tutorialspoint.com

Digital Circuits - Quick Guide

160-203 minutes

If base or radix of a number system is ‘r’, then the numbers present in that number system are ranging from zero to r-1. The total numbers present in that number system is ‘r’. So, we will get various number systems, by choosing the values of radix as greater than or equal to two.

In this chapter, let us discuss about the **popular number systems** and how to represent a number in the respective number system. The following number systems are the most commonly used.

- Decimal Number system
- Binary Number system
- Octal Number system
- Hexadecimal Number system

Decimal Number System

The **base** or radix of Decimal number system is **10**. So, the numbers ranging from 0 to 9 are used in this number system. The part of the number that lies to the left of the **decimal point** is known as integer part. Similarly, the part of the number that lies to the right of the decimal point is known as fractional part.

In this number system, the successive positions to the left of the decimal point having weights of 10^0, 10^1, 10^2, 10^3 and so on. Similarly, the successive positions to the right of the decimal point having weights of 10^{-1}, 10^{-2}, 10^{-3} and so on. That means, each position has specific weight, which is **power of base 10**

Example

Consider the **decimal number 1358.246**. Integer part of this number is 1358 and fractional part of this number is 0.246. The digits 8, 5, 3 and 1 have weights of 100, 101, 10^2 and 10^3 respectively. Similarly, the digits 2, 4 and 6 have weights of 10^{-1}, 10^{-2} and 10^{-3} respectively.

Mathematically, we can write it as

$1358.246 = (1 \times 10^3) + (3 \times 10^2) + (5 \times 10^1) + (8 \times 10^0) + (2 \times 10^{-1}) +$

$(4 \times 10^{-2}) + (6 \times 10^{-3})$

After simplifying the right hand side terms, we will get the decimal number, which is on left hand side.

Binary Number System

All digital circuits and systems use this binary number system. The **base** or radix of this number system is **2**. So, the numbers 0 and 1 are used in this number system.

The part of the number, which lies to the left of the **binary point** is known as integer part. Similarly, the part of the number, which lies to the right of the binary point is known as fractional part.

In this number system, the successive positions to the left of the binary point having weights of 2^0, 2^1, 2^2, 2^3 and so on. Similarly, the successive positions to the right of the binary point having weights of 2^{-1}, 2^{-2}, 2^{-3} and so on. That means, each position has specific weight, which is **power of base 2**.

Example

Consider the **binary number 1101.011**. Integer part of this number is 1101 and fractional part of this number is 0.011. The digits 1, 0, 1 and 1 of integer part have weights of 2^0, 2^1, 2^2, 2^3 respectively. Similarly, the digits 0, 1 and 1 of fractional part have weights of 2^{-1}, 2^{-2}, 2^{-3} respectively.

Mathematically, we can write it as

$1101.011 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) +$

$(1 \times 2^{-2}) + (1 \times 2^{-3})$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of binary number on left hand side.

Octal Number System

The **base** or radix of octal number system is **8**. So, the numbers ranging from 0 to 7 are used in this number system. The part of the number that lies to the left of the **octal point** is known as integer part. Similarly, the part of the number that lies to the right of the octal point is known as fractional part.

In this number system, the successive positions to the left of the octal point having weights of 8^0, 8^1, 8^2, 8^3 and so on. Similarly, the successive positions to the right of the octal point having weights of 8^{-1}, 8^{-2}, 8^{-3} and so on. That means, each position has specific weight, which is **power of base 8**.

Example

Consider the **octal number 1457.236**. Integer part of this number is 1457 and fractional part of this number is 0.236. The digits 7, 5, 4 and 1 have weights of

8^0, 8^1, 8^2 and 8^3 respectively. Similarly, the digits 2, 3 and 6 have weights of 8^{-1}, 8^{-2}, 8^{-3} respectively.

Mathematically, we can write it as

$1457.236 = (1 \times 8^3) + (4 \times 8^2) + (5 \times 8^1) + (7 \times 8^0) + (2 \times 8^{-1}) +$

$(3 \times 8^{-2}) + (6 \times 8^{-3})$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of octal number on left hand side.

Hexadecimal Number System

The **base** or radix of Hexa-decimal number system is **16**. So, the numbers ranging from 0 to 9 and the letters from A to F are used in this number system. The decimal equivalent of Hexa-decimal digits from A to F are 10 to 15.

The part of the number, which lies to the left of the **hexadecimal point** is known as integer part. Similarly, the part of the number, which lies to the right of the Hexa-decimal point is known as fractional part.

In this number system, the successive positions to the left of the Hexa-decimal point having weights of 16^0, 16^1, 16^2, 16^3 and so on. Similarly, the successive positions to the right of the Hexa-decimal point having weights of 16^{-1}, 16^{-2}, 16^{-3} and so on. That means, each position has specific weight, which is **power of base 16**.

Example

Consider the **Hexa-decimal number 1A05.2C4**. Integer part of this number is 1A05 and fractional part of this number is 0.2C4. The digits 5, 0, A and 1 have weights of 16^0, 16^1, 16^2 and 16^3 respectively. Similarly, the digits 2, C and 4 have weights of 16^{-1}, 16^{-2} and 16^{-3} respectively.

Mathematically, we can write it as

$1A05.2C4 = (1 \times 16^3) + (10 \times 16^2) + (0 \times 16^1) + (5 \times 16^0) + (2 \times 16^{-1}) +$

$(12 \times 16^{-2}) + (4 \times 16^{-3})$

After simplifying the right hand side terms, we will get a decimal number, which is an equivalent of Hexa-decimal number on left hand side.

In previous chapter, we have seen the four prominent number systems. In this chapter, let us convert the numbers from one number system to the other in order to find the equivalent value.

Decimal Number to other Bases Conversion

If the decimal number contains both integer part and fractional part, then convert both the parts of decimal number into other base individually. Follow these steps for converting the decimal number into its equivalent number of any base ‘r’.

- Do **division** of integer part of decimal number and **successive quotients** with base ‘r’ and note down the remainders till the quotient is zero. Consider the remainders in reverse order to get the integer part of equivalent number of base ‘r’. That means, first and last remainders denote the least significant digit and most significant digit respectively.
- Do **multiplication** of fractional part of decimal number and **successive fractions** with base ‘r’ and note down the carry till the result is zero or the desired number of equivalent digits is obtained. Consider the normal sequence of carry in order to get the fractional part of equivalent number of base ‘r’.

Decimal to Binary Conversion

The following two types of operations take place, while converting decimal number into its equivalent binary number.

- Division of integer part and successive quotients with base 2.
- Multiplication of fractional part and successive fractions with base 2.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and fractional part is 0.25.

Step 1 – Division of 58 and successive quotients with base 2.

Operation	Quotient	Remainder
58/2	29	**0 LSB**
29/2	14	**1**
14/2	7	**0**
7/2	3	**1**
3/2	1	**1**
1/2	0	**1MSB**

$\Rightarrow 58_{10} = 111010_2$

Therefore, the **integer part** of equivalent binary number is **111010**.

Step 2 – Multiplication of 0.25 and successive fractions with base 2.

Operation	Result	Carry
0.25 x 2	0.5	0
0.5 x 2	1.0	1
-	0.0	-

$\Rightarrow .25_{10} = .01_2$

Therefore, the **fractional part** of equivalent binary number is **.01**

⇒$58.25_{10} = 111010.01_2$

Therefore, the **binary equivalent** of decimal number 58.25 is 111010.01.

Decimal to Octal Conversion

The following two types of operations take place, while converting decimal number into its equivalent octal number.

- Division of integer part and successive quotients with base 8.
- Multiplication of fractional part and successive fractions with base 8.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and fractional part is 0.25.

Step 1 – Division of 58 and successive quotients with base 8.

Operation	Quotient	Remainder
58/8	7	**2**
7/8	0	**7**

⇒$58_{10} = 72_8$

Therefore, the **integer part** of equivalent octal number is **72**.

Step 2 – Multiplication of 0.25 and successive fractions with base 8.

Operation	Result	Carry
0.25 x 8	2.00	2
-	0.00	-

⇒ $.25_{10} = .2_8$

Therefore, the **fractional part** of equivalent octal number is .2

⇒ $58.25_{10} = 72.2_8$

Therefore, the **octal equivalent** of decimal number 58.25 is 72.2.

Decimal to Hexa-Decimal Conversion

The following two types of operations take place, while converting decimal number into its equivalent hexa-decimal number.

- Division of integer part and successive quotients with base 16.
- Multiplication of fractional part and successive fractions with base 16.

Example

Consider the **decimal number 58.25**. Here, the integer part is 58 and decimal part is 0.25.

Step 1 – Division of 58 and successive quotients with base 16.

Operation	Quotient	Remainder
58/16	3	10=A
3/16	0	3

⇒ $58_{10} = 3A_{16}$

Therefore, the **integer part** of equivalent Hexa-decimal number is 3A.

Step 2 – Multiplication of 0.25 and successive fractions with base 16.

Operation	Result	Carry
0.25 x 16	4.00	4
-	0.00	-

⇒$.25_{10} = .4_{16}$

Therefore, the **fractional part** of equivalent Hexa-decimal number is .4.

⇒$\mathbf{58.25_{10} = 3A.4_{16}}$

Therefore, the **Hexa-decimal equivalent** of decimal number 58.25 is 3A.4.

Binary Number to other Bases Conversion

The process of converting a number from binary to decimal is different to the process of converting a binary number to other bases. Now, let us discuss about the conversion of a binary number to decimal, octal and Hexa-decimal number systems one by one.

Binary to Decimal Conversion

For converting a binary number into its equivalent decimal number, first multiply the bits of binary number with the respective positional weights and then add all those products.

Example

Consider the **binary number 1101.11**.

Mathematically, we can write it as

$1101.11_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (1 \times 2^{-1}) +$

(1×2^{-2})

⇒ $1101.11_2 = 8 + 4 + 0 + 1 + 0.5 + 0.25 = 13.75$

⇒ $1101.11_2 = 13.75_{10}$

Therefore, the **decimal equivalent** of binary number 1101.11 is 13.75.

Binary to Octal Conversion

We know that the bases of binary and octal number systems are 2 and 8 respectively. Three bits of binary number is equivalent to one octal digit, since $2^3 = 8$.

Follow these two steps for converting a binary number into its equivalent octal number.

- Start from the binary point and make the groups of 3 bits on both sides of binary point. If one or two bits are less while making the group of 3 bits, then include required number of zeros on extreme sides.
- Write the octal digits corresponding to each group of 3 bits.

Example

Consider the **binary number 101110.01101**.

Step 1 – Make the groups of 3 bits on both sides of binary point.

101 110.011 01

Here, on right side of binary point, the last group is having only 2 bits. So, include one zero on extreme side in order to make it as group of 3 bits.

⇒ 101 110.011 010

Step 2 – Write the octal digits corresponding to each group of 3 bits.

⇒ $101110.011010_2 = 56.32_8$

Therefore, the **octal equivalent** of binary number 101110.01101 is 56.32.

Binary to Hexa-Decimal Conversion

We know that the bases of binary and Hexa-decimal number systems are 2 and 16 respectively. Four bits of binary number is equivalent to one Hexa-decimal digit, since $2^4 = 16$.

Follow these two steps for converting a binary number into its equivalent Hexa-decimal number.

- Start from the binary point and make the groups of 4 bits on both sides of binary point. If some bits are less while making the group of 4 bits, then include required number of zeros on extreme sides.
- Write the Hexa-decimal digits corresponding to each group of 4 bits.

Example

Consider the **binary number 101110.01101**

Step 1 – Make the groups of 4 bits on both sides of binary point.

10 1110.0110 1

Here, the first group is having only 2 bits. So, include two zeros on extreme side in order to make it as group of 4 bits. Similarly, include three zeros on extreme side in order to make the last group also as group of 4 bits.

⇒ 0010 1110.0110 1000

Step 2 – Write the Hexa-decimal digits corresponding to each group of 4 bits.

⇒ $00101110.01101000_2 = 2E.68_{16}$

Therefore, the **Hexa-decimal equivalent** of binary number 101110.01101 is 2E.68.

Octal Number to other Bases Conversion

The process of converting a number from octal to decimal is different to the process of converting an octal number to other bases. Now, let us discuss about the conversion of an octal number to decimal, binary and Hexa-decimal number systems one by one.

Octal to Decimal Conversion

For converting an octal number into its equivalent decimal number, first multiply the digits of octal number with the respective positional weights and then add all those products.

Example

Consider the **octal number 145.23**.

Mathematically, we can write it as

$145.23_8 = (1 \times 8^2) + (4 \times 8^1) + (5 \times 8^0) + (2 \times 8^{-1}) + (3 \times 8^{-2})$

⇒ $145.23_8 = 64 + 32 + 5 + 0.25 + 0.05 = 101.3$

⇒ $145.23_8 = 101.3_{10}$

Therefore, the **decimal equivalent** of octal number 145.23 is 101.3.

Octal to Binary Conversion

The process of converting an octal number to an equivalent binary number is just opposite to that of binary to octal conversion. By representing each octal digit with 3 bits, we will get the equivalent binary number.

Example

Consider the **octal number 145.23**.

Represent each octal digit with 3 bits.

$145.23_8 = 001100101.010011_2$

The value doesn't change by removing the zeros, which are on the extreme side.

⇒ $145.23_8 = 1100101.010011_2$

Therefore, the **binary equivalent** of octal number 145.23 is 1100101.010011.

Octal to Hexa-Decimal Conversion

Follow these two steps for converting an octal number into its equivalent Hexa-decimal number.

- Convert octal number into its equivalent binary number.
- Convert the above binary number into its equivalent Hexa-decimal number.

Example

Consider the **octal number 145.23**

In previous example, we got the binary equivalent of octal number 145.23 as 1100101.010011.

By following the procedure of binary to Hexa-decimal conversion, we will get

1100101.010011_2 = 65.4C16

$\Rightarrow 145.23_8 = 65.4C_{16}$

Therefore, the **Hexa-decimal equivalent** of octal number 145.23 is 65.4*C*.

Hexa-Decimal Number to other Bases Conversion

The process of converting a number from Hexa-decimal to decimal is different to the process of converting Hexa-decimal number into other bases. Now, let us discuss about the conversion of Hexa-decimal number to decimal, binary and octal number systems one by one.

