTL NOR gate. Q_1 and Q_2 are the two transistors. A and B are the two inputs, given to the base of two transistors and Y is the output.

When both the inputs A and B are at 0V or logic 0, it is not enough to turn on the gates of both the transistor. So the transistors will not conduct. Due to this, the voltage $+V_{CC}$ will appear at the output Y. Hence the output is logic 1 or logic HIGH at terminal Y.

When any one of the inputs, either A or B is given HIGH voltage or logic 1, then the transistor with HIGH gate input will be turned on. This will make a path for the supply voltage to go to the ground through the resistor R_C and transistor. Thus there will be 0 v at the output terminal Y.

When both the inputs are HIGH, it will drive both the transistor to turn on. It will make a path for the supply voltage to flow to the ground through resistor R_C and transistor. Therefore, there will be 0v at the output terminal Y.

NAND and NOR gate using CMOS Technology

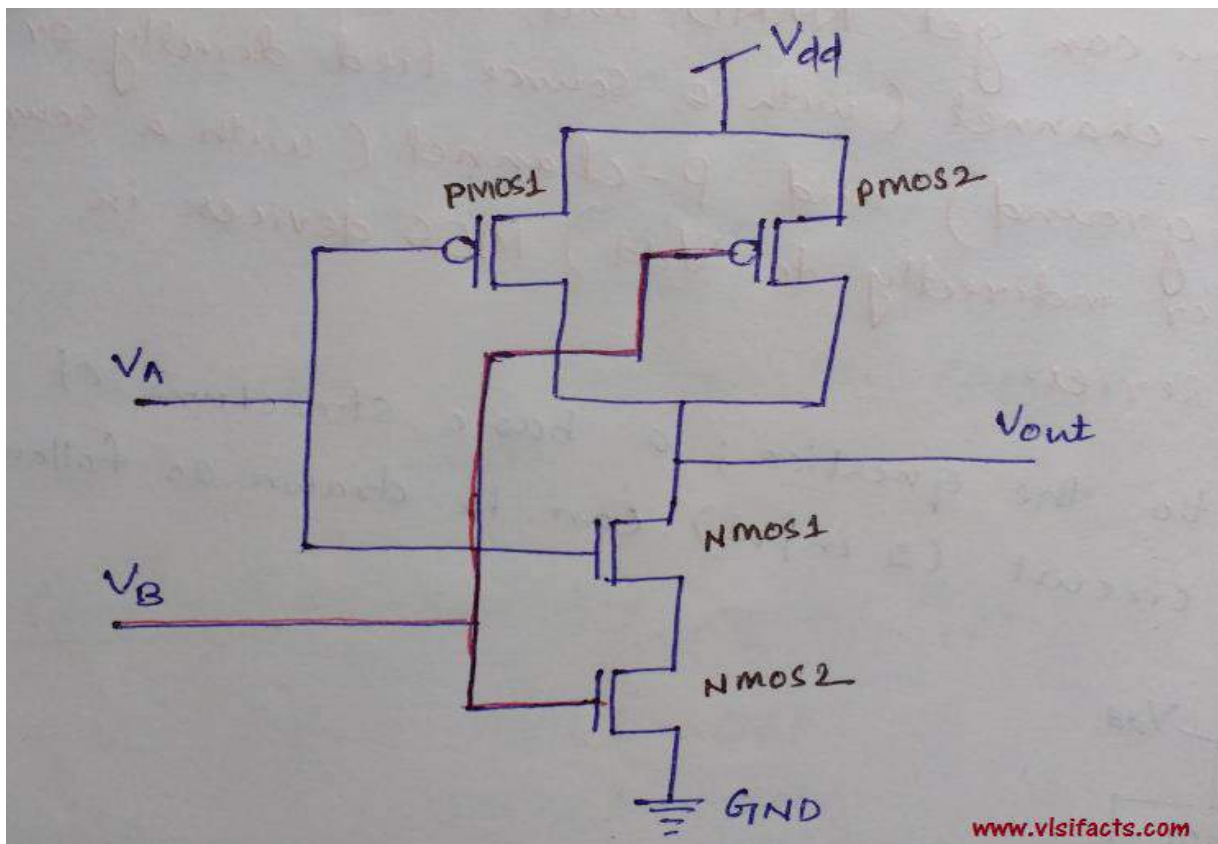
For the design of any circuit with the CMOS technology; We need parallel or series connections of nMOS and pMOS with a nMOS source tied directly or indirectly to ground and a pMOS source tied directly or indirectly to V_{dd} .

2

Input

NAND

Gate



V_A	V_B	V_{out}
Low	Low	High
Low	High	High
High	Low	High
High	High	Low

Case-1 : V_A – Low & V_B – Low

As V_A and V_B both are low, both the pMOS will be ON and both the nMOS will be OFF. So the output V_{out} will get two paths through two ON pMOS to get connected with V_{dd} . The output will be charged to the V_{dd} level. The output line will not get any path to the GND as both the nMOS are off. So, there is no path through which the output line can discharge. The output line will maintain the voltage level at V_{dd} ; so, **High**.

Case-2 : V_A – Low & V_B – High

V_A – Low: pMOS1 – ON; nMOS1 – OFF

V_B – High: pMOS2 – OFF; nMOS2 – ON

pMOS1 and pMOS2 are in parallel. Though pMOS2 is OFF, still the output line will get a path through pMOS1 to get connected with V_{dd} . nMOS1 and nMOS2 are in series. As

nMOS1 is OFF, so V_{out} will not be able to find a path to GND to get discharged. This in turn results the V_{out} to be maintained at the level of V_{dd} ; so, **High**.

Case-3 : V_A – High & V_B – Low

V_A – **High**: pMOS1 – OFF; nMOS1 – ON

V_B – **Low**: pMOS2 – ON; nMOS2 – OFF

The explanation is similar as case-2. V_{out} level will be **High**.

Case-4 : V_A – High & V_B – High

V_A – **High**: pMOS1 – OFF; nMOS1 – ON

V_B – **High**: pMOS2 – OFF; nMOS2 – ON

In this case, both the pMOS are OFF. So, V_{out} will not find any path to get connected with V_{dd} . As both the nMOS are ON, the series connected nMOS will create a path from V_{out} to GND. Since, the path to ground is established, V_{out} will be discharged; so, **Low**.

2 Input NOR Gate

Case-1 : V_A – Low & V_B – Low

V_A – **Low**: pMOS1 – ON; nMOS1 – OFF

V_B – **Low**: pMOS2 – ON; nMOS2 – OFF

Path establishes from V_{dd} to V_{out} through the series connected ON pMOS transistors and V_{out} gets charged to V_{dd} level. No path from V_{out} to GND. Therefore, no discharging and hence V_{out} will be **High**.

Case-2 : V_A – Low & V_B – High

V_A – **Low**: pMOS1 – ON; nMOS1 – OFF

V_B – **High**: pMOS2 – OFF; nMOS2 – ON

In this case path establishes from V_{out} to GND through nMOS2, but no path to V_{dd} . So, V_{out} would get discharged and will be at level **Low**.