Hexa-Decimal to Decimal Conversion

For converting Hexa-decimal number into its equivalent decimal number, first multiply the digits of Hexa-decimal number with the respective positional weights and then add all those products.

Example

Consider the **Hexa-decimal number 1A5.2**

Mathematically, we can write it as

$1A5.2_{16} = (1 \times 16^2) + (10 \times 16^1) + (5 \times 16^0) + (2 \times 16^{-1})$

$\Rightarrow 1A5.2_{16} = 256 + 160 + 5 + 0.125 = 421.125$

$\Rightarrow 1A5.2_{16} = 421.125_{10}$

Therefore, the **decimal equivalent** of Hexa-decimal number 1A5.2 is 421.125.

Hexa-Decimal to Binary Conversion

The process of converting Hexa-decimal number into its equivalent binary number is just opposite to that of binary to Hexa-decimal conversion. By representing each Hexa-decimal digit with 4 bits, we will get the equivalent binary number.

Example

Consider the **Hexa-decimal number 65.4C**

Represent each Hexa-decimal digit with 4 bits.

$65.4C_6 = 01100101.01001100_2$

The value doesn't change by removing the zeros, which are at two extreme sides.

$\Rightarrow 65.4C_{16} = 1100101.010011_2$

Therefore, the **binary equivalent** of Hexa-decimal number 65.4C is 1100101.010011.

Hexa-Decimal to Octal Conversion

Follow these two steps for converting Hexa-decimal number into its equivalent octal number.

- Convert Hexa-decimal number into its equivalent binary number.
- Convert the above binary number into its equivalent octal number.

Example

Consider the **Hexa-decimal number 65.4C**

In previous example, we got the binary equivalent of Hexa-decimal number 65.4C as 1100101.010011.

By following the procedure of binary to octal conversion, we will get

$1100101.010011_2 = 145.23_8$

$\Rightarrow 65.4C_{16} = 145.23_{\mathbf{8}}$

Therefore, the **octal equivalent** of Hexa-decimal number $65.4C$ is 145.23.

We can make the binary numbers into the following two groups – **Unsigned numbers** and **Signed numbers**.

Unsigned Numbers

Unsigned numbers contain only magnitude of the number. They don't have any sign. That means all unsigned binary numbers are positive. As in decimal number system, the placing of positive sign in front of the number is optional for representing positive numbers. Therefore, all positive numbers including zero can be treated as unsigned numbers if positive sign is not assigned in front of the number.

Signed Numbers

Signed numbers contain both sign and magnitude of the number. Generally, the sign is placed in front of number. So, we have to consider the positive sign for positive numbers and negative sign for negative numbers. Therefore, all numbers can be treated as signed numbers if the corresponding sign is assigned in front of the number.

If sign bit is zero, which indicates the binary number is positive. Similarly, if sign bit is one, which indicates the binary number is negative.

Representation of Un-Signed Binary Numbers

The bits present in the un-signed binary number holds the **magnitude** of a number. That means, if the un-signed binary number contains **'N'** bits, then all **N** bits represent the magnitude of the number, since it doesn't have any sign bit.

Example

Consider the **decimal number 108**. The binary equivalent of this number is **1101100**. This is the representation of unsigned binary number.

$108_{10} = 1101100_2$

It is having 7 bits. These 7 bits represent the magnitude of the number 108.

Representation of Signed Binary Numbers

The Most Significant Bit MSB of signed binary numbers is used to indicate the sign of the numbers. Hence, it is also called as **sign bit**. The positive sign is represented by placing '0' in the sign bit. Similarly, the negative sign is represented by placing '1' in the sign bit.

If the signed binary number contains 'N' bits, then N−1 bits only represent the magnitude of the number since one bit MSB is reserved for representing sign of the number.

There are three **types of representations** for signed binary numbers

- Sign-Magnitude form
- 1's complement form
- 2's complement form

Representation of a positive number in all these 3 forms is same. But, only the representation of negative number will differ in each form.

Example

Consider the **positive decimal number +108**. The binary equivalent of magnitude of this number is 1101100. These 7 bits represent the magnitude of the number 108. Since it is positive number, consider the sign bit as zero, which is placed on left most side of magnitude.

$+108_{10} = 01101100_2$

Therefore, the **signed binary representation** of positive decimal number +108 is $\mathbf{01101100}$. So, the same representation is valid in sign-magnitude form, 1's complement form and 2's complement form for positive decimal number +108.

Sign-Magnitude form

In sign-magnitude form, the MSB is used for representing **sign** of the number and the remaining bits represent the **magnitude** of the number. So, just include sign bit at the left most side of unsigned binary number. This representation is similar to the signed decimal numbers representation.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the unsigned binary representation of 108 is 1101100. It is having 7 bits. All these bits represent the magnitude.

Since the given number is negative, consider the sign bit as one, which is placed on left most side of magnitude.

$-108_{10} = 11101100_2$

Therefore, the sign-magnitude representation of -108 is **11101100**.

1’s complement form

The 1’s complement of a number is obtained by **complementing all the bits** of signed binary number. So, 1’s complement of positive number gives a negative number. Similarly, 1’s complement of negative number gives a positive number.

That means, if you perform two times 1’s complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**. The magnitude of this number is 108. We know the signed binary representation of 108 is 01101100.

It is having 8 bits. The MSB of this number is zero, which indicates positive number. Complement of zero is one and vice-versa. So, replace zeros by ones and ones by zeros in order to get the negative number.

$-108_{10} = 10010011_2$

Therefore, the **1’s complement of 108_{10}** is **10010011_2**.

2’s complement form

The 2’s complement of a binary number is obtained by **adding one to the 1’s complement** of signed binary number. So, 2’s complement of positive number gives a negative number. Similarly, 2’s complement of negative number gives a positive number.

That means, if you perform two times 2’s complement of a binary number including sign bit, then you will get the original signed binary number.

Example

Consider the **negative decimal number -108**.

We know the 1’s complement of $(\mathbf{108})_{\mathbf{10}}$ is $(\mathbf{10010011})_{\mathbf{2}}$

2’s compliment of 108_{10} = 1’s compliment of 108_{10} + 1.

= 10010011 + 1

= 10010100

Therefore, the **2’s complement of 108_{10}** is **10010100_2**.

In this chapter, let us discuss about the basic arithmetic operations, which can be performed on any two signed binary numbers using 2’s complement method. The **basic arithmetic operations** are addition and subtraction.

Addition of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2’s complement form. We can perform the **addition** of these two numbers, which is similar to the addition of two unsigned binary numbers. But, if the resultant sum contains carry out from sign bit, then discard ignore it in order to get the correct value.

If resultant sum is positive, you can find the magnitude of it directly. But, if the resultant sum is negative, then take 2’s complement of it in order to get the magnitude.

Example 1

Let us perform the **addition** of two decimal numbers **+7 and +4** using 2’s complement method.

The **2’s complement** representations of +7 and +4 with 5 bits each are shown below.

$+7_{10} = 00111_2$

$+4_{10} = 00100_2$

The addition of these two numbers is

$+7_{10} ++4_{10} = 00111_2+00100_2$

$\Rightarrow +7_{10} ++4_{10} = 01011_2$.

The resultant sum contains 5 bits. So, there is no carry out from sign bit. The sign bit ‘0’ indicates that the resultant sum is **positive**. So, the magnitude of sum is 11 in decimal number system. Therefore, addition of two positive numbers will give another positive number.

Example 2

Let us perform the **addition** of two decimal numbers **-7** and **-4** using 2’s complement method.

The **2’s complement** representation of -7 and -4 with 5 bits each are shown below.

$-7_{10} = 11001_2$

$-4_{10} = 11100_2$

The addition of these two numbers is

$-7_{10} + -4_{10} = 11001_2 + 11100_2$

$\Rightarrow -7_{10} + -4_{10} = 110101_2$.

The resultant sum contains 6 bits. In this case, carry is obtained from sign bit. So, we can remove it

Resultant sum after removing carry is $-7_{10} + -4_{10} = \mathbf{10101_2}$.

The sign bit '1' indicates that the resultant sum is **negative**. So, by taking 2's complement of it we will get the magnitude of resultant sum as 11 in decimal number system. Therefore, addition of two negative numbers will give another negative number.

Subtraction of two Signed Binary Numbers

Consider the two signed binary numbers A & B, which are represented in 2's complement form. We know that 2's complement of positive number gives a negative number. So, whenever we have to subtract a number B from number A, then take 2's complement of B and add it to A. So, **mathematically** we can write it as

A - B = A + $2'scomplementofB$

Similarly, if we have to subtract the number A from number B, then take 2's complement of A and add it to B. So, **mathematically** we can write it as

B - A = B + $2'scomplementofA$

So, the subtraction of two signed binary numbers is similar to the addition of two signed binary numbers. But, we have to take 2's complement of the number, which is supposed to be subtracted. This is the **advantage** of 2's complement technique. Follow, the same rules of addition of two signed binary numbers.

Example 3

Let us perform the **subtraction** of two decimal numbers **+7 and +4** using 2's complement method.

The subtraction of these two numbers is

$+7_{10} - +4_{10} = +7_{10} + -4_{10}$.

The **2's complement** representation of +7 and -4 with 5 bits each are shown below.

$+7_{10} = 00111_2$

$+4_{10} = 11100_2$

⇒$+7_{10} + +4_{10} = 00111_2 + 11100_2 = 00011_2$

Here, the carry obtained from sign bit. So, we can remove it. The resultant sum after removing carry is

$+7_{10} + +4_{10} =$ **00011_2**

The sign bit ‘0’ indicates that the resultant sum is **positive**. So, the magnitude of it is 3 in decimal number system. Therefore, subtraction of two decimal numbers +7 and +4 is +3.

Example 4

Let us perform the **subtraction of** two decimal numbers **+4** and **+7** using 2’s complement method.

The subtraction of these two numbers is

$+4_{10} - +7_{10} = +4_{10} + -7_{10}$.

The **2’s complement** representation of +4 and -7 with 5 bits each are shown below.

$+4_{10} = 00100_2$

$-7_{10} = 11001_2$

⇒$+4_{10} + -7_{10} = 00100_2 + 11001_2 = 11101_2$

Here, carry is not obtained from sign bit. The sign bit ‘1’ indicates that the resultant sum is **negative**. So, by taking 2’s complement of it we will get the magnitude of resultant sum as 3 in decimal number system. Therefore, subtraction of two decimal numbers +4 and +7 is -3.

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 required.

8 4 2 1 code

- The weights of this code are 8, 4, 2 and 1.
- This code has all positive weights. So, it is a **positively weighted code**.
- This code is also called as **natural BCD** Binary Coded Decimal **code**.

Example

Let us find the BCD equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the BCD 8421 codes of 7, 8 and 6 are 0111, 1000 and 0110 respectively.

$\therefore$ **$786_{10} = 011110000110_{BCD}$**

There are 12 bits in BCD representation, since each BCD code of decimal digit has 4 bits.

2 4 2 1 code

- The weights of this code are 2, 4, 2 and 1.
- This code has all positive weights. So, it is a **positively weighted code**.
- It is an **unnatural BCD** code. Sum of weights of unnatural BCD codes is equal to 9.
- It is a **self-complementing** code. Self-complementing codes provide the 9's complement of a decimal number, just by interchanging 1's and 0's in its equivalent 2421 representation.

Example

Let us find the 2421 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the 2421 codes of 7, 8 and 6 are 1101, 1110 and 1100 respectively.

Therefore, the 2421 equivalent of the decimal number 786 is **110111101100**.

8 4 -2 -1 code

- The weights of this code are 8, 4, -2 and -1.
- This code has negative weights along with positive weights. So, it is a **negatively weighted code**.
- It is an **unnatural BCD** code.
- It is a **self-complementing** code.

Example

Let us find the 8 4-2-1 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the 8 4 -2 -1 codes of 7, 8 and 6 are 1001, 1000 and 1010 respectively.

Therefore, the 8 4 -2 -1 equivalent of the decimal number 786 is **100110001010**.

Excess 3 code

- This code doesn’t have any weights. So, it is an **un-weighted code**.
- We will get the Excess 3 code of a decimal number by adding three 0011 to the binary equivalent of that decimal number. Hence, it is called as Excess 3 code.
- It is a **self-complementing** code.

Example

Let us find the Excess 3 equivalent of the decimal number 786. This number has 3 decimal digits 7, 8 and 6. From the table, we can write the Excess 3 codes of 7, 8 and 6 are 1010, 1011 and 1001 respectively.

Therefore, the Excess 3 equivalent of the decimal number 786 is **101010111001**

Gray Code

The following table shows the 4-bit Gray codes corresponding to each 4-bit binary code.

Decimal Number	Binary Code	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

- This code doesn’t have any weights. So, it is an **un-weighted code**.
- In the above table, the successive Gray codes are differed in one bit position only. Hence, this code is called as **unit distance** code.

Binary code to Gray Code Conversion

Follow these steps for converting a binary code into its equivalent Gray code.

- Consider the given binary code and place a zero to the left of MSB.
- Compare the successive two bits starting from zero. If the 2 bits are same, then the output is zero. Otherwise, output is one.
- Repeat the above step till the LSB of Gray code is obtained.

Example

From the table, we know that the Gray code corresponding to binary code 1000 is 1100. Now, let us verify it by using the above procedure.

Given, binary code is 1000.

Step 1 − By placing zero to the left of MSB, the binary code will be 01000.

Step 2 − By comparing successive two bits of new binary code, we will get the gray code as **1100**.

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

Error detection codes – are used to detect the errors present in the received data bit stream. These codes contain some bits, which are included appended to the original bit stream. These codes detect the error, if it is occurred during transmission of the original data bit stream.**Example** – Parity code, Hamming code.

Error correction codes – are used to correct the errors present in the received data bit stream so that, we will get the original data. Error correction codes also use the similar strategy of error detection codes.**Example** – Hamming code.

Therefore, to detect and correct the errors, additional bits are appended to the data bits at the time of transmission.

Parity Code

It is easy to include append one parity bit either to the left of MSB or to the right of LSB of original bit stream. There are two types of parity codes, namely even parity code and odd parity code based on the type of parity being chosen.

Even Parity Code

The value of even parity bit should be zero, if even number of ones present in the binary code. Otherwise, it should be one. So that, even number of ones present in **even parity code**. Even parity code contains the data bits and even parity bit.

The following table shows the **even parity codes** corresponding to each 3-bit binary code. Here, the even parity bit is included to the right of LSB of binary code.

Binary Code	Even Parity bit	Even Parity Code
000	0	0000
001	1	0011
010	1	0101
011	0	0110
100	1	1001
101	0	1010
110	0	1100

111	1	1111

Here, the number of bits present in the even parity codes is 4. So, the possible even number of ones in these even parity codes are 0, 2 & 4.

- If the other system receives one of these even parity codes, then there is no error in the received data. The bits other than even parity bit are same as that of binary code.
- If the other system receives other than even parity codes, then there will be an errors in the received data. In this case, we can’t predict the original binary code because we don’t know the bit positions of error.

Therefore, even parity bit is useful only for detection of error in the received parity code. But, it is not sufficient to correct the error.

Odd Parity Code

The value of odd parity bit should be zero, if odd number of ones present in the binary code. Otherwise, it should be one. So that, odd number of ones present in **odd parity code**. Odd parity code contains the data bits and odd parity bit.

The following table shows the **odd parity codes** corresponding to each 3-bit binary code. Here, the odd parity bit is included to the right of LSB of binary code.

Binary Code	Odd Parity bit	Odd Parity Code
000	1	0001
001	0	0010
010	0	0100
011	1	0111
100	0	1000
101	1	1011
110	1	1101
111	0	1110

Here, the number of bits present in the odd parity codes is 4. So, the possible odd number of ones in these odd parity codes are 1 & 3.

- If the other system receives one of these odd parity codes, then there is no error in the received data. The bits other than odd parity bit are same as that of binary code.
- If the other system receives other than odd parity codes, then there is an errors in the received data. In this case, we can’t predict the original binary code because we don’t know the bit positions of error.

Therefore, odd parity bit is useful only for detection of error in the received parity code. But, it is not sufficient to correct the error.

Hamming Code

Hamming code is useful for both detection and correction of error present in the received data. This code uses multiple parity bits and we have to place these parity bits in the positions of powers of 2.

The **minimum value of 'k'** for which the following relation is correct valid is nothing but the required number of parity bits.

2^k\geq n+k+1

Where,

‘n’ is the number of bits in the binary code information

‘k’ is the number of parity bits

Therefore, the number of bits in the Hamming code is equal to n + k.

Let the **Hamming code** is b_{n+k}b_{n+k-1}.....b_{3}b_{2}b_{1} & parity bits p_{k}, p_{k-1}, ...p_{1}. We can place the ‘k’ parity bits in powers of 2 positions only. In remaining bit positions, we can place the ‘n’ bits of binary code.

Based on requirement, we can use either even parity or odd parity while forming a Hamming code. But, the same parity technique should be used in order to find whether any error present in the received data.

Follow this procedure for finding **parity bits**.

- Find the value of $\mathbf{p_1}$, based on the number of ones present in bit positions b_3, b_5, b_7 and so on. All these bit positions suffixes in their equivalent binary have ‘1’ in the place value of 2^0.
- Find the value of $\mathbf{p_2}$, based on the number of ones present in bit positions b_3, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary have ‘1’ in the place value of 2^1.
- Find the value of $\mathbf{p_3}$, based on the number of ones present in bit positions b_5, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary have ‘1’ in the place value of 2^2.
- Similarly, find other values of parity bits.

Follow this procedure for finding **check bits**.

- Find the value of c_1, based on the number of ones present in bit positions b_1, b_3, b_5, b_7 and so on. All these bit positions suffixes in their equivalent binary have ‘1’ in the place value of 2^0.
- Find the value of c_2, based on the number of ones present in bit positions b_2, b_3, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary have ‘1’ in the place value of 2^1.
- Find the value of c_3, based on the number of ones present in bit positions b_4, b_5, b_6, b_7 and so on. All these bit positions suffixes in their equivalent binary

have ‘1’ in the place value of 2^2.

- Similarly, find other values of check bits.

The decimal equivalent of the check bits in the received data gives the value of bit position, where the error is present. Just complement the value present in that bit position. Therefore, we will get the original binary code after removing parity bits.

Example 1

Let us find the Hamming code for binary code, $d_4d_3d_2d_1$ = 1000. Consider even parity bits.

The number of bits in the given binary code is n=4.

We can find the required number of parity bits by using the following mathematical relation.

$2^k\geq n+k+1$

Substitute, n=4 in the above mathematical relation.

$\Rightarrow 2^k\geq 4+k+1$

$\Rightarrow 2^k\geq 5+k$

The minimum value of k that satisfied the above relation is 3. Hence, we require 3 parity bits p_1, p_2, and p_3. Therefore, the number of bits in Hamming code will be 7, since there are 4 bits in binary code and 3 parity bits. We have to place the parity bits and bits of binary code in the Hamming code as shown below.

The **7-bit Hamming code** is $b_{7}b_{6}b_{5}b_{4}b_{3}b_{2}b_{1}=d_{4}d_{3}d_{2}p_{3}d_{1}p_{2}bp_{1}$

By substituting the bits of binary code, the Hamming code will be $b_{7}b_{6}b_{5}b_{4}b_{3}b_{2}b_{1} = 100p_{3}Op_{2}p_{1}$. Now, let us find the parity bits.

$p_{1}=b_{7}\oplus b_{5}\oplus b_{3}=1 \oplus 0 \oplus 0=1$

$p_{2}=b_{7}\oplus b_{6}\oplus b_{3}=1 \oplus 0 \oplus 0=1$

$p_{3}=b_{7}\oplus b_{6}\oplus b_{5}=1 \oplus 0 \oplus 0=1$

By substituting these parity bits, the **Hamming code** will be $b_{7}b_{6}b_{5}b_{4}b_{3}b_{2}b_{1}= 1001011$.

Example 2

In the above example, we got the Hamming code as $b_{7}b_{6}b_{5}b_{4}b_{3}b_{2}b_{1}= 1001011$. Now, let us find the error position when the code received is $b_{7}b_{6}b_{5}b_{4}b_{3}b_{2}b_{1}= 1001111$.

Now, let us find the check bits.

$c_{1}=b_{7}\oplus b_{5}\oplus b_{3}\oplus b_{1}=1 \oplus 0 \oplus 1 \oplus1$

$=1$

$c_{2}=b_{7}\oplus b_{6}\oplus b_{3}\oplus b_{2}=1 \oplus 0 \oplus 1 \oplus1 =1$

$c_{3}=b_{7}\oplus b_{6}\oplus b_{5}\oplus b_{4}=1 \oplus 0 \oplus 0 \oplus1 =0$

The decimal value of check bits gives the position of error in received Hamming code.

$c_{3}c_{2}c_{1} = \left (011 \right)_{2}=\left (3 \right)_{10}$

Therefore, the error present in third bit (b_3) of Hamming code. Just complement the value present in that bit and remove parity bits in order to get the original binary code.

Boolean Algebra is an algebra, which deals with binary numbers & binary variables. Hence, it is also called as Binary Algebra or logical Algebra. A mathematician, named George Boole had developed this algebra in 1854. The variables used in this algebra are also called as Boolean variables.

The range of voltages corresponding to Logic 'High' is represented with '1' and the range of voltages corresponding to logic 'Low' is represented with '0'.

Postulates and Basic Laws of Boolean Algebra

In this section, let us discuss about the Boolean postulates and basic laws that are used in Boolean algebra. These are useful in minimizing Boolean functions.

Boolean Postulates

Consider the binary numbers 0 and 1, Boolean variable x and its complement x'. Either the Boolean variable or complement of it is known as **literal**. The four possible **logical OR** operations among these literals and binary numbers are shown below.

$x + 0 = x$

$x + 1 = 1$

$x + x = x$

$x + x' = 1$

Similarly, the four possible **logical AND** operations among those literals and binary numbers are shown below.

$x.1 = x$

$x.0 = 0$

$x.x = x$

$x.x' = 0$

These are the simple Boolean postulates. We can verify these postulates easily, by substituting the Boolean variable with '0' or '1'.

Note– The complement of complement of any Boolean variable is equal to the variable itself. i.e., x''=x.

Basic Laws of Boolean Algebra

Following are the three basic laws of Boolean Algebra.

- Commutative law
- Associative law
- Distributive law

Commutative Law

If any logical operation of two Boolean variables give the same result irrespective of the order of those two variables, then that logical operation is said to be **Commutative**. The logical OR & logical AND operations of two Boolean variables x & y are shown below

$x + y = y + x$

$x.y = y.x$

The symbol '+' indicates logical OR operation. Similarly, the symbol '.' indicates logical AND operation and it is optional to represent. Commutative law obeys for logical OR & logical AND operations.

Associative Law

If a logical operation of any two Boolean variables is performed first and then the same operation is performed with the remaining variable gives the same result, then that logical operation is said to be **Associative**. The logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

$x + y + z = x + y + z$

$x.y.z = x.y.z$

Associative law obeys for logical OR & logical AND operations.

Distributive Law

If any logical operation can be distributed to all the terms present in the Boolean function, then that logical operation is said to be **Distributive**. The distribution of logical OR & logical AND operations of three Boolean variables x, y & z are shown below.

$x.y + z = x.y + x.z$

$x + y.z = x + y.x + z$

Distributive law obeys for logical OR and logical AND operations.

These are the Basic laws of Boolean algebra. We can verify these laws easily, by substituting the Boolean variables with '0' or '1'.

Theorems of Boolean Algebra

The following two theorems are used in Boolean algebra.

- Duality theorem
- DeMorgan’s theorem

Duality Theorem

This theorem states that the **dual** of the Boolean function is obtained by interchanging the logical AND operator with logical OR operator and zeros with ones. For every Boolean function, there will be a corresponding Dual function.

Let us make the Boolean equations relations that we discussed in the section of Boolean postulates and basic laws into two groups. The following table shows these two groups.

Group1	Group2
x + 0 = x	x.1 = x
x + 1 = 1	x.0 = 0
x + x = x	x.x = x
x + x’ = 1	x.x’ = 0
x + y = y + x	x.y = y.x
x + y + z = x + y + z	x.y.z = x.y.z
x.y + z = x.y + x.z	x + y.z = x + y.x + z

In each row, there are two Boolean equations and they are dual to each other. We can verify all these Boolean equations of Group1 and Group2 by using duality theorem.

DeMorgan’s Theorem

This theorem is useful in finding the **complement of Boolean function**. It states that the complement of logical OR of at least two Boolean variables is equal to the logical AND of each complemented variable.

DeMorgan’s theorem with 2 Boolean variables x and y can be represented as

x + y’ = x’.y’

The dual of the above Boolean function is

x.y’ = x’ + y’

Therefore, the complement of logical AND of two Boolean variables is equal to the logical OR of each complemented variable. Similarly, we can apply DeMorgan’s theorem for more than 2 Boolean variables also.

Simplification of Boolean Functions

Till now, we discussed the postulates, basic laws and theorems of Boolean algebra. Now, let us simplify some Boolean functions.

Example 1

Let us **simplify** the Boolean function, f = p’qr + pq’r + pqr’ + pqr

We can simplify this function in two methods.

Method 1

Given Boolean function, f = p’qr + pq’r + pqr’ +pqr.

Step 1 – In first and second terms r is common and in third and fourth terms pq is common. So, take the common terms by using **Distributive law**.

⇒ f = p’q + pq’r + pqr’ + r

Step 2 – The terms present in first parenthesis can be simplified to Ex-OR operation. The terms present in second parenthesis can be simplified to ‘1’ using **Boolean postulate**

⇒ f = p ⊕qr + pq1

Step 3 – The first term can’t be simplified further. But, the second term can be simplified to pq using **Boolean postulate**.

⇒ f = p ⊕qr + pq

Therefore, the simplified Boolean function is **f = p⊕qr + pq**

Method 2

Given Boolean function, f = p’qr + pq’r + pqr’ + pqr.

Step 1 – Use the **Boolean postulate**, x + x = x. That means, the Logical OR operation with any Boolean variable ‘n’ times will be equal to the same variable. So, we can write the last term pqr two more times.

⇒ f = p’qr + pq’r + pqr’ + pqr + pqr + pqr

Step 2 – Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rd and 6th terms.

⇒ f = qrp’ + p + prq’ + q + pqr’ + r

Step 3 – Use **Boolean postulate**, x + x’ = 1 for simplifying the terms present in each parenthesis.

⇒ f = qr1 + pr1 + pq1

Step 4 – Use **Boolean postulate**, x.1 = x for simplifying the above three terms.

⇒ f = qr + pr + pq

⇒ f = pq + qr + pr

Therefore, the simplified Boolean function is **f = pq + qr + pr**.

So, we got two different Boolean functions after simplifying the given Boolean function in each method. Functionally, those two Boolean functions are same. So, based on the requirement, we can choose one of those two Boolean functions.

Example 2

Let us find the **complement** of the Boolean function, f = p’q + pq’.

The complement of Boolean function is f’ = p’q + pq’’.

Step 1 – Use DeMorgan’s theorem, x + y’ = x’.y’.

⇒ f’ = p’q’.pq’’

Step 2 – Use DeMorgan’s theorem, x.y’ = x’ + y’

⇒ f’ = {p’’ + q’}.{p’ + q’’}

Step3 – Use the Boolean postulate, x’’=x.

⇒ f’ = {p + q’}.{p’ + q}

⇒ f’ = pp’ + pq + p’q’ + qq’

Step 4 – Use the Boolean postulate, xx’=0.

⇒ f = 0 + pq + p’q’ + 0

⇒ f = pq + p’q’

Therefore, the **complement** of Boolean function, p’q + pq’ is **pq + p’q’**.

We will get four Boolean product terms by combining two variables x and y with logical AND operation. These Boolean product terms are called as **min terms** or **standard product terms**. The min terms are x’y’, x’y, xy’ and xy.

Similarly, we will get four Boolean sum terms by combining two variables x and y with logical OR operation. These Boolean sum terms are called as **Max terms** or **standard sum terms**. The Max terms are x + y, x + y’, x’ + y and x’ + y’.

The following table shows the representation of min terms and MAX terms for 2 variables.

x	y	Min terms	Max terms
0	0	m_0=x’y’	M_0=x + y
0	1	m_1=x’y	M_1=x + y’
1	0	m_2=xy’	M_2=x’ + y
1	1	m_3=xy	M_3=x’ + y’

If the binary variable is ‘0’, then it is represented as complement of variable in min term and as the variable itself in Max term. Similarly, if the binary variable is ‘1’, then it is represented as complement of variable in Max term and as the variable itself in min term.