Case-3 : V_A – High & V_B – Low

V_A – **High**: pMOS1 – OFF; nMOS1 – ON

V_B – **Low**: pMOS2 – ON; nMOS2 – OFF

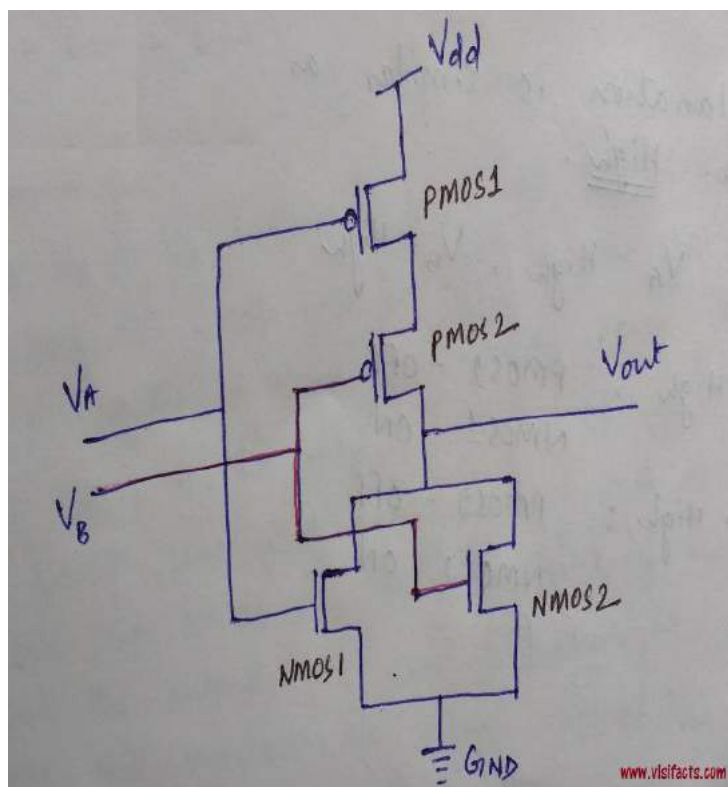
The explanation is similar as case-2. V_{out} will be at level **Low**.

Case-4 : V_A – High & V_B – High

V_A – **High**: pMOS1 – OFF; nMOS1 – ON

V_B – **High**: pMOS2 – OFF; nMOS2 – ON

No path to V_{dd} . Path establishes from V_{out} to GND. So, V_{out} will be at level **Low**.



V_A	V_B	V_{out}
Low	Low	High
Low	High	Low
High	Low	Low
High	High	Low

UNIT IV

SEQUENTIAL DIGITAL CIRCUITS

The sequential circuit is a special type of circuit that has a series of inputs and outputs. The outputs of the sequential circuits depend on both the combination of present inputs and previous outputs. The previous output is treated as the present state. So, the sequential circuit contains the combinational circuit and its memory storage elements. A sequential circuit doesn't need to always contain a combinational circuit. So, the sequential circuit can contain only the memory element.

Types of Sequential Circuits

Asynchronous sequential circuits

The clock signals are not used by the **Asynchronous sequential circuits**. The asynchronous circuit is operated through the pulses. So, the changes in the input can change the state of the circuit. The asynchronous circuits do not use clock pulses. The internal state is changed when the input variable is changed.

Synchronous sequential circuits

In synchronous sequential circuits, synchronization of the memory element's state is done by the clock signal. The output is stored in either flip-flops or latches (memory devices). The synchronization of the outputs is done with either only negative edges of the clock signal or only positive edges.

Clock signal

A clock signal is a periodic signal in which ON time and OFF time need not be the same. When ON time and OFF time of the clock signal are the same, a square wave is used to represent the clock signal.



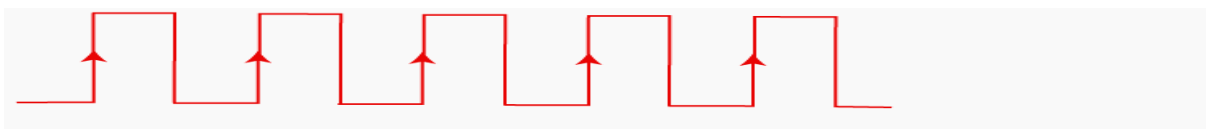
A clock signal is considered as the square wave. Sometimes, the signal stays at logic, either high 5V or low 0V, to an equal amount of time. It repeats with a certain time period, which will be equal to twice the 'ON time' or 'OFF time'.

Types of Triggering

In clock signal of edge triggering, two types of transitions occur, i.e., transition either from Logic Low to Logic High or Logic High to Logic Low.

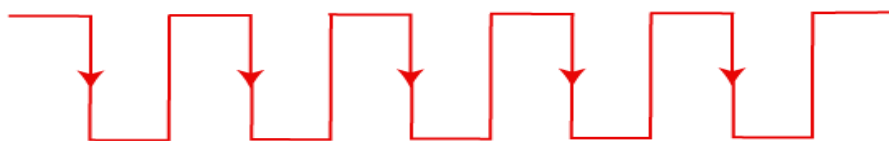
Positive edge triggering

The transition from Logic Low to Logic High occurs in the clock signal of positive edge triggering. So, in positive edge triggering, the circuit is operated with such type of clock signal.



Negative edge triggering

The transition from Logic High to Logic low occurs in the clock signal of negative edge triggering. So, in negative edge triggering, the circuit is operated with such type of clock signal.



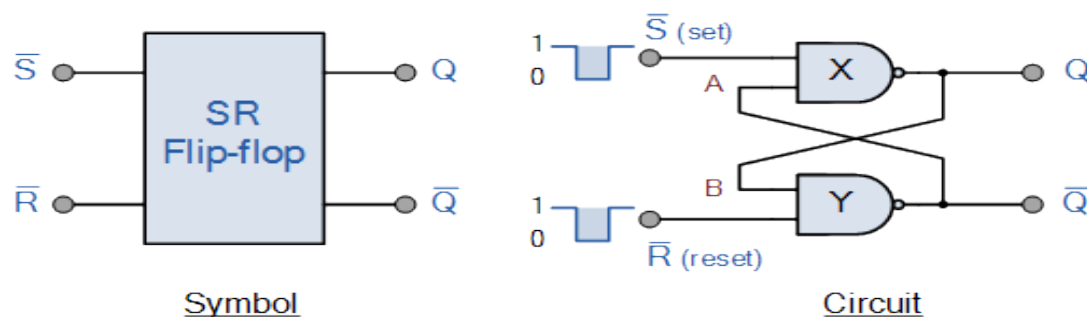
Basics of Flip Flop

A circuit that has two stable states is treated as a **flip flop**. These stable states are used to store binary data that can be changed by applying varying inputs. The flip flops are the fundamental building blocks of the digital system. The flip flop is the basic storage element. The latches and flip flops are the basic storage elements but different in working.

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 one which will “RESET” the device (meaning the output = “0”), labelled **R**. Then the SR description stands for “Set-Reset”. The reset input resets the flip-flop back to its original state with an output **Q** that will be either at a logic level “1” or logic “0” depending upon this set/reset condition.



The simplest way to make any basic single bit set-reset SR flip-flop is to connect together a pair of cross-coupled 2-input NAND gates as shown, to form a Set-Reset Bistable also known as an active LOW SR NAND Gate Latch, so that there is feedback from each output to one of the other NAND gate inputs. This device consists of two inputs, one called the *Set*, S and the other called the *Reset*, R with two corresponding outputs Q and its inverse or complement \bar{Q} .

Case 1: $R = 1$ and $S = 1$

When both the S and R inputs are HIGH, the output remains in previous state i.e., it holds the previous data.

Case 2: $R = 1$ and $S = 0$

When R input is HIGH and S input is LOW, the flip flop will be in SET state. As R is HIGH, the output of NAND gate B i.e., \bar{Q} becomes LOW. This causes both the inputs of NAND gate A to become LOW and hence, the output of NAND gate A i.e., Q becomes HIGH.

Case 3: $R = 0$ and $S = 1$

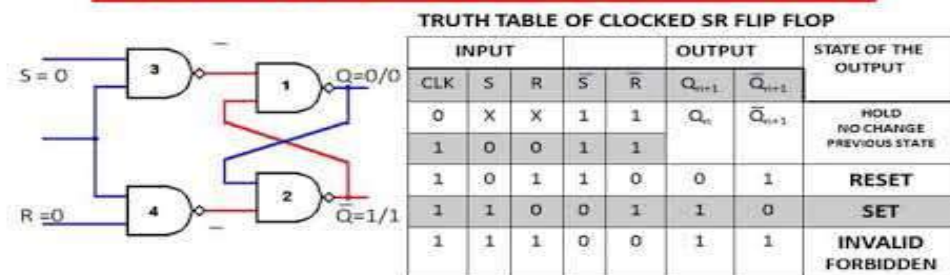
When R input is LOW and S input is HIGH, the flip flop will be in RESET state. As S is HIGH, the output of NAND gate A i.e., Q becomes LOW. This causes both the inputs of NAND gate B to become LOW and hence, the output of NAND gate B i.e., \bar{Q} becomes HIGH.