From the above table, we can easily notice that min terms and Max terms are complement of each other. If there are ‘n’ Boolean variables, then there will be 2^n min terms and 2^n Max terms.

Canonical SoP and PoS forms

A truth table consists of a set of inputs and outputs. If there are ‘n’ input variables, then there will be 2^n possible combinations with zeros and ones. So the value of each output variable depends on the combination of input variables. So, each output variable will have ‘1’ for some combination of input variables and ‘0’ for some other combination of input variables.

Therefore, we can express each output variable in following two ways.

- Canonical SoP form
- Canonical PoS form

Canonical SoP form

Canonical SoP form means Canonical Sum of Products form. In this form, each product term contains all literals. So, these product terms are nothing but the min terms. Hence, canonical SoP form is also called as **sum of min terms** form.

First, identify the min terms for which, the output variable is one and then do the logical OR of those min terms in order to get the Boolean expression function corresponding to that output variable. This Boolean function will be in the form of sum of min terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example

Consider the following **truth table**.

Inputs			**Output**
p	**q**	**r**	**f**
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1

1	1	1	1

Here, the output f is ‘1’ for four combinations of inputs. The corresponding min terms are p’qr, pq’r, pqr’, pqr. By doing logical OR of these four min terms, we will get the Boolean function of output f.

Therefore, the Boolean function of output is, f = p’qr + pq’r + pqr’ + pqr. This is the **canonical SoP form** of output, f. We can also represent this function in following two notations.

$f = m_{3}+m_{5}+m_{6}+m_{7}$

$f = \sum m\left (3,5,6,7 \right)$

In one equation, we represented the function as sum of respective min terms. In other equation, we used the symbol for summation of those min terms.

Canonical PoS form

Canonical PoS form means Canonical Product of Sums form. In this form, each sum term contains all literals. So, these sum terms are nothing but the Max terms. Hence, canonical PoS form is also called as **product of Max terms** form.

First, identify the Max terms for which, the output variable is zero and then do the logical AND of those Max terms in order to get the Boolean expression function corresponding to that output variable. This Boolean function will be in the form of product of Max terms.

Follow the same procedure for other output variables also, if there is more than one output variable.

Example

Consider the same truth table of previous example. Here, the output f is ‘0’ for four combinations of inputs. The corresponding Max terms are p + q + r, p + q + r’, p + q’ + r, p’ + q + r. By doing logical AND of these four Max terms, we will get the Boolean function of output f.

Therefore, the Boolean function of output is, f = p + q + r.p + q + r’.p + q’ + r.p’ + q + r. This is the **canonical PoS form** of output, f. We can also represent this function in following two notations.

$f=M_{0}.M_{1}.M_{2}.M_{4}$

$f=\prod M\left (0,1,2,4 \right)$

In one equation, we represented the function as product of respective Max terms. In other equation, we used the symbol for multiplication of those Max terms.

The Boolean function, f = p + q + r.p + q + r’.p + q’ + r.p’ + q + r is the dual of the Boolean function, f = p’qr + pq’r + pqr’ + pqr.

Therefore, both canonical SoP and canonical PoS forms are **Dual** to each other. Functionally, these two forms are same. Based on the requirement, we can use one of these two forms.

Standard SoP and PoS forms

We discussed two canonical forms of representing the Boolean outputs. Similarly, there are two standard forms of representing the Boolean outputs. These are the simplified version of canonical forms.

- Standard SoP form
- Standard PoS form

We will discuss about Logic gates in later chapters. The main **advantage** of standard forms is that the number of inputs applied to logic gates can be minimized. Sometimes, there will be reduction in the total number of logic gates required.

Standard SoP form

Standard SoP form means **Standard Sum of Products** form. In this form, each product term need not contain all literals. So, the product terms may or may not be the min terms. Therefore, the Standard SoP form is the simplified form of canonical SoP form.

We will get Standard SoP form of output variable in two steps.

- Get the canonical SoP form of output variable
- Simplify the above Boolean function, which is in canonical SoP form.

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical SoP form. In that case, both canonical and standard SoP forms are same.

Example

Convert the following Boolean function into Standard SoP form.

f = p’qr + pq’r + pqr’ + pqr

The given Boolean function is in canonical SoP form. Now, we have to simplify this Boolean function in order to get standard SoP form.

Step 1 – Use the **Boolean postulate**, x + x = x. That means, the Logical OR operation with any Boolean variable ‘n’ times will be equal to the same variable. So, we can write the last term pqr two more times.

⇒ f = p’qr + pq’r + pqr’ + pqr + pqr + pqr

Step 2 – Use **Distributive law** for 1st and 4th terms, 2nd and 5th terms, 3rd and 6th terms.

⇒ f = qrp’ + p + prq’ + q + pqr’ + r

Step 3 – Use **Boolean postulate**, x + x’ = 1 for simplifying the terms present in each parenthesis.

⇒ f = qr1 + pr1 + pq1

Step 4 – Use **Boolean postulate**, x.1 = x for simplifying above three terms.

⇒ f = qr + pr + pq

⇒ f = pq + qr + pr

This is the simplified Boolean function. Therefore, the **standard SoP form** corresponding to given canonical SoP form is **f = pq + qr + pr**

Standard PoS form

Standard PoS form means **Standard Product of Sums** form. In this form, each sum term need not contain all literals. So, the sum terms may or may not be the Max terms. Therefore, the Standard PoS form is the simplified form of canonical PoS form.

We will get Standard PoS form of output variable in two steps.

- Get the canonical PoS form of output variable
- Simplify the above Boolean function, which is in canonical PoS form.

Follow the same procedure for other output variables also, if there is more than one output variable. Sometimes, it may not possible to simplify the canonical PoS form. In that case, both canonical and standard PoS forms are same.

Example

Convert the following Boolean function into Standard PoS form.

f = p + q + r.p + q + r’.p + q’ + r.p’ + q + r

The given Boolean function is in canonical PoS form. Now, we have to simplify this Boolean function in order to get standard PoS form.

Step 1 – Use the **Boolean postulate**, x.x = x. That means, the Logical AND operation with any Boolean variable ‘n’ times will be equal to the same variable. So, we can write the first term p+q+r two more times.

⇒ f = p + q + r.p + q + r.p + q + r.p + q + r’.p +q’ + r.p’ + q + r

Step 2 – Use **Distributive law,** x + y.z = x + y.x + z for 1st and 4th parenthesis, 2nd and 5th parenthesis, 3rd and 6th parenthesis.

⇒ f = p + q + rr’.p + r + qq’.q + r + pp’

Step 3 – Use **Boolean postulate**, x.x’=0 for simplifying the terms present in each parenthesis.

⇒ f = p + q + 0.p + r + 0.q + r + 0

Step 4 – Use **Boolean postulate**, x + 0 = x for simplifying the terms present in each parenthesis

⇒ f = p + q.p + r.q + r

⇒ f = p + q.q + r.p + r

This is the simplified Boolean function. Therefore, the **standard PoS form** corresponding to given canonical PoS form is **f = p + q.q + r.p + r**. This is the **dual** of the Boolean function, f = pq + qr + pr.

Therefore, both Standard SoP and Standard PoS forms are Dual to each other.

In previous chapters, we have simplified the Boolean functions using Boolean postulates and theorems. It is a time consuming process and we have to re-write the simplified expressions after each step.

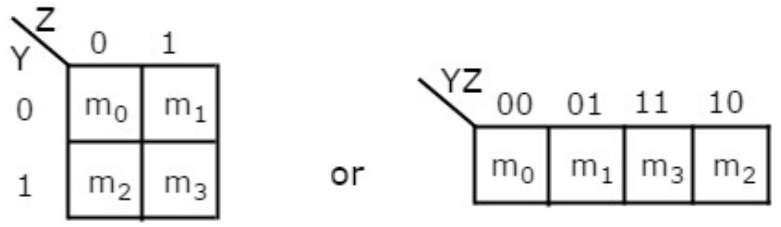
To overcome this difficulty, **Karnaugh** introduced a method for simplification of Boolean functions in an easy way. This method is known as Karnaugh map method or K-map method. It is a graphical method, which consists of 2^n cells for 'n' variables. The adjacent cells are differed only in single bit position.

K-Maps for 2 to 5 Variables

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

2 Variable K-Map

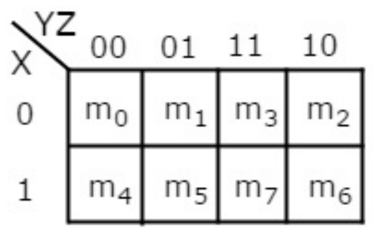
The number of cells in 2 variable K-map is four, since the number of variables is two. The following figure shows **2 variable K-Map**.



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

3 Variable K-Map

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



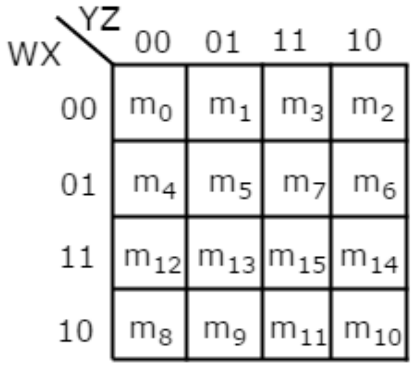
- There is only one possibility of grouping 8 adjacent min terms.
- The possible combinations of grouping 4 adjacent min terms are {(m_0, m_1, m_3, m_2), (m_4, m_5, m_7, m_6), (m_0, m_1, m_4, m_5), (m_1, m_3, m_5, m_7), (m_3, m_2, m_7, m_6)

and (m_2, m_0, m_6, m_4)}.

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

4 Variable K-Map

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

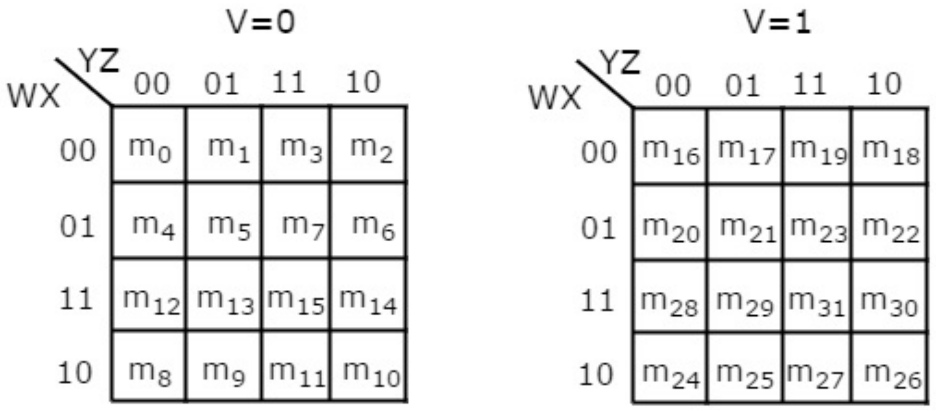


WX \ YZ	00	01	11	10
00	m_0	m_1	m_3	m_2
01	m_4	m_5	m_7	m_6
11	m_{12}	m_{13}	m_{15}	m_{14}
10	m_8	m_9	m_{11}	m_{10}

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

5 Variable K-Map

The number of cells in 5 variable K-map is thirty-two, since the number of variables is 5. The following figure shows **5 variable K-Map**.



V=0

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

V=1

WX \ YZ	00	01	11	10
00	m_{16}	m_{17}	m_{19}	m_{18}
01	m_{20}	m_{21}	m_{23}	m_{22}
11	m_{28}	m_{29}	m_{31}	m_{30}
10	m_{24}	m_{25}	m_{27}	m_{26}

- There is only one possibility of grouping 32 adjacent min terms.
- There are two possibilities of grouping 16 adjacent min terms. i.e., grouping of min terms from m_0 to m_{15} and m_{16} to m_{31}.
- If v=0, then 5 variable K-map becomes 4 variable K-map.

In the above all K-maps, we used exclusively the min terms notation. Similarly, you can use exclusively the Max terms notation.

Minimization of Boolean Functions using K-Maps

If we consider the combination of inputs for which the Boolean function is ‘1’, then we will get the Boolean function, which is in **standard sum of products** form after simplifying the K-map.

Similarly, if we consider the combination of inputs for which the Boolean function is ‘0’, then we will get the Boolean function, which is in **standard product of sums** form after simplifying the K-map.

Follow these **rules for simplifying K-maps** in order to get standard sum of products form.

- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as sum of min terms form, then place the ones at respective min term cells in the K-map. If the Boolean function is given as sum of products form, then place the ones in all possible cells of K-map for which the given product terms are valid.
- Check for the possibilities of grouping maximum number of adjacent ones. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one product term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single ‘1’ is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

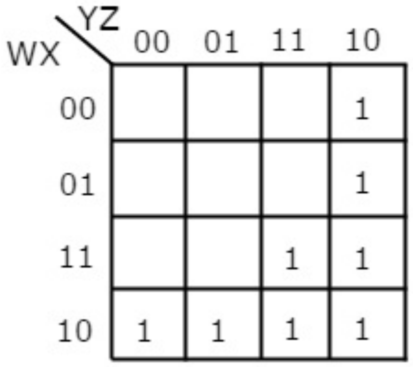
Note 1 – If outputs are not defined for some combination of inputs, then those output values will be represented with **don’t care symbol ‘x’**. That means, we can consider them as either ‘0’ or ‘1’.

Note 2 – If don’t care terms also present, then place don’t cares ‘x’ in the respective cells of K-map. Consider only the don’t cares ‘x’ that are helpful for grouping maximum number of adjacent ones. In those cases, treat the don’t care value as ‘1’.

Example

Let us **simplify** the following Boolean function, **fW, X, Y, Z= WX’Y’ + WY + W’YZ’** using K-map.

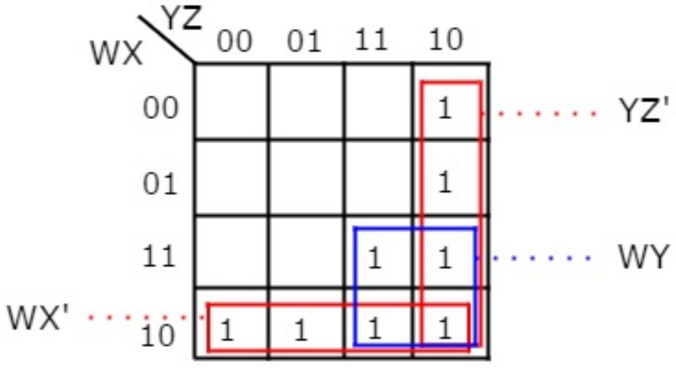
The given Boolean function is in sum of products form. It is having 4 variables W, X, Y & Z. So, we require **4 variable K-map**. The **4 variable K-map** with ones corresponding to the given product terms is shown in the following figure.



Here, 1s are placed in the following cells of K-map.

- The cells, which are common to the intersection of Row 4 and columns 1 & 2 are corresponding to the product term, **WX’Y’**.
- The cells, which are common to the intersection of Rows 3 & 4 and columns 3 & 4 are corresponding to the product term, **WY**.
- The cells, which are common to the intersection of Rows 1 & 2 and column 4 are corresponding to the product term, **W’YZ’**.

There are no possibilities of grouping either 16 adjacent ones or 8 adjacent ones. There are three possibilities of grouping 4 adjacent ones. After these three groupings, there is no single one left as ungrouped. So, we no need to check for grouping of 2 adjacent ones. The **4 variable K-map** with these three **groupings** is shown in the following figure.



Here, we got three prime implicants WX’, WY & YZ’. All these prime implicants are **essential** because of following reasons.

- Two ones **(m_8 & m_9)** of fourth row grouping are not covered by any other groupings. Only fourth row grouping covers those two ones.
- Single one **(m_{15})** of square shape grouping is not covered by any other

groupings. Only the square shape grouping covers that one.

- Two ones **(m_2 & m_6)** of fourth column grouping are not covered by any other groupings. Only fourth column grouping covers those two ones.

Therefore, the **simplified Boolean function** is

f = WX' + WY + YZ'

Follow these **rules for simplifying K-maps** in order to get standard product of sums form.

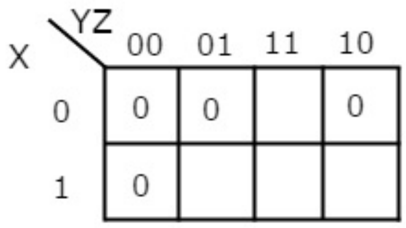
- Select the respective K-map based on the number of variables present in the Boolean function.
- If the Boolean function is given as product of Max terms form, then place the zeroes at respective Max term cells in the K-map. If the Boolean function is given as product of sums form, then place the zeroes in all possible cells of K-map for which the given sum terms are valid.
- Check for the possibilities of grouping maximum number of adjacent zeroes. It should be powers of two. Start from highest power of two and upto least power of two. Highest power is equal to the number of variables considered in K-map and least power is zero.
- Each grouping will give either a literal or one sum term. It is known as **prime implicant**. The prime implicant is said to be **essential prime implicant**, if atleast single '0' is not covered with any other groupings but only that grouping covers.
- Note down all the prime implicants and essential prime implicants. The simplified Boolean function contains all essential prime implicants and only the required prime implicants.

Note – If don't care terms also present, then place don't cares 'x' in the respective cells of K-map. Consider only the don't cares 'x' that are helpful for grouping maximum number of adjacent zeroes. In those cases, treat the don't care value as '0'.

Example

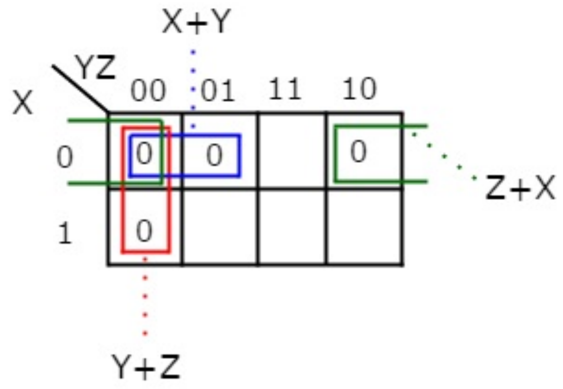
Let us **simplify** the following Boolean function, f\left (X,Y,Z \right)=\prod M\left (0,1,2,4 \right) using K-map.

The given Boolean function is in product of Max terms form. It is having 3 variables X, Y & Z. So, we require 3 variable K-map. The given Max terms are M_0, M_1, M_2 & M_4. The 3 **variable K-map** with zeroes corresponding to the given Max terms is shown in the following figure.



X \ YZ	00	01	11	10
0	0	0		0
1	0			

There are no possibilities of grouping either 8 adjacent zeroes or 4 adjacent zeroes. There are three possibilities of grouping 2 adjacent zeroes. After these three groupings, there is no single zero left as ungrouped. The **3 variable K-map** with these three **groupings** is shown in the following figure.



Here, we got three prime implicants X + Y, Y + Z & Z + X. All these prime implicants are **essential** because one zero in each grouping is not covered by any other groupings except with their individual groupings.

Therefore, the **simplified Boolean function** is

f = X + Y.Y + Z.Z + X

In this way, we can easily simplify the Boolean functions up to 5 variables using K-map method. For more than 5 variables, it is difficult to simplify the functions using K-Maps. Because, the number of **cells** in K-map gets **doubled** by including a new variable.

Due to this checking and grouping of adjacent ones min terms or adjacent zeros Max terms will be complicated. We will discuss **Tabular method** in next chapter to overcome the difficulties of K-map method.

In previous chapter, we discussed K-map method, which is a convenient method for minimizing Boolean functions up to 5 variables. But, it is difficult to simplify the Boolean functions having more than 5 variables by using this method.

Quine-McClukey tabular method is a tabular method based on the concept of prime implicants. We know that **prime implicant** is a product or sum term, which can’t be further reduced by combining with any other product or sum terms of the given Boolean function.

This tabular method is useful to get the prime implicants by repeatedly using the following Boolean identity.

xy + xy’ = xy + y’ = x.1 = x

Procedure of Quine-McCluskey Tabular Method

Follow these steps for simplifying Boolean functions using Quine-McClukey tabular method.

Step 1 – Arrange the given min terms in an **ascending order** and make the groups based on the number of ones present in their binary representations.

So, there will be **at most 'n+1' groups** if there are 'n' Boolean variables in a Boolean function or 'n' bits in the binary equivalent of min terms.

Step 2 – Compare the min terms present in **successive groups**. If there is a change in only one-bit position, then take the pair of those two min terms. Place this symbol '_' in the differed bit position and keep the remaining bits as it is.

Step 3 – Repeat step2 with newly formed terms till we get all **prime implicants**.

Step 4 – Formulate the **prime implicant table**. It consists of set of rows and columns. Prime implicants can be placed in row wise and min terms can be placed in column wise. Place '1' in the cells corresponding to the min terms that are covered in each prime implicant.

Step 5 – Find the essential prime implicants by observing each column. If the min term is covered only by one prime implicant, then it is **essential prime implicant**. Those essential prime implicants will be part of the simplified Boolean function.

Step 6 – Reduce the prime implicant table by removing the row of each essential prime implicant and the columns corresponding to the min terms that are covered in that essential prime implicant. Repeat step 5 for Reduced prime implicant table. Stop this process when all min terms of given Boolean function are over.

Example

Let us **simplify** the following Boolean function, f\left (W,X,Y,Z \right)=\sum m\left (2,6,8,9,10,11,14,15 \right) using Quine-McClukey tabular method.

The given Boolean function is in **sum of min terms** form. It is having 4 variables W, X, Y & Z. The given min terms are 2, 6, 8, 9, 10, 11, 14 and 15. The ascending order of these min terms based on the number of ones present in their binary equivalent is 2, 8, 6, 9, 10, 11, 14 and 15. The following table shows these **min terms and their equivalent binary** representations.

Group Name	Min terms	W	X	Y	Z
GA1	2	0	0	1	0
	8	1	0	0	0
GA2	6	0	1	1	0
	9	1	0	0	1
	10	1	0	1	0
GA3	11	1	0	1	1
	14	1	1	1	0
GA4	15	1	1	1	1

The given min terms are arranged into 4 groups based on the number of ones present in their binary equivalents. The following table shows the possible

merging of min terms from adjacent groups.

Group Name	Min terms	W	X	Y	Z
GB1	2,6	0	-	1	0
	2,10	-	0	1	0
	8,9	1	0	0	-
	8,10	1	0	-	0
GB2	6,14	-	1	1	0
	9,11	1	0	-	1
	10,11	1	0	1	-
	10,14	1	-	1	0
GB3	11,15	1	-	1	1
	14,15	1	1	1	-

The min terms, which are differed in only one-bit position from adjacent groups are merged. That differed bit is represented with this symbol, '-'. In this case, there are three groups and each group contains combinations of two min terms. The following table shows the possible **merging of min term pairs** from adjacent groups.

Group Name	Min terms	W	X	Y	Z
GB1	2,6,10,14	-	-	1	0
	2,10,6,14	-	-	1	0
	8,9,10,11	1	0	-	-
	8,10,9,11	1	0	-	-
GB2	10,11,14,15	1	-	1	-
	10,14,11,15	1	-	1	-

The successive groups of min term pairs, which are differed in only one-bit position are merged. That differed bit is represented with this symbol, '-'. In this case, there are two groups and each group contains combinations of four min terms. Here, these combinations of 4 min terms are available in two rows. So, we can remove the repeated rows. The reduced table after removing the redundant rows is shown below.

Group Name	Min terms	W	X	Y	Z
GC1	2,6,10,14	-	-	1	0
	8,9,10,11	1	0	-	-
GC2	10,11,14,15	1	-	1	-

Further merging of the combinations of min terms from adjacent groups is not possible, since they are differed in more than one-bit position. There are three rows in the above table. So, each row will give one prime implicant. Therefore, the **prime implicants** are YZ', WX' & WY.

The **prime implicant table** is shown below.

Min terms / Prime Implicants	**2**	**6**	**8**	**9**	**10**	**11**	**14**	**15**
YZ'	1	1			1		1	
WX'			1	1	1	1		
WY					1	1	1	1

The prime implicants are placed in row wise and min terms are placed in column wise. 1s are placed in the common cells of prime implicant rows and the corresponding min term columns.

The min terms 2 and 6 are covered only by one prime implicant **YZ'**. So, it is an **essential prime implicant**. This will be part of simplified Boolean function. Now, remove this prime implicant row and the corresponding min term columns. The reduced prime implicant table is shown below.

Min terms / Prime Implicants	**8**	**9**	**11**	**15**
WX'	1	1	1	
WY			1	1

The min terms 8 and 9 are covered only by one prime implicant **WX'**. So, it is an **essential prime implicant**. This will be part of simplified Boolean function. Now, remove this prime implicant row and the corresponding min term columns. The reduced prime implicant table is shown below.

Min terms / Prime Implicants	**15**
WY	1

The min term 15 is covered only by one prime implicant **WY**. So, it is an **essential prime implicant**. This will be part of simplified Boolean function.

In this example problem, we got three prime implicants and all the three are essential. Therefore, the **simplified Boolean function** is

fW,X,Y,Z = YZ' + WX' + WY.

Digital electronic circuits operate with voltages of **two logic levels**namely Logic Low and Logic High. The range of voltages corresponding to Logic Low is represented with '0'. Similarly, the range of voltages corresponding to Logic High is represented with '1'.

The basic digital electronic circuit that has one or more inputs and single output is known as **Logic gate**. Hence, the Logic gates are the building blocks of any digital system. We can classify these Logic gates into the following three categories.

- Basic gates

- Universal gates
- Special gates

Now, let us discuss about the Logic gates come under each category one by one.

Basic Gates

In earlier chapters, we learnt that the Boolean functions can be represented either in sum of products form or in product of sums form based on the requirement. So, we can implement these Boolean functions by using basic gates. The basic gates are AND, OR & NOT gates.

AND gate

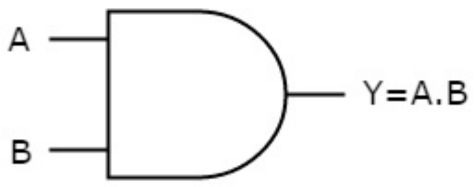
An AND gate is a digital circuit that has two or more inputs and produces an output, which is the **logical AND** of all those inputs. It is optional to represent the **Logical AND** with the symbol '.'.

The following table shows the **truth table** of 2-input AND gate.

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

Here A, B are the inputs and Y is the output of two input AND gate. If both inputs are '1', then only the output, Y is '1'. For remaining combinations of inputs, the output, Y is '0'.

The following figure shows the **symbol** of an AND gate, which is having two inputs A, B and one output, Y.



This AND gate produces an output Y, which is the **logical AND** of two inputs A, B. Similarly, if there are 'n' inputs, then the AND gate produces an output, which is the logical AND of all those inputs. That means, the output of AND gate will be '1', when all the inputs are '1'.

OR gate

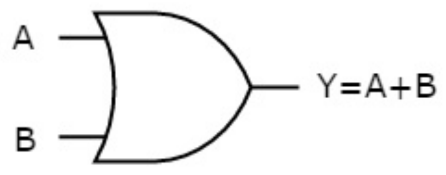
An OR gate is a digital circuit that has two or more inputs and produces an output, which is the logical OR of all those inputs. This **logical OR** is represented with the symbol ‘+’.

The following table shows the **truth table** of 2-input OR gate.

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

Here A, B are the inputs and Y is the output of two input OR gate. If both inputs are ‘0’, then only the output, Y is ‘0’. For remaining combinations of inputs, the output, Y is ‘1’.

The following figure shows the **symbol** of an OR gate, which is having two inputs A, B and one output, Y.



This OR gate produces an output Y, which is the **logical OR** of two inputs A, B. Similarly, if there are ‘n’ inputs, then the OR gate produces an output, which is the logical OR of all those inputs. That means, the output of an OR gate will be ‘1’, when at least one of those inputs is ‘1’.

NOT gate

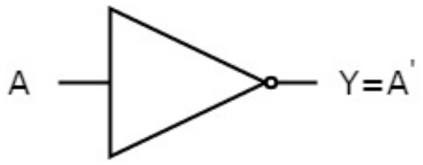
A NOT gate is a digital circuit that has single input and single output. The output of NOT gate is the **logical inversion** of input. Hence, the NOT gate is also called as inverter.

The following table shows the **truth table** of NOT gate.