Case 3: $R = 0$ and $S = 0$

When both the R and S inputs are LOW, the flip flop will be in undefined state. Because the low inputs of S and R, violates the rule of flip – flop that the outputs should complement to each other. So, the flip flop is in undefined state (or forbidden state).

Clocked SR Flip – Flop

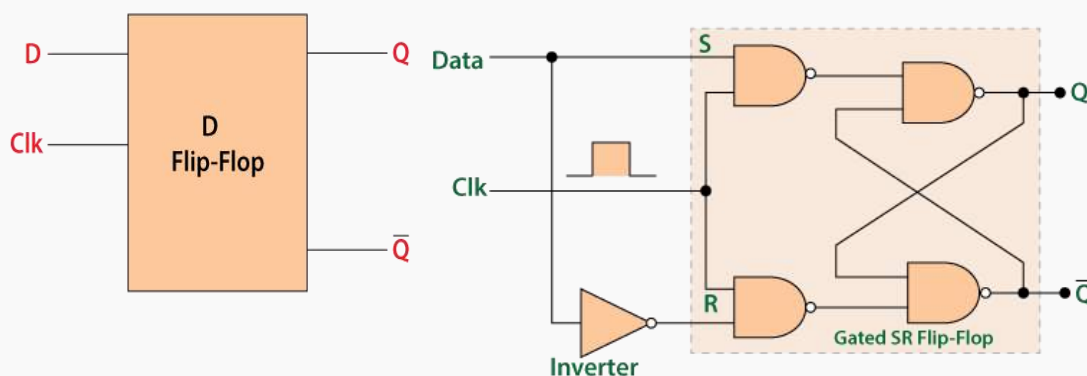
CLOCKED SR FLIP FLOP



This circuit is formed by adding two NAND gates to NAND based SR flip – flop. The inputs are active high as the extra NAND gate inverts the inputs. A clock pulse is given as input to both the extra NAND gates. Hence the transition of the clock pulse is a key factor in functioning if this device.

D Flip Flop

The D flip flop is the most important flip flop from other clocked types. It ensures that at the same time, both the inputs, i.e., S and R, are never equal to 1. The Delay flip-flop is designed using a gated [SR flip-flop](#) with an inverter connected between the inputs allowing for a single input D(Data).



Clock	D	Q	Q'	Description
↓ » 0	X	Q	Q'	Memory no change
↑ » 1	0	0	1	Reset Q » 0
↑ » 1	1	1	0	Set Q » 1

In D flip flop, the single input "D" is referred to as the "Data" input. When the data input is set to 1, the flip flop would be set, and when it is set to 0, the flip flop would change and become reset.

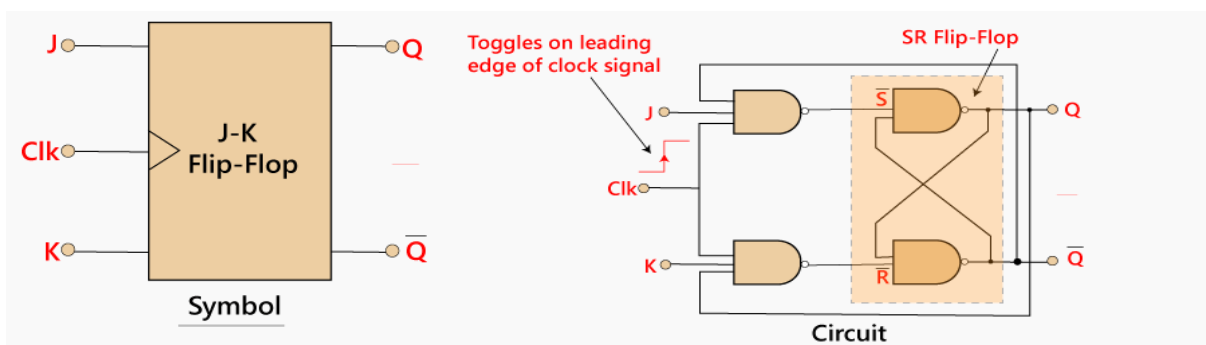
When the clock input is set to true, the D input condition is only copied to the output Q. This forms the basis of another sequential device referred to as **D Flip Flop**.

When the clock input is set to 1, the "set" and "reset" inputs of the flip-flop are both set to 1. So it will not change the state and store the data present on its output before the clock transition occurred. In simple words, the output is "latched" at either 0 or 1.

JK Flip Flop

The [JK flip flop](#) is one of the most used flip flops in digital circuits. The JK flip flop is a universal flip flop having two inputs 'J' and 'K'.

The JK Flip Flop is a gated SR flip-flop having the addition of a clock input circuitry. The invalid or illegal output condition occurs when both of the inputs are set to 1 and are prevented by the addition of a clock input circuit. So, the JK flip-flop has four possible input combinations, i.e., 1, 0, "no change" and "toggle".



Same as for SR Latch	Clock	Input		Output		Description
	Clk	J	K	Q	Q'	
	X	0	0	1	0	Memory no change
	X	0	0	0	1	
	$\bar{\downarrow}$	0	1	1	0	Reset Q>>0
	X	0	1	0	1	Set Q>>1
	$\bar{\downarrow}$	1	0	0	1	
Toggle action	$\bar{\downarrow}$	1	1	0	1	Toggle
	$\bar{\downarrow}$	1	1	1	0	

In SR flip flop, both the inputs 'S' and 'R' are replaced by two inputs J and K. It means the J and K input equates to S and R, respectively.

The two 2-input AND gates are replaced by two 3-input [NAND gates](#). The third input of each gate is connected to the outputs at Q and Q'. The cross-coupling of the SR flip-flop permits the previous invalid condition of (S = "1", R = "1") to be used to produce the "toggle action" as the two inputs are now interlocked.

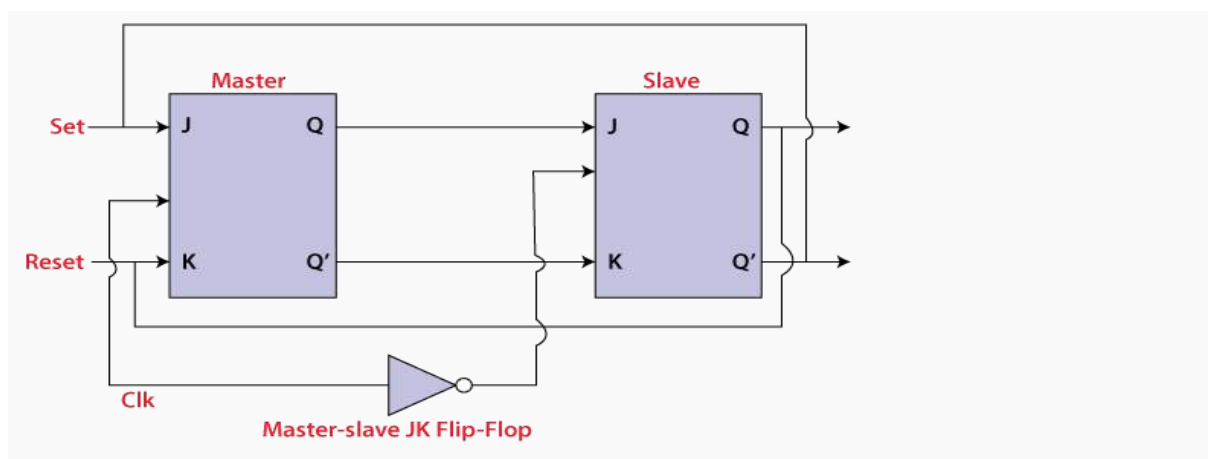
If the circuit is "set", the J input is interrupted from the "0" position of Q' through the lower NAND gate. If the circuit is "RESET", K input is interrupted from 0 positions of Q through the upper NAND gate. Since Q and Q' are always different, we can use them to control the input. When both inputs 'J' and 'K' are set to 1, the JK toggles the flip flop. When both of the inputs of JK flip flop are set to 1 and clock input is also pulse "High" then from the SET state to a RESET state, the circuit will be toggled.