A	Y = A’
0	1
1	0

Here A and Y are the input and output of NOT gate respectively. If the input, A is ‘0’, then the output, Y is ‘1’. Similarly, if the input, A is ‘1’, then the output, Y is ‘0’.

The following figure shows the **symbol** of NOT gate, which is having one input, A and one output, Y.



This NOT gate produces an output Y, which is the **complement** of input, A.

Universal gates

NAND & NOR gates are called as **universal gates**. Because we can implement any Boolean function, which is in sum of products form by using NAND gates alone. Similarly, we can implement any Boolean function, which is in product of sums form by using NOR gates alone.

NAND gate

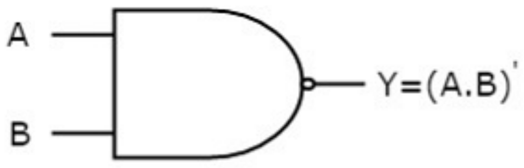
NAND gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical AND** of all those inputs.

The following table shows the **truth table** of 2-input NAND gate.

A	**B**	**Y = A.B’**
0	0	1
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input NAND gate. When both inputs are ‘1’, the output, Y is ‘0’. If at least one of the input is zero, then the output, Y is ‘1’. This is just opposite to that of two input AND gate operation.

The following image shows the **symbol** of NAND gate, which is having two inputs A, B and one output, Y.



NAND gate operation is same as that of AND gate followed by an inverter. That’s why the NAND gate symbol is represented like that.

NOR gate

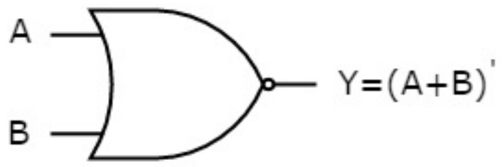
NOR gate is a digital circuit that has two or more inputs and produces an output, which is the **inversion of logical OR** of all those inputs.

The following table shows the **truth table** of 2-input NOR gate

A	B	Y = A+B’
0	0	1
0	1	0
1	0	0
1	1	0

Here A, B are the inputs and Y is the output. If both inputs are ‘0’, then the output, Y is ‘1’. If at least one of the input is ‘1’, then the output, Y is ‘0’. This is just opposite to that of two input OR gate operation.

The following figure shows the **symbol** of NOR gate, which is having two inputs A, B and one output, Y.



NOR gate operation is same as that of OR gate followed by an inverter. That’s why the NOR gate symbol is represented like that.

Special Gates

Ex-OR & Ex-NOR gates are called as special gates. Because, these two gates are special cases of OR & NOR gates.

Ex-OR gate

The full form of Ex-OR gate is **Exclusive-OR** gate. Its function is same as that of OR gate except for some cases, when the inputs having even number of ones.

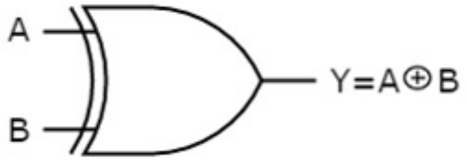
The following table shows the **truth table** of 2-input Ex-OR gate.

A	B	Y = A⊕B
0	0	0
0	1	1
1	0	1
1	1	0

Here A, B are the inputs and Y is the output of two input Ex-OR gate. The truth table of Ex-OR gate is same as that of OR gate for first three rows. The only modification is in the fourth row. That means, the output Y is zero instead of one, when both the inputs are one, since the inputs having even number of ones.

Therefore, the output of Ex-OR gate is ‘1’, when only one of the two inputs is ‘1’. And it is zero, when both inputs are same.

Below figure shows the **symbol** of Ex-OR gate, which is having two inputs A, B and one output, Y.



Ex-OR gate operation is similar to that of OR gate, except for few combinations of inputs. That’s why the Ex-OR gate symbol is represented like that. The output of Ex-OR gate is ‘1’, when odd number of ones present at the inputs. Hence, the output of Ex-OR gate is also called as an **odd function**.

Ex-NOR gate

The full form of Ex-NOR gate is **Exclusive-NOR** gate. Its function is same as that of NOR gate except for some cases, when the inputs having even number of ones.

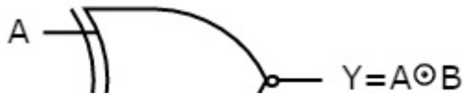
The following table shows the **truth table** of 2-input Ex-NOR gate.

A	B	Y = A⊙B
0	0	1
0	1	0
1	0	0
1	1	1

Here A, B are the inputs and Y is the output. The truth table of Ex-NOR gate is same as that of NOR gate for first three rows. The only modification is in the fourth row. That means, the output is one instead of zero, when both the inputs are one.

Therefore, the output of Ex-NOR gate is ‘1’, when both inputs are same. And it is zero, when both the inputs are different.

The following figure shows the **symbol** of Ex-NOR gate, which is having two inputs A, B and one output, Y.



Ex-NOR gate operation is similar to that of NOR gate, except for few combinations of inputs. That's why the Ex-NOR gate symbol is represented like that. The output of Ex-NOR gate is '1', when even number of ones present at the inputs. Hence, the output of Ex-NOR gate is also called as an **even function**.

From the above truth tables of Ex-OR & Ex-NOR logic gates, we can easily notice that the Ex-NOR operation is just the logical inversion of Ex-OR operation.

The maximum number of levels that are present between inputs and output is two in **two level logic**. That means, irrespective of total number of logic gates, the maximum number of Logic gates that are present cascaded between any input and output is two in two level logic. Here, the outputs of first level Logic gates are connected as inputs of second level Logic gates.

Consider the four Logic gates AND, OR, NAND & NOR. Since, there are 4 Logic gates, we will get 16 possible ways of realizing two level logic. Those are AND-AND, AND-OR, ANDNAND, AND-NOR, OR-AND, OR-OR, OR-NAND, OR-NOR, NAND-AND, NAND-OR, NANDNAND, NAND-NOR, NOR-AND, NOR-OR, NOR-NAND, NOR-NOR.

These two level logic realizations can be classified into the following two categories.

- Degenerative form
- Non-degenerative form

Degenerative Form

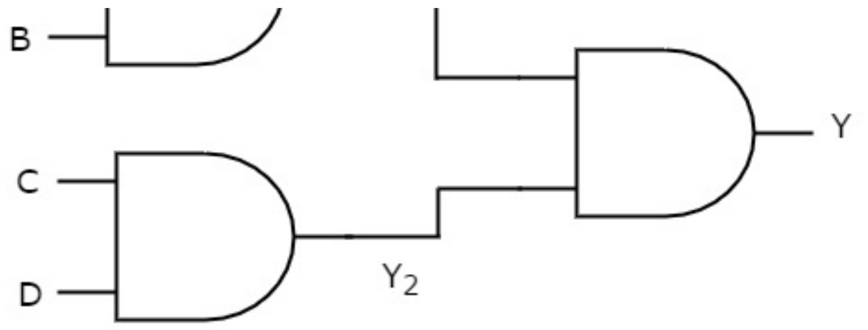
If the output of two level logic realization can be obtained by using single Logic gate, then it is called as **degenerative form**. Obviously, the number of inputs of single Logic gate increases. Due to this, the fan-in of Logic gate increases. This is an advantage of degenerative form.

Only **6 combinations** of two level logic realizations out of 16 combinations come under degenerative form. Those are AND-AND, AND-NAND, OR-OR, OR-NOR, NAND-NOR, NORNAND.

In this section, let us discuss some realizations. Assume, A, B, C & D are the inputs and Y is the output in each logic realization.

AND-AND Logic

In this logic realization, AND gates are present in both levels. Below figure shows an example for **AND-AND logic** realization.



We will get the outputs of first level logic gates as Y_{1}=AB and Y_{2}=CD

These outputs, Y_{1} and Y_{2} are applied as inputs of AND gate that is present in second level. So, the output of this AND gate is

Y=Y_{1}Y_{2}

Substitute Y_{1} and Y_{2} values in the above equation.

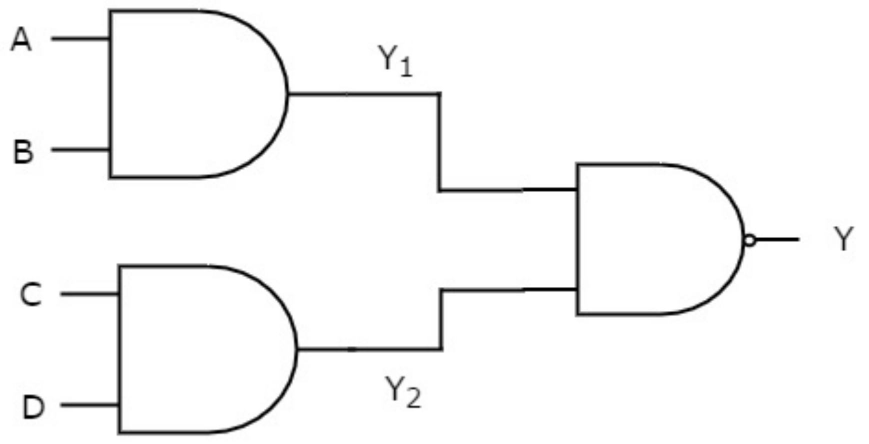
Y=\left (AB \right)\left (CD \right)

\Rightarrow Y=ABCD

Therefore, the output of this AND-AND logic realization is **ABCD**. This Boolean function can be implemented by using a 4 input AND gate. Hence, it is **degenerative form**.

AND-NAND Logic

In this logic realization, AND gates are present in first level and NAND gates are present in second level. The following figure shows an example for **AND-NAND logic** realization.



Previously, we got the outputs of first level logic gates as Y_{1} = AB and Y_{2} = CD

These outputs,Y_{1} and Y_{2} are applied as inputs of NAND gate that is present in second level. So, the output of this NAND gate is

Y={\left (Y_{1}Y_{2} \right)}'

Substitute Y_{1} and Y_{2} values in the above equation.

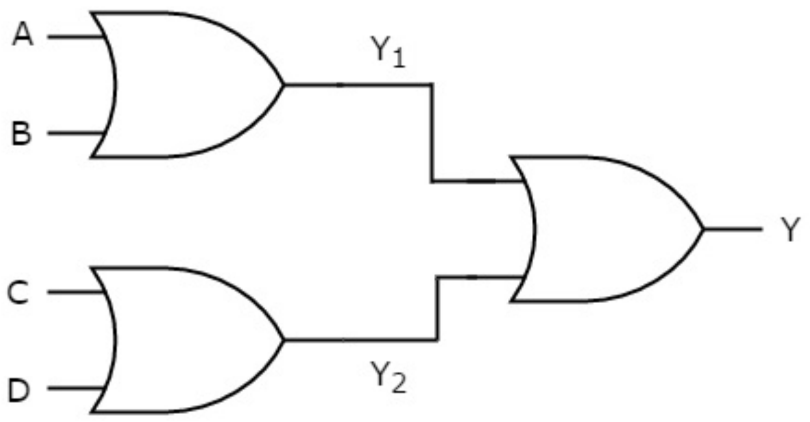
$Y={\left (\left (AB \right) \left (CD \right)\right)}'$

$\Rightarrow Y={\left (ABCD \right)}'$

Therefore, the output of this AND-NAND logic realization is ${\left (ABCD \right)}'$. This Boolean function can be implemented by using a 4 input NAND gate. Hence, it is **degenerative form**.

OR-OR Logic

In this logic realization, OR gates are present in both levels. The following figure shows an example for **OR-OR logic** realization.



We will get the outputs of first level logic gates as $Y_{1}=A+B$ and $Y_{2}=C+D$.

These outputs, Y_{1} and Y_{2} are applied as inputs of OR gate that is present in second level. So, the output of this OR gate is

$Y=Y_{1}+Y_{2}$

Substitute Y_{1} and Y_{2} values in the above equation.

$Y=\left (A+B \right)+\left (C+D \right)$

$\Rightarrow Y=A+B+C+D$

Therefore, the output of this OR-OR logic realization is **A+B+C+D**. This Boolean function can be implemented by using a 4 input OR gate. Hence, it is **degenerative form**.

Similarly, you can verify whether the remaining realizations belong to this category or not.

Non-degenerative Form

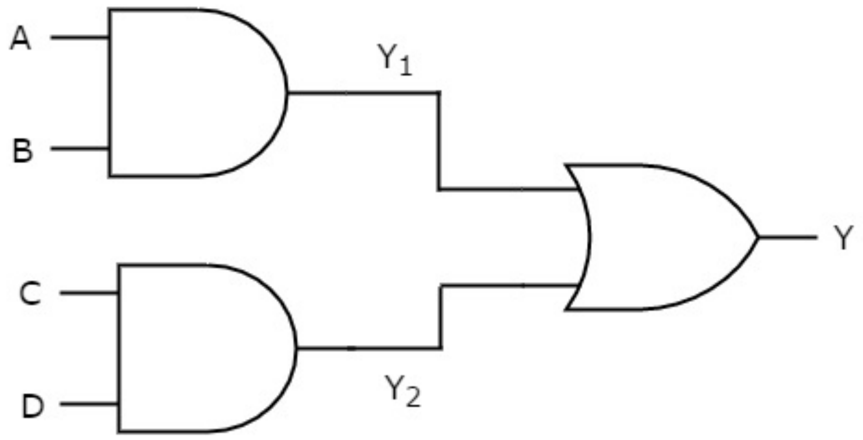
If the output of two level logic realization can’t be obtained by using single logic gate, then it is called as **non-degenerative form**.

The remaining **10 combinations** of two level logic realizations come under nondegenerative form. Those are AND-OR, AND-NOR, OR-AND, OR-NAND, NAND-AND, NANDOR, NAND-NAND, NOR-AND, NOR-OR, NOR-NOR.

Now, let us discuss some realizations. Assume, A, B, C & D are the inputs and Y is the output in each logic realization.

AND-OR Logic

In this logic realization, AND gates are present in first level and OR gates are present in second level. Below figure shows an example for **AND-OR logic** realization.



Previously, we got the outputs of first level logic gates as Y_{1} = AB and Y_{2} = CD.