Master-Slave JK Flip Flop

In "JK Flip Flop", when both the inputs and CLK set to 1 for a long time, then Q output toggle until the CLK is 1. Thus, the uncertain or unreliable output produces. This problem is referred to as a race-round condition in JK flip-flop and avoided by ensuring that the CLK set to 1 only for a very short time.

The master-slave flip flop is constructed by combining two [JK flip flops](#). These flip flops are connected in a series configuration. In these two flip flops, the 1st flip flop work as "master", called the master flip flop, and the 2nd work as a "slave", called slave flip flop. The master-slave flip flop is designed in such a way that the output of the "master" flip flop is passed to both the inputs of the "slave" [flip flop](#). The output of the "slave" flip flop is passed to inputs of the master flip flop.

In "master-slave flip flop", apart from these two flip flops, an inverter or [NOT gate](#) is also used. For passing the inverted clock pulse to the "slave" flip flop, the inverter is connected to the clock's pulse. In simple words, when CP set to false for "master", then CP is set to true for "slave", and when CP set to true for "master", then CP is set to false for "slave".

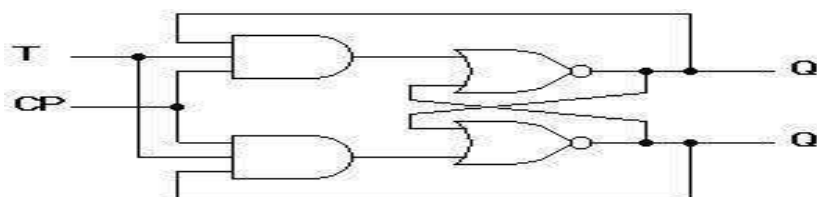


- When the clock pulse is true, the slave flip flop will be in the isolated state, and the system's state may be affected by the J and K inputs. The "slave" remains isolated until the CP is 1. When the CP set to 0, the master flip-flop passes the information to the slave flip flop to obtain the output.
- The master flip flop responds first from the slave because the master flip flop is the positive level trigger, and the slave flip flop is the negative level trigger.

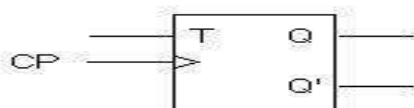
- The output $Q'=1$ of the master flip flop is passed to the slave flip flop as an input K when the input J set to 0 and K set to 1. The clock forces the slave flip flop to work as reset, and then the slave copies the master flip flop.
- When $J=1$, and $K=0$, the output $Q=1$ is passed to the J input of the slave. The clock's negative transition sets the slave and copies the master.
- The master flip flop toggles on the clock's positive transition when the inputs J and K set to 1. At that time, the slave flip flop toggles on the clock's negative transition.
- The flip flop will be disabled, and Q remains unchanged when both the inputs of the JK flip flop set to 0.

TFlipFlop

This is a much simpler version of the J-K flip flop. Both the J and K inputs are connected together and thus are also called as single input J-K flip flop. When clock pulse is given to the flip flop, the output begins to toggle. Here also the restriction on the pulse width can be eliminated with a master-slave or edge-triggered construction.



(a) Logic diagram



(b) Graphical symbol

Q	T	Q(t+1)
0	0	0
0	1	1
1	0	1
1	1	0

(c) Transition table

Clocked T flip-flop

Shift Register

A group of flip flops which is used to store multiple bits of data and the data is moved from one flip flop to another is known as **Shift Register**. The bits stored in registers shifted when the clock pulse is applied within and inside or outside the registers. To form an n-bit shift register, we have to connect n number of flip flops. So, the number of bits of the binary

number is directly proportional to the number of flip flops. The flip flops are connected in such a way that the first flip flop's output becomes the input of the other flip flop.

A **Shift Register** can shift the bits either to the left or to the right. A **Shift Register**, which shifts the bit to the left, is known as "**Shift left register**", and it shifts the bit to the right, known as "**Right left register**".

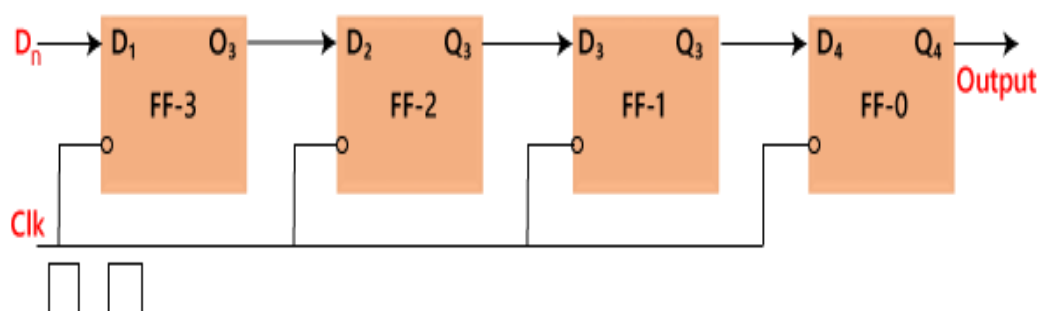
Types of shift Register

- Serial In Serial Out
- Serial In Parallel Out
- Parallel In Serial Out
- Parallel In Parallel Out

Serial IN Serial OUT

In "Serial Input Serial Output", the data is shifted "IN" or "OUT" serially. In SISO, a single bit is shifted at a time in either right or left direction under clock control.

Initially, all the flip-flops are set in "reset" condition i.e. $Y_3 = Y_2 = Y_1 = Y_0 = 0$. If we pass the binary number 1111, the LSB bit of the number is applied first to the Din bit. The D3 input of the third flip flop, i.e., FF-3, is directly connected to the serial data input D3. The output Y_3 is passed to the data input d_2 of the next flip flop. This process remains the same for the remaining flip flops. The block diagram of the "**Serial IN Serial OUT**" is given below.



	Clk	$D_n = Q_3$	$Q_3 = D_2$	$Q_2 = D_1$	$Q_1 = D_0$	Q_0
Initially			0	0	0	0
(1)	↓	1	1	0	0	0
(2)	↓	1	1	1	0	0
(3)	↓	1	1	1	1	0
(4)	↓	1	1	1	1	1

→ Direction of data travel

Operation

When the clock signal application is disabled, the outputs $Y_3 Y_2 Y_1 Y_0 = 0000$. The LSB bit of the number is passed to the data input D_{in} , i.e., D_3 . We will apply the clock, and this time the value of D_3 is 1. The first flip flop, i.e., FF-3, is set, and the word is stored in the register at the first falling edge of the clock. Now, the stored word is 1000.

The next bit of the binary number, i.e., 1, is passed to the data input D_2 . The second flip flop, i.e., FF-2, is set, and the word is stored when the next negative edge of the clock hits. The stored word is changed to 1100.

The next bit of the binary number, i.e., 1, is passed to the data input D_1 , and the clock is applied. The third flip flop, i.e., FF-1, is set, and the word is stored when the negative edge of the clock hits again. The stored word is changed to 1110.

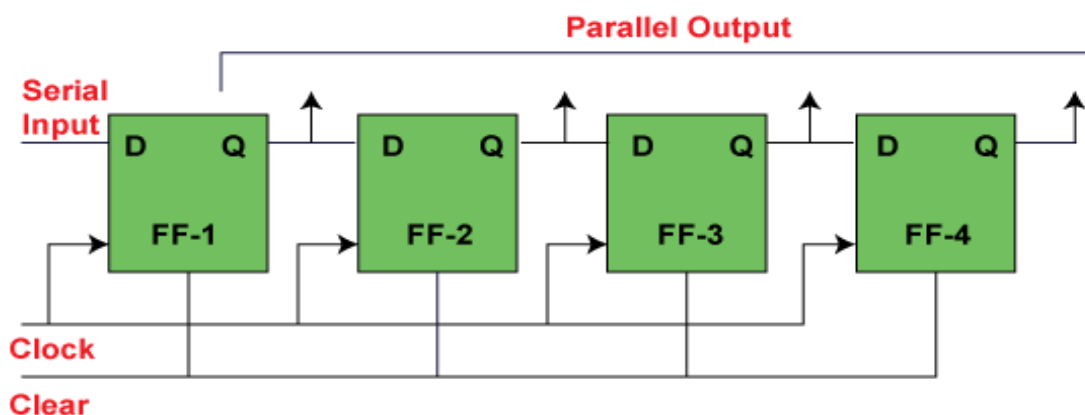
Similarly, the last bit of the binary number, i.e., 1, is passed to the data input D_0 , and the clock is applied. The last flip flop, i.e., FF-0, is set, and the word is stored when the clock's negative edge arrives. The stored word is changed to 1111.

Serial IN Parallel OUT

In the "Serial IN Parallel OUT" shift register, the data is passed serially to the flip flop, and outputs are fetched in a parallel way. The data is passed bit by bit in the register, and the output remains disabled until the data is not passed to the data input. When the data is passed to the register, the outputs are enabled, and the flip flops contain their return value

Below is the block diagram of the 4-bit **serial in the parallel-out** shift register. The circuit having four D flip-flops contains a clear and clock signal to reset these four flip flops.

In **SIPO**, the input of the second flip flop is the output of the first flip flop, and so on. The same clock signal is applied to each flip flop since the flip flops synchronize each other. The parallel outputs are used for communication.

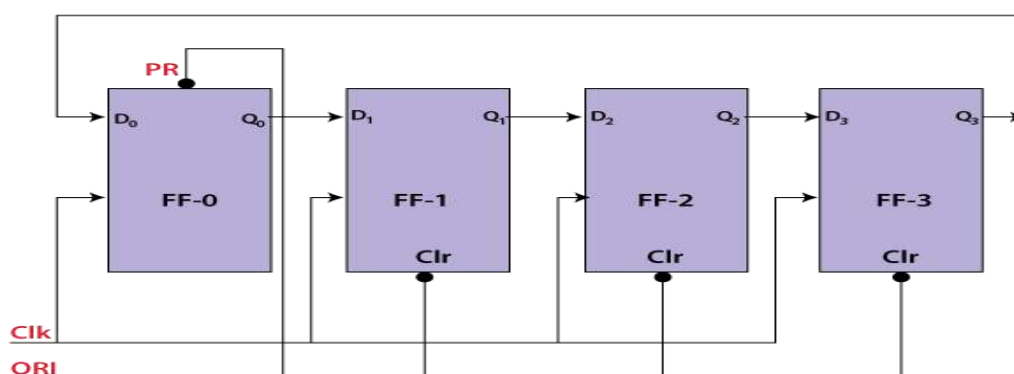


Ring Counter

A **ring counter** is a special type of application of the **Serial IN Serial OUT** Shift register. The only difference between the shift register and the ring counter is that the last flip flop outcome is taken as the output in the shift register. But in the ring counter, this outcome is passed to the first flip flop as an input. All of the remaining things in the ring counter are the same as the shift register.

In the Ring counter

No. of states in Ring counter = No. of flip-flop used



4 **D flip flops** are used. The same clock pulse is passed to the clock input of all the flip flops as a synchronous counter.

The output is 1 when the pre-set set to 0. The output is 0 when the clear set to 0. Both PR and CLR always work in value 0 because they are active low signals.

1. PR = 0, Q = 1
2. CLR = 0, Q = 0

Working

The ORI input is passed to the PR input of the first flip flop, i.e., FF-0, and it is also passed to the clear input of the remaining three flip flops, i.e., FF-1, FF-2, and FF-3. The pre-set input set to 0 for the first flip flop. So, the output of the first flip flop is one, and the outputs of the remaining flip flops are 0. The output of the first flip flop is used to form the ring in the **ring counter** and referred to as **Pre-set 1**.

ORI	Clk	Q ₀	Q ₁	Q ₂	Q ₃
low	X	1	0	0	0
high	low	0	1	0	0
high	low	0	0	1	0
high	low	0	0	0	1
high	low	1	0	0	0

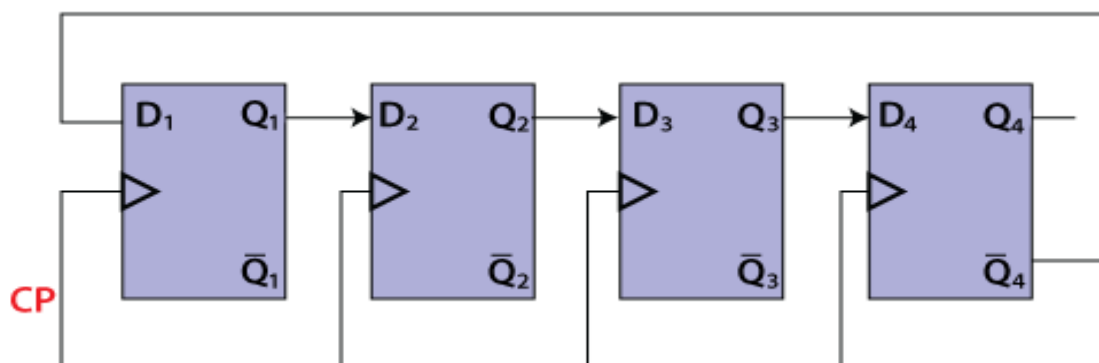
Johnson Counter

The **Johnson counter** is similar to the **Ring counter**. The only difference between the **Johnson counter** and the **ring counter** is that the outcome of the last flip flop is passed to the first flip flop as an input. But in **Johnson counter**, the inverted outcome Q' of the last flip flop is passed as an input. The remaining work of the **Johnson counter** is the same as a **ring counter**. The **Johnson counter** is also referred to as the **Creeping counter**.

In Johnson counter

1. No. of states in Johnson counter = No. of flip-flop used
2. Number of used states = $2n$
3. Number of unused states = $2^n - 2 \cdot n$

Like Ring counter, four D flip flops are used in the 4-bit Johnson counter, and the same clock pulse is passed to all the input of the flip flops.



Truth Table

CP	Q ₁	Q ₂	Q ₃	Q ₄
0	0	0	0	0
1	1	0	0	0
2	1	1	0	0
3	1	1	1	0
4	1	1	1	1
5	0	1	1	1
6	0	0	1	1
7	0	1	1	1

Advantages

- The number of flip flops in the Johnson counter is equal to the number of flip flops in the ring counter, and the Johnson counter counts twice the number of states the ring counter can count.
- The Johnson counter can also be designed by using D or JK flip flop.
- The data is count in a continuous loop in the Johnson ring counter.
- The circuit of the Johnson counter is self-decoding.

Disadvantages

- The Johnson counter is not able to count the states in a binary sequence.
- In the Johnson counter, the unutilized states are greater than the states being utilized.
- The number of flip flops is equal to one half of the number of timing signals.
- It is possible to design the Johnson counter for any number of timing sequences.

Counters

A special type of sequential circuit used to count the pulse is known as a counter, or a collection of flip flops where the clock signal is applied is known as counters.

The counter is one of the widest applications of the flip flop. Based on the clock pulse, the output of the counter contains a predefined state. The number of the pulse can be counted using the output of the counter.

There are the following types of counters:

- Asynchronous Counters
- Synchronous Counters

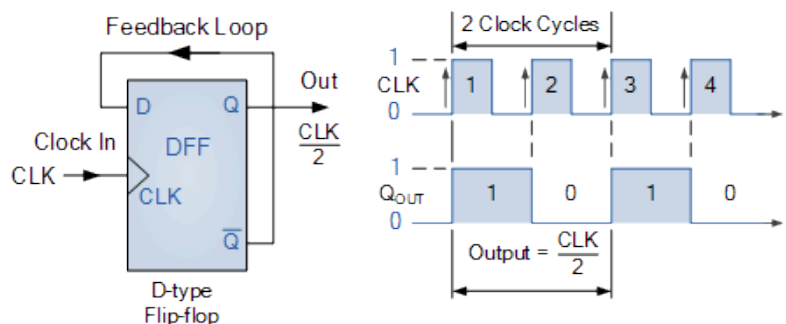
Asynchronous Counters

If the flip-flops do not receive the same clock signal, then that counter is called as **Asynchronous counter**. The output of system clock is applied as clock signal only to first flip-flop. The remaining flip-flops receive the clock signal from output of its previous stage flip-flop. Hence, the outputs of all flip-flops do not change at the same time.