These outputs, Y1 and Y2 are applied as inputs of OR gate that is present in second level. So, the output of this OR gate is

Y=Y_{1}+Y_{2}

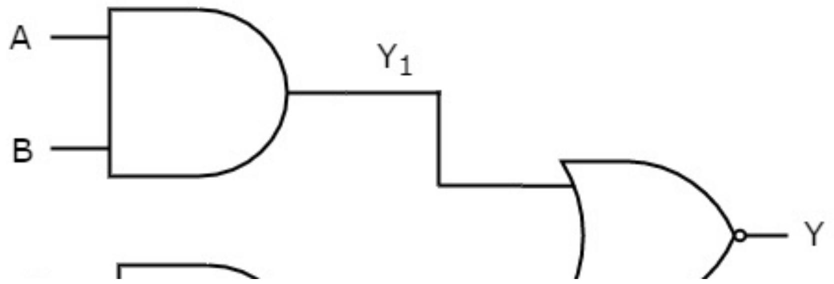
Substitute Y_{1} and Y_{2} values in the above equation

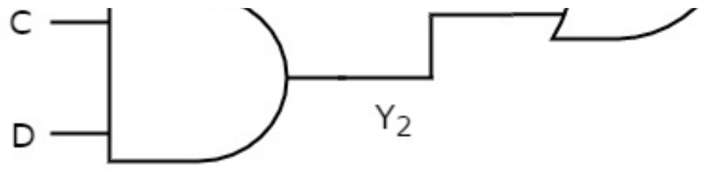
Y=AB+CD

Therefore, the output of this AND-OR logic realization is **AB+CD**. This Boolean function is in **Sum of Products** form. Since, we can’t implement it by using single logic gate, this AND-OR logic realization is a **non-degenerative form**.

AND-NOR Logic

In this logic realization, AND gates are present in first level and NOR gates are present in second level. The following figure shows an example for **AND-NOR logic** realization.





We know the outputs of first level logic gates as $Y_{1} = AB$ and $Y_{2} = CD$

These outputs, Y1 and Y2 are applied as inputs of NOR gate that is present in second level. So, the output of this NOR gate is

$Y={\left (Y_{1}+Y_{2} \right)}'$

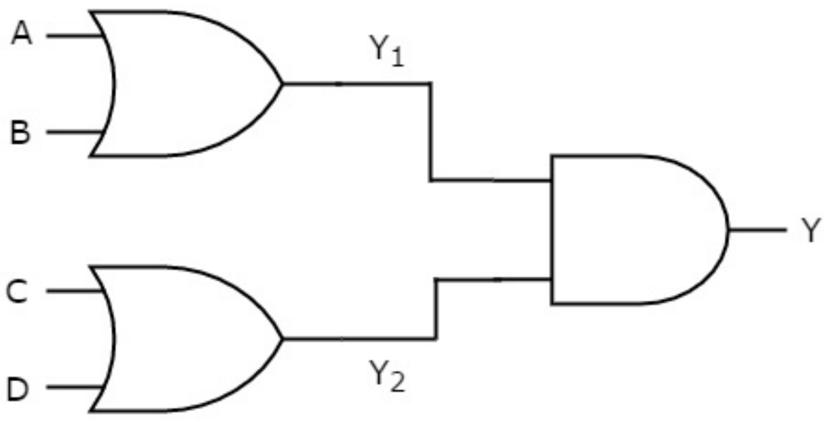
Substitute Y_{1} and Y_{2} values in the above equation.

$Y={\left (AB+CD \right)}'$

Therefore, the output of this AND-NOR logic realization is ${\left (AB+CD \right)}'$. This Boolean function is in **AND-OR-Invert** form. Since, we can’t implement it by using single logic gate, this AND-NOR logic realization is a **non-degenerative form**

OR-AND Logic

In this logic realization, OR gates are present in first level & AND gates are present in second level. The following figure shows an example for **OR-AND logic** realization.



Previously, we got the outputs of first level logic gates as $Y_{1}=A+B$ and $Y_{2}=C+D$.

These outputs, Y_{1} and Y_{2} are applied as inputs of AND gate that is present in second level. So, the output of this AND gate is

$Y=Y_{1}Y_{2}$

Substitute Y_{1} and Y_{2} values in the above equation.

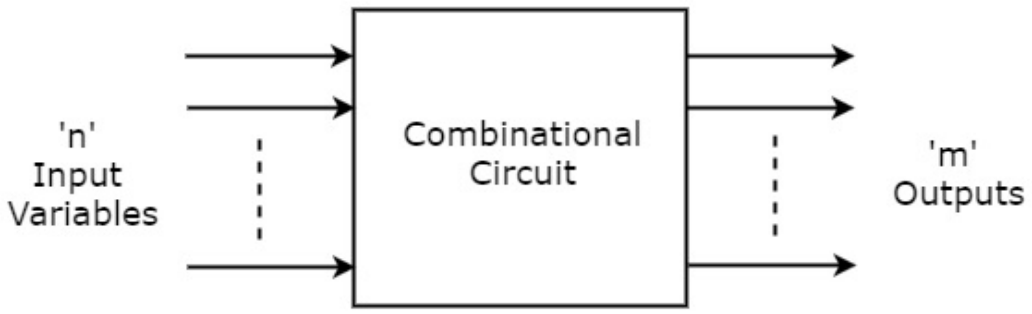
$Y = \left (A+B \right)\left (C+D \right)$

Therefore, the output of this OR-AND logic realization is **A + B C + D**. This Boolean function is in **Product of Sums** form. Since, we can’t implement it by using single logic gate, this OR-AND logic realization is a **non-degenerative**

form.

Similarly, you can verify whether the remaining realizations belong to this category or not.

Combinational circuits consist of Logic gates. These circuits operate with binary values. The outputs of combinational circuit depends on the combination of present inputs. The following figure shows the **block diagram** of combinational circuit.



This combinational circuit has ‘n’ input variables and ‘m’ outputs. Each combination of input variables will affect the outputs.

Design procedure of Combinational circuits

- Find the required number of input variables and outputs from given specifications.
- Formulate the **Truth table**. If there are ‘n’ input variables, then there will be 2n possible combinations. For each combination of input, find the output values.
- Find the **Boolean expressions** for each output. If necessary, simplify those expressions.
- Implement the above Boolean expressions corresponding to each output by using **Logic gates**.

Code Converters

We have discussed various codes in the chapter named codes. The converters, which convert one code to other code are called as **code converters**. These code converters basically consist of Logic gates.

Example

Binary code to Gray code converter

Let us implement a converter, which converts a 4-bit binary code WXYZ into its equivalent Gray code ABCD.

The following table shows the **Truth table** of a 4-bit binary code to Gray code converter.

Binary code WXYZ	WXYZ Gray code ABCD
0000	0000
0001	0001
0010	0011
0011	0010
0100	0110
0101	0111
0110	0101
0111	0100
1000	1100
1001	1101
1010	1111
1011	1110
1100	1010
1101	1011
1110	1001
1111	1000

From Truth table, we can write the **Boolean functions** for each output bit of Gray code as below.

A=\sum m\left (8,9,10,11,12,13,14,15 \right)

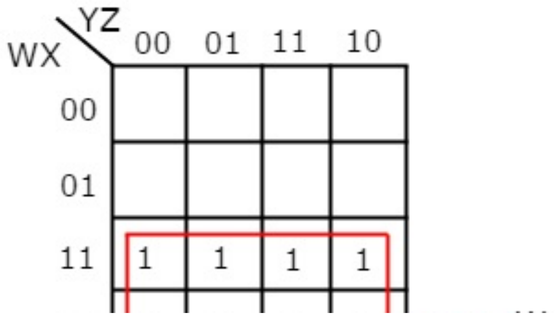
B=\sum m\left (4,5,6,7,8,9,10,11 \right)

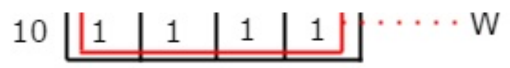
C=\sum m\left (2,3,4,5,10,11,12,13 \right)

D=\sum m\left (1,2,5,6,9,10,13,14 \right)

Let us simplify the above functions using 4 variable K-Maps.

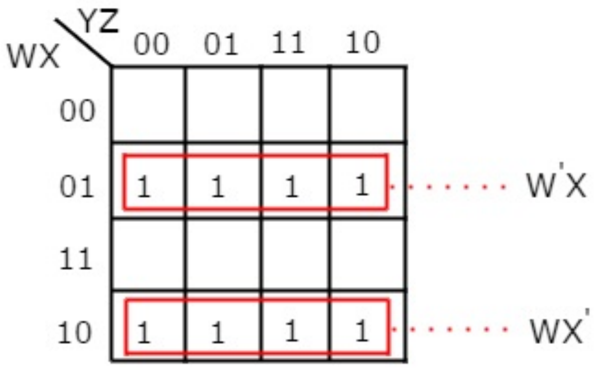
The following figure shows the **4 variable K-Map** for simplifying **Boolean function, A**.





By grouping 8 adjacent ones, we got A=W.

The following figure shows the **4 variable K-Map** for simplifying **Boolean function, B**.



There are two groups of 4 adjacent ones. After grouping, we will get B as

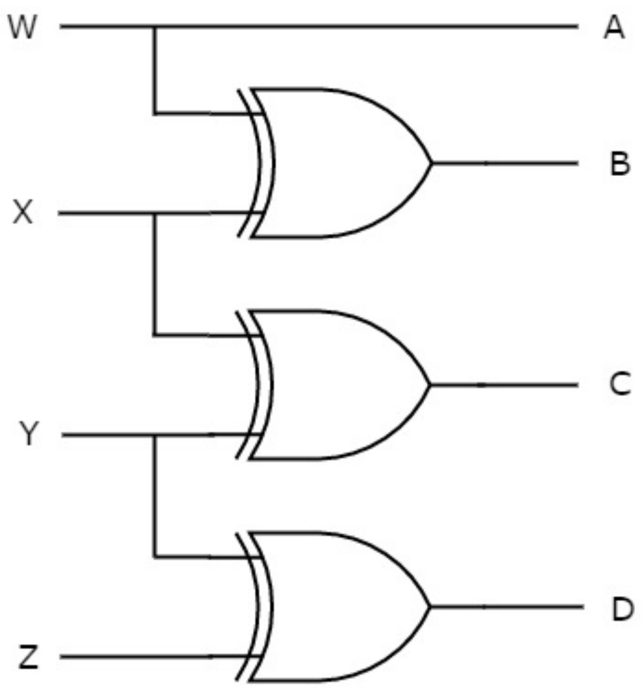
$B={W}'X+W{X}'=W\oplus X$

Similarly, we will get the following Boolean functions for C & D after simplifying.

$C={X}'Y+X{Y}'=X \oplus Y$

$D={Y}'Z+Y{Z}'=Y \oplus Z$

The following figure shows the **circuit diagram** of 4-bit binary code to Gray code converter.



Since the outputs depend only on the present inputs, this 4-bit Binary code to Gray code converter is a combinational circuit. Similarly, you can implement other code converters.

Parity Bit Generator

There are two types of parity bit generators based on the type of parity bit being generated. **Even parity generator** generates an even parity bit. Similarly, **odd parity generator** generates an odd parity bit.

Even Parity Generator

Now, let us implement an even parity generator for a 3-bit binary input, WXY. It generates an even parity bit, P. If odd number of ones present in the input, then even parity bit, P should be ‘1’ so that the resultant word contains even number of ones. For other combinations of input, even parity bit, P should be ‘0’. The following table shows the **Truth table** of even parity generator.

Binary Input WXY	Even Parity bit P
000	0
001	1
010	1
011	0
100	1
101	0
110	0
111	1

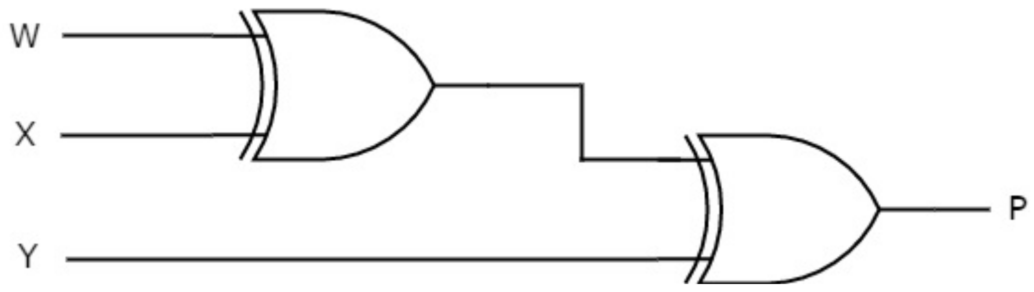
From the above Truth table, we can write the **Boolean function** for even parity bit as

$P={W}'{X}'Y+{W}'X{Y}'+W{X}'{Y}'+WXY$

$\Rightarrow P={W}'\left ({X}'Y+X{Y}' \right)+W\left ({X}'{Y}'+XY \right)$

$\Rightarrow P={W}'\left (X \oplus Y \right)+W{\left (X \oplus Y \right)}'=W \oplus X \oplus Y$

The following figure shows the **circuit diagram** of even parity generator.

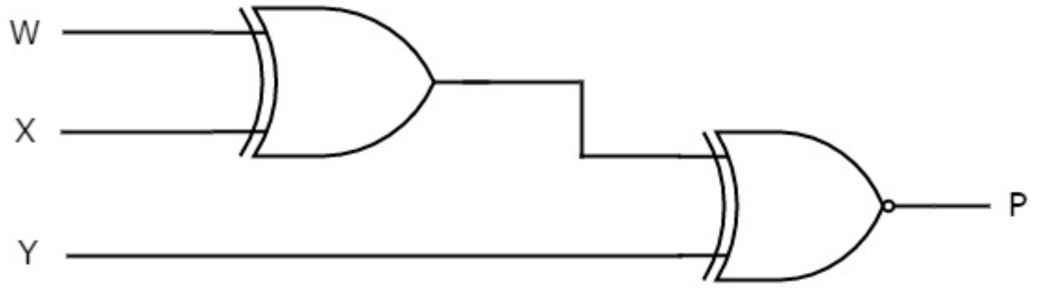


This circuit consists of two **Exclusive-OR gates** having two inputs each. First ExclusiveOR gate having two inputs W & X and produces an output W ⊕ X. This output is given as one input of second Exclusive-OR gate. The other input of this second Exclusive-OR gate is Y and produces an output of W ⊕ X ⊕ Y.

Odd Parity Generator

If even number of ones present in the input, then odd parity bit, P should be ‘1’ so that the resultant word contains odd number of ones. For other combinations of input, odd parity bit, P should be ‘0’.