Modulus Counters, or simply *MOD counters*, are defined based on the number of states that the counter will sequence through before returning back to its original value. For example, a 2-bit counter that counts from 00_2 to 11_2 in binary, that is 0 to 3 in decimal, has a modulus value of 4 ($00 \rightarrow 01 \rightarrow 10 \rightarrow 11$, and return back to 00) so would therefore be called a modulo-4, or mod-4, counter. Note also that it has taken four clock pulses to get from 00 to

11. Therefore, a “Mod-N” counter will require “N” number of flip-flops connected together to count a single data bit while providing 2^n different output states, (n is the number of bits).

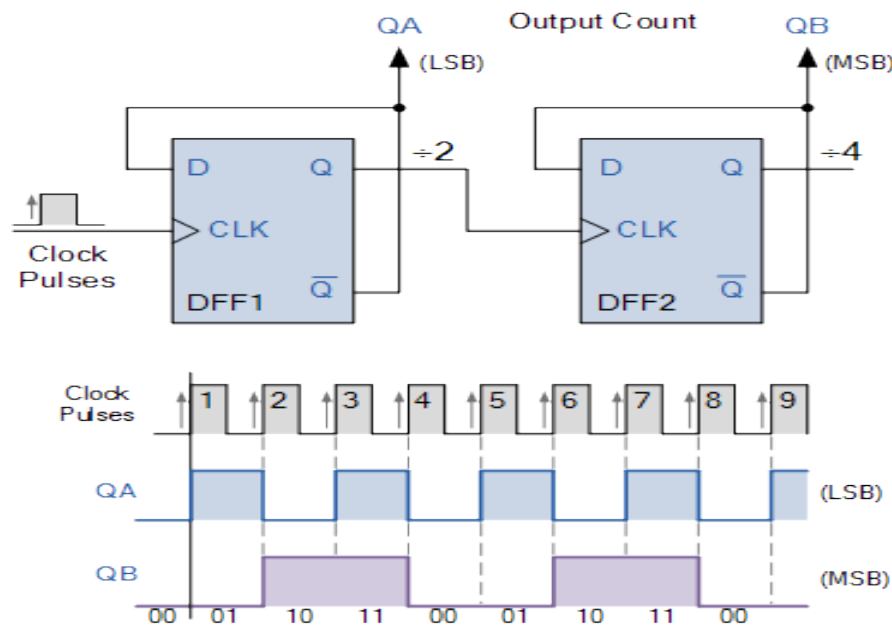
MOD 2 Counter



We will use the D-type flip-flop, (DFF) also known as a Data Latch, because a single data input and external clock signal are used, and is also positive edge triggered. The timing diagrams show that the “Q” output waveform has a frequency exactly one-half that of the clock input, thus the flip-flop acts as a frequency divider. If we added another D-type flip-flop so that the output at “Q” was the input to the second DFF, then the output signal from this second DFF would be one-quarter of the clock input frequency, and so on. So for an “n” number of flip-flops, the output frequency is divided by 2^n , in steps of 2.

MOD-4 Counter

If a single flip-flop can be considered as a modulo-2 or MOD-2 counter, then adding a second flip-flop would give us a MOD-4 counter allowing it to count in four discrete steps. The overall effect would be to divide the original clock input signal by four. Then the binary sequence for this 2-bit MOD-4 counter would be: 00, 01, 10, and 11

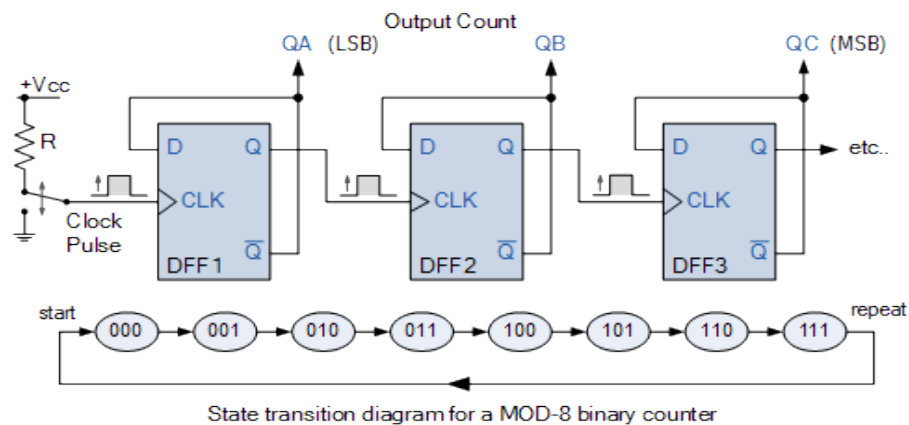


When $QA = 0$ and $QB = 0$, the count is 00. After the application of the clock pulse, the values become $QA = 1$, $QB = 0$, giving a count of 01. After the arrival of the next clock pulse, the

values change and become $Q_A = 0$, $Q_B = 1$, giving a count of 10. Finally the values become $Q_A = 1$, $Q_B = 1$, giving a count of 11. The application of the next clock pulse causes the count to return back to 00, and thereafter it counts continuously up in a binary sequence of: 00, 01, 10, 11, 00, 01 ...etc.

MOD-8 Counter

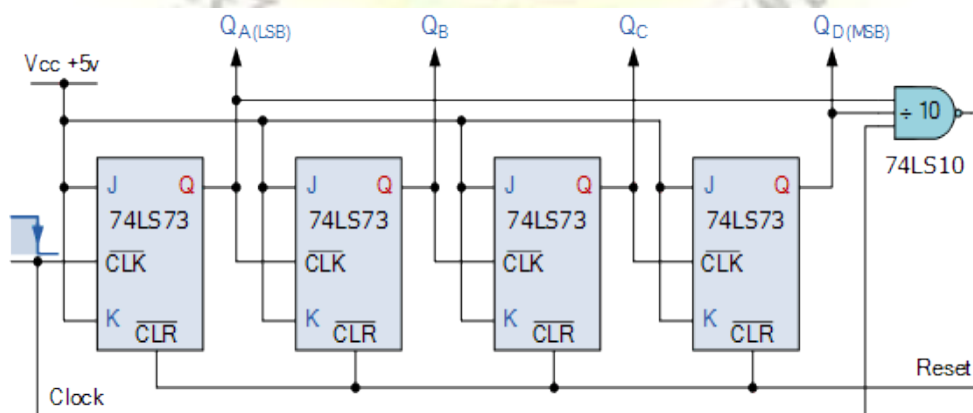
We could easily add another flip-flop onto the end of a MOD-4 counter to produce a MOD-8 counter giving us a 2^3 binary sequence of counting from 000 up to 111, before resetting back to 000.



Modulus 10 Counter

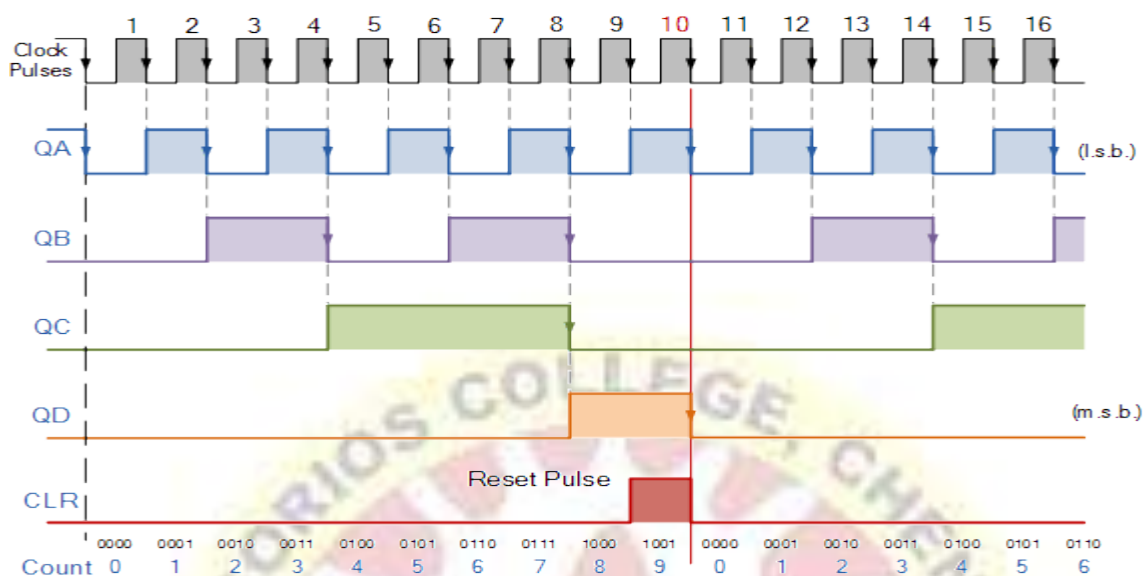
A good example of a modulo-m counter circuit which uses external combinational circuits to produce a counter with a modulus of 10 is the Decade Counter.

The decade counter has four outputs producing a 4-bit binary number and by using external AND and OR gates we can detect the occurrence of the 9th counting state to reset the counter back to zero.



As with other mod counters, it receives an input clock pulse, one by one, and counts up from 0 to 9 repeatedly. Once it reaches the count 9 (1001 in binary), the counter goes back to 0000

instead of continuing on to 1010. Decade counters are useful for interfacing to digital displays.



UNIT V

MEMORY DEVICES

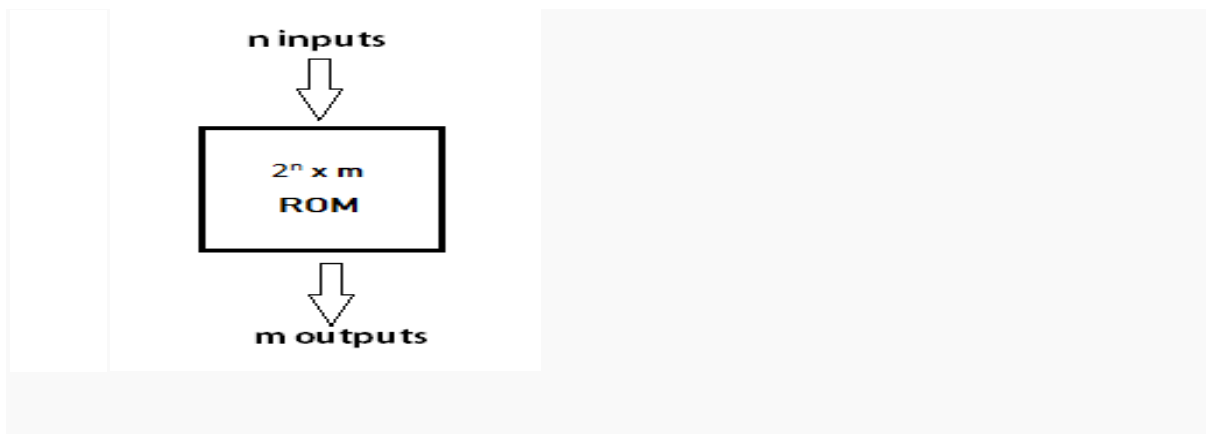
ROM Organization

ROM, which stands for read only memory, is a memory device or storage medium that stores information permanently. It is also the primary memory unit of a computer along with the random access memory (RAM). It is called read only memory as we can only read the programs and data stored on it but cannot write on it. It is restricted to reading words that are permanently stored within the unit.

The manufacturer of ROM fills the programs into the ROM at the time of manufacturing the ROM. After this, the content of the ROM can't be altered, which means you can't reprogram, rewrite, or erase its content later. However, there are some types of ROM where you can modify the data.

ROM contains special internal electronic fuses that can be programmed for a specific interconnection pattern (information). The binary information stored in the chip is specified by the designer and then embedded in the unit at the time of manufacturing to form the required interconnection pattern (information). Once the pattern (information) is established, it stays within the unit even when the power is turned off. So, it is a non-volatile memory as it holds the information even when the power is turned off, or you shut down your computer.

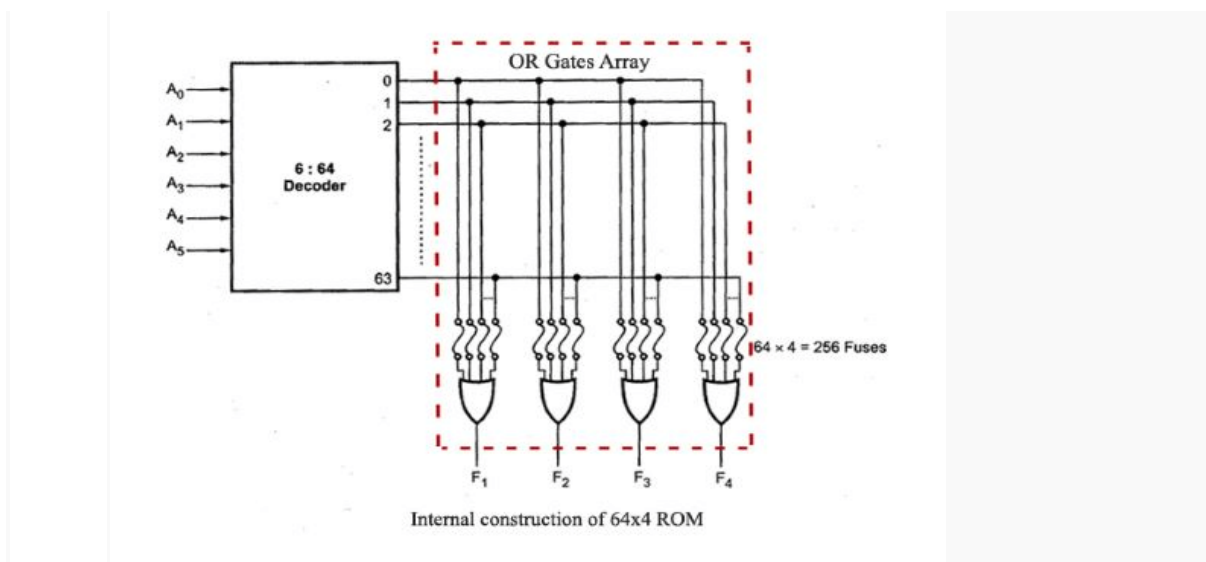
Block Diagram of ROM:



The block of ROM has 'n' input lines and 'm' output lines. Each bit combination of the input variables is known as an address. Each bit combination that comes out through output lines is called a word. The number of bits per word is equal to the number of output lines, m.

The address of a binary number refers to one of the addresses of n variables. So, the number of possible addresses with 'n' input variables is 2^n . An output word has a unique address, and as there are 2^n distinct addresses in a ROM, there are 2^n separate words in the ROM. The words on the output lines at a given time depends on the address value applied to the input lines.

Internal Structure of ROM:



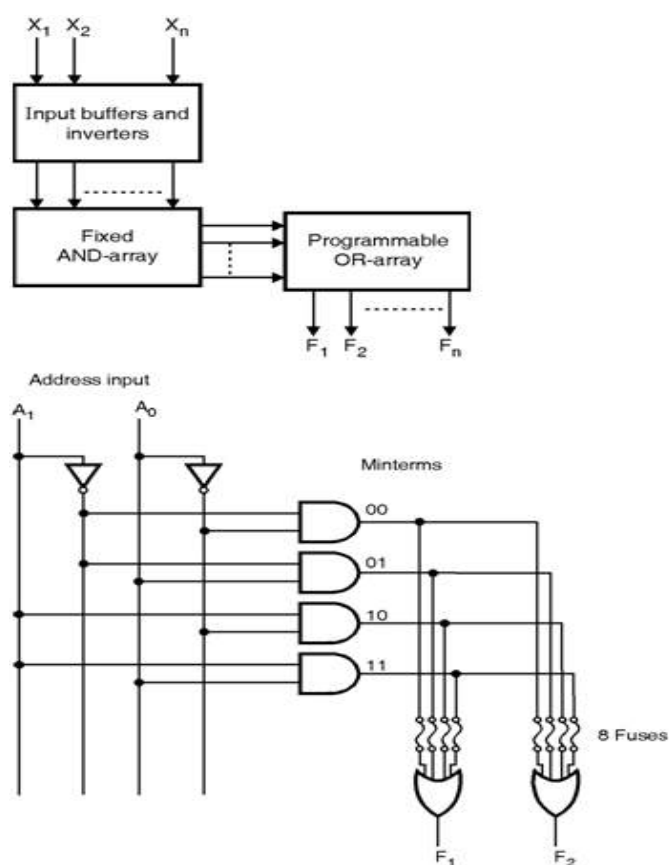
The internal structure comprises two basic components: decoder and OR gates. A decoder is a circuit that decodes an encoded form (such as binary coded decimal, BCD) to a decimal form. So, the input is in binary form, and the output is its decimal equivalent. All the OR gates present in the ROM will have outputs of the decoder as their output.

This Read Only Memory consists of 64 words of 4 bits each. So, there would be four output lines, and one of the 64 words available on the output lines is determined from the six input

lines as we have only six inputs because in this ROM we have $2^6 = 64$, so we can specify 64 addresses or minterms.

Programmable Read only Memory (PROM)

A programmable read only memory is a device that includes both the AND plane and OR-plane within a single IC package. Out of these two arrays AND plane is fixed and OR plane is programmable. Figure below shows the block diagram view of PROM. In the PROM the AND array will act as a decoder which will decode the address lines. The gate level structure of PROM is also shown in figure. In mask PROM it is necessary to specify the bit pattern to be stored according to the requirements of the circuits. Since PROMs are used in logic designs these are also referred as PLDs.

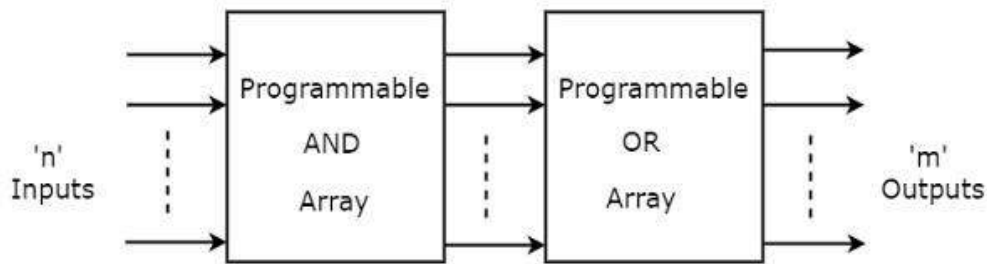


Advantages of ROM

- 1) As no minimization of logic circuits is needed the circuits can be designed easily.
- 2) It is possible to modify the circuit faster.
- 3) These are high speed as compared to discrete SSI/MSI circuits.
- 4) The Cost is lower.

Programmable Logic Array PLA

PLA is a programmable logic device that has both Programmable AND array & Programmable OR array. Hence, it is the most flexible PLD.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are also programmable. So, we can program any number of required product terms, since all the outputs of AND gates are applied as inputs to each OR gate. Therefore, the outputs of PAL will be in the form of **sum of products form**.

Example

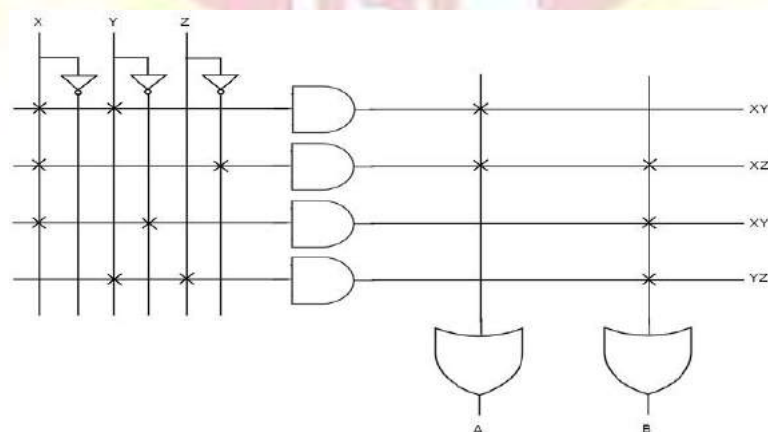
Let us implement the following **Boolean functions** using PLA.

$$A = XY + XZ'$$

$$B = XY' + YZ + XZ'$$

The given two functions are in sum of products form. The number of product terms present in the given Boolean functions A & B are two and three respectively. One product term, $Z'XZ'X$ is common in each function.

So, we require four programmable AND gates & two programmable OR gates for producing those two functions. The corresponding **PLA** is shown in the following figure.

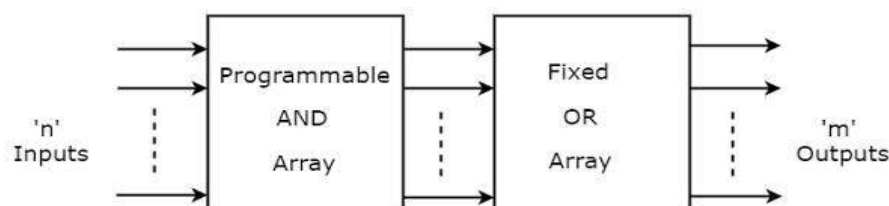


The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, X', Y, Y', Z & Z', are available at the inputs of

each AND gate. So, program only the required literals in order to generate one product term by each AND gate.

Programmable Array Logic PAL

PAL is a programmable logic device that has Programmable AND array & fixed OR array. The advantage of PAL is that we can generate only the required product terms of Boolean function instead of generating all the min terms by using programmable AND gates. The **block diagram** of PAL is shown in the following figure.



Here, the inputs of AND gates are programmable. That means each AND gate has both normal and complemented inputs of variables. So, based on the requirement, we can program any of those inputs. So, we can generate only the required **product terms** by using these AND gates.

Here, the inputs of OR gates are not of programmable type. So, the number of inputs to each OR gate will be of fixed type. Hence, apply those required product terms to each OR gate as inputs. Therefore, the outputs of PAL will be in the form of **sum of products form**.

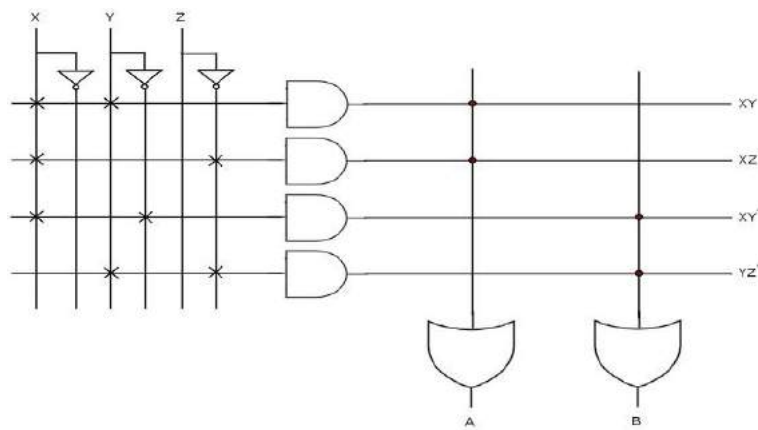
Example

Let us implement the following **Boolean functions** using PAL.

$$A = XY + XZ'$$

$$A = XY' + YZ'$$

The given two functions are in sum of products form. There are two product terms present in each Boolean function. So, we require four programmable AND gates & two fixed OR gates for producing those two functions. The corresponding **PAL** is shown in the following figure.



The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, X', Y, Y', Z & Z' , are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate. The symbol 'X' is used for programmable connections.