Follow the same procedure of even parity generator for implementing odd parity generator. The **circuit diagram** of odd parity generator is shown in the following figure.



The above circuit diagram consists of Ex-OR gate in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity generator. In that case, the first and second levels contain an ExOR gate in each level and third level consist of an inverter.

Parity Checker

There are two types of parity checkers based on the type of parity has to be checked. **Even parity checker** checks error in the transmitted data, which contains message bits along with even parity. Similarly, **odd parity checker** checks error in the transmitted data, which contains message bits along with odd parity.

Even parity checker

Now, let us implement an even parity checker circuit. Assume a 3-bit binary input, WXY is transmitted along with an even parity bit, P. So, the resultant word data contains 4 bits, which will be received as the input of even parity checker.

It generates an **even parity check bit, E**. This bit will be zero, if the received data contains an even number of ones. That means, there is no error in the received data. This even parity check bit will be one, if the received data contains an odd number of ones. That means, there is an error in the received data.

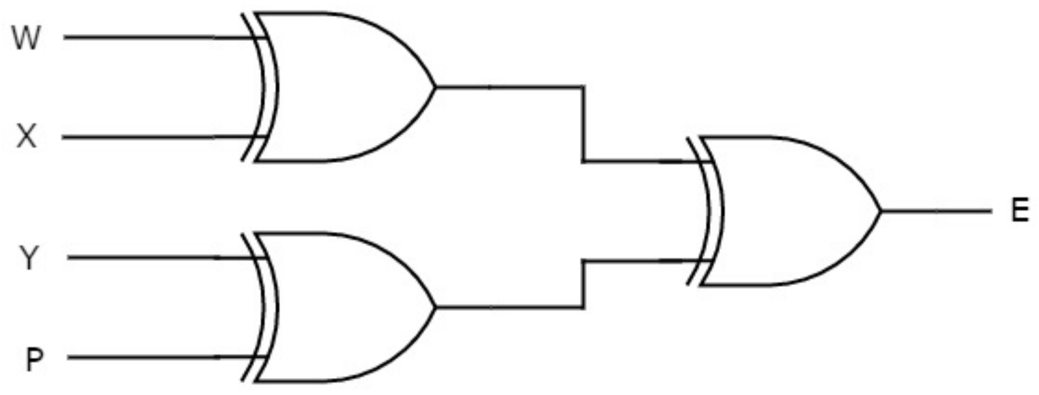
The following table shows the **Truth table** of an even parity checker.

4-bit Received Data WXYP	Even Parity Check bit E
0000	0
0001	1
0010	1
0011	0
0100	1
0101	0
0110	0
0111	1
1000	1
1001	0
1010	0
1011	1
1100	0
1101	1
1110	1
1111	0

From the above Truth table, we can observe that the even parity check bit value is ‘1’, when odd number of ones present in the received data. That means the Boolean function of even parity check bit is an **odd function**. Exclusive-OR function satisfies this condition. Hence, we can directly write the **Boolean function** of even parity check bit as

E=W \oplus X \oplus Y \oplus P

The following figure shows the **circuit diagram** of even parity checker.



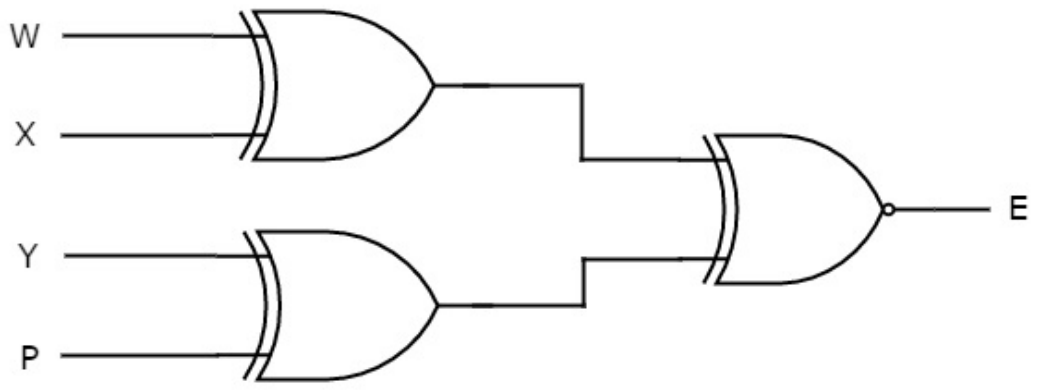
This circuit consists of three **Exclusive-OR gates** having two inputs each. The first level gates produce outputs of W \oplus X & Y \oplus P. The Exclusive-OR gate, which is in second level produces an output of W \oplus X \oplus Y \oplus P

Odd Parity Checker

Assume a 3-bit binary input, WXY is transmitted along with odd parity bit, P. So, the resultant word data contains 4 bits, which will be received as the input of odd parity checker.

It generates an **odd parity check bit, E**. This bit will be zero, if the received data contains an odd number of ones. That means, there is no error in the received data. This odd parity check bit will be one, if the received data contains even number of ones. That means, there is an error in the received data.

Follow the same procedure of an even parity checker for implementing an odd parity checker. The **circuit diagram** of odd parity checker is shown in the following figure.



The above circuit diagram consists of Ex-OR gates in first level and Ex-NOR gate in second level. Since the odd parity is just opposite to even parity, we can place an inverter at the output of even parity checker. In that case, the first, second and third levels contain two Ex-OR gates, one Ex-OR gate and one inverter respectively.

Binary Adder

The most basic arithmetic operation is addition. The circuit, which performs the addition of two binary numbers is known as **Binary adder**. First, let us implement an adder, which performs the addition of two bits.

Half Adder

Half adder is a combinational circuit, which performs the addition of two binary numbers A and B are of **single bit**. It produces two outputs sum, S & carry, C.

The **Truth table** of Half adder is shown below.

Inputs		Outputs	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

When we do the addition of two bits, the resultant sum can have the values ranging from 0 to 2 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent decimal digit 2 with single bit in binary. So, we require two bits for representing it in binary.

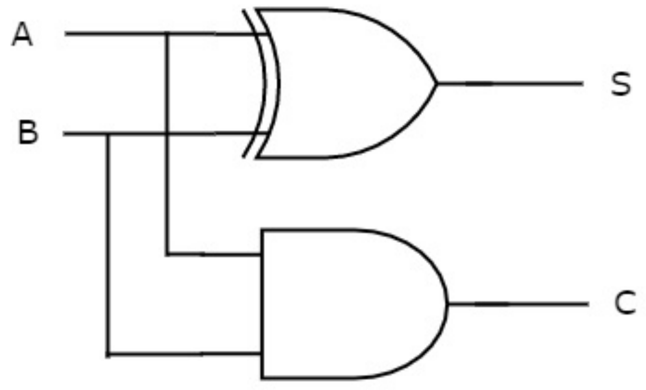
Let, sum, S is the Least significant bit and carry, C is the Most significant bit of the resultant sum. For first three combinations of inputs, carry, C is zero and the value of S will be either zero or one based on the **number of ones** present at the inputs. But, for last combination of inputs, carry, C is one and sum, S is zero, since the resultant sum is two.

From Truth table, we can directly write the **Boolean functions** for each output as

S=A \oplus B

C=AB

We can implement the above functions with 2-input Ex-OR gate & 2-input AND gate. The **circuit diagram** of Half adder is shown in the following figure.



In the above circuit, a two input Ex-OR gate & two input AND gate produces sum, S & carry, C respectively. Therefore, Half-adder performs the addition of two bits.

Full Adder

Full adder is a combinational circuit, which performs the **addition of three bits** A, B and C_{in}. Where, A & B are the two parallel significant bits and C_{in} is the carry bit, which is generated from previous stage. This Full adder also produces two outputs sum, S & carry, C_{out}, which are similar to Half adder.

The **Truth table** of Full adder is shown below.

Inputs			**Outputs**	
A	**B**	**C_{in}**	**C_{out}**	**S**
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

When we do the addition of three bits, the resultant sum can have the values ranging from 0 to 3 in decimal. We can represent the decimal digits 0 and 1 with single bit in binary. But, we can't represent the decimal digits 2 and 3 with single bit in binary. So, we require two bits for representing those two decimal digits in binary.

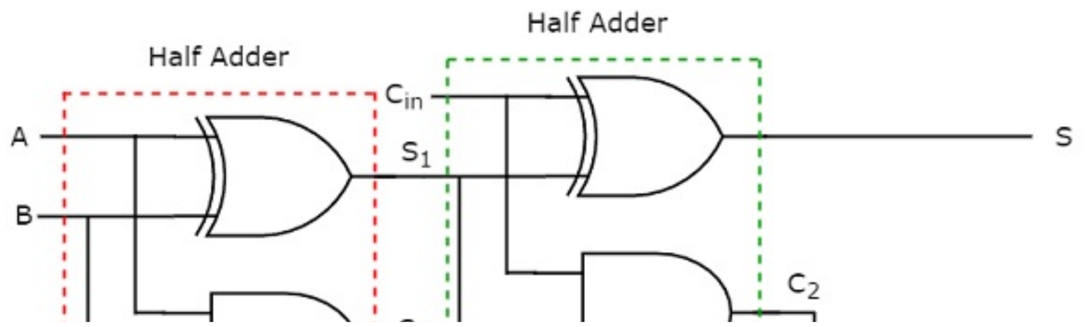
Let, sum, S is the Least significant bit and carry, C_{out} is the Most significant bit of resultant sum. It is easy to fill the values of outputs for all combinations of inputs in the truth table. Just count the **number of ones** present at the inputs and write the equivalent binary number at outputs. If C_{in} is equal to zero, then Full adder truth table is same as that of Half adder truth table.

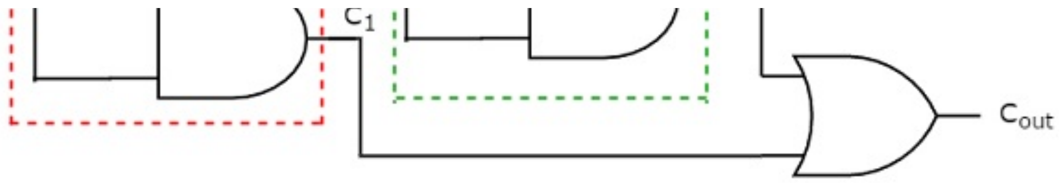
We will get the following **Boolean functions** for each output after simplification.

$S=A \oplus B \oplus C_{in}$

$c_{out} = AB + \left(A \oplus B \right)c_{in}$

The sum, S is equal to one, when odd number of ones present at the inputs. We know that Ex-OR gate produces an output, which is an odd function. So, we can use either two 2input Ex-OR gates or one 3-input Ex-OR gate in order to produce sum, S. We can implement carry, C_{out} using two 2-input AND gates & one OR gate. The **circuit diagram** of Full adder is shown in the following figure.





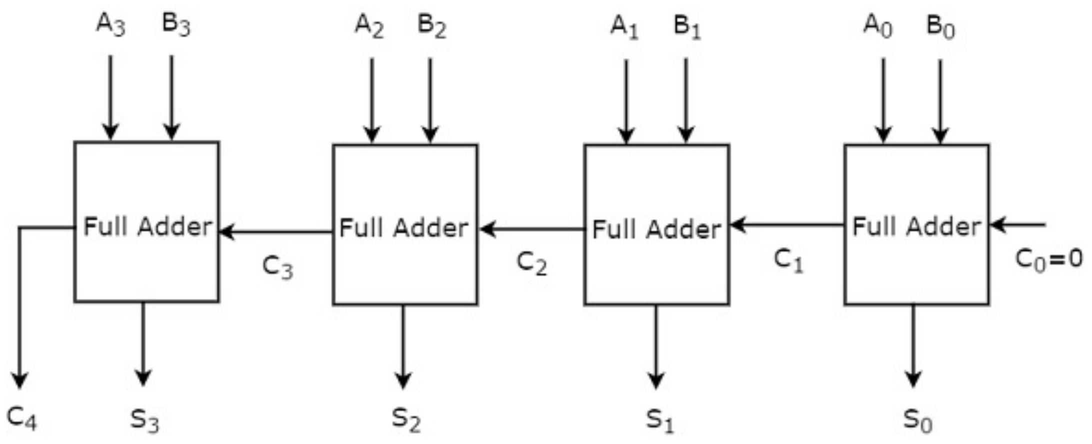
This adder is called as **Full adder** because for implementing one Full adder, we require two Half adders and one OR gate. If C_{in} is zero, then Full adder becomes Half adder. We can verify it easily from the above circuit diagram or from the Boolean functions of outputs of Full adder.

4-bit Binary Adder

The 4-bit binary adder performs the **addition of two 4-bit numbers**. Let the 4-bit binary numbers, A=A_{3}A_{2}A_{1}A_{0} and B= B_{3}B_{2}B_{1}B_{0}. We can implement 4-bit binary adder in one of the two following ways.

- Use one Half adder for doing the addition of two Least significant bits and three Full adders for doing the addition of three higher significant bits.
- Use four Full adders for uniformity. Since, initial carry C_{in} is zero, the Full adder which is used for adding the least significant bits becomes Half adder.

For the time being, we considered second approach. The **block diagram** of 4-bit binary adder is shown in the following figure.



Here, the 4 Full adders are cascaded. Each Full adder is getting the respective bits of two parallel inputs A & B. The carry output of one Full adder will be the carry input of subsequent higher order Full adder. This 4-bit binary adder produces the resultant sum having at most 5 bits. So, carry out of last stage Full adder will be the MSB.

In this way, we can implement any higher order binary adder just by cascading the required number of Full adders. This binary adder is also called as **ripple carry binary adder** because the carry propagates ripples from one stage to the next stage.

Binary Subtractor

The circuit, which performs the subtraction of two binary numbers is known as **Binary subtractor**. We can implement Binary subtractor in following two methods.

- Cascade Full subtractors
- 2’s complement method

In first method, we will get an n-bit binary subtractor by cascading ‘n’ Full subtractors. So, first you can implement Half subtractor and Full subtractor, similar to Half adder & Full adder. Then, you can implement an n-bit binary subtractor, by cascading ‘n’ Full subtractors. So, we will be having two separate circuits for binary addition and subtraction of two binary numbers.

In second method, we can use same binary adder for subtracting two binary numbers just by doing some modifications in the second input. So, internally binary addition operation takes place but, the output is resultant subtraction.

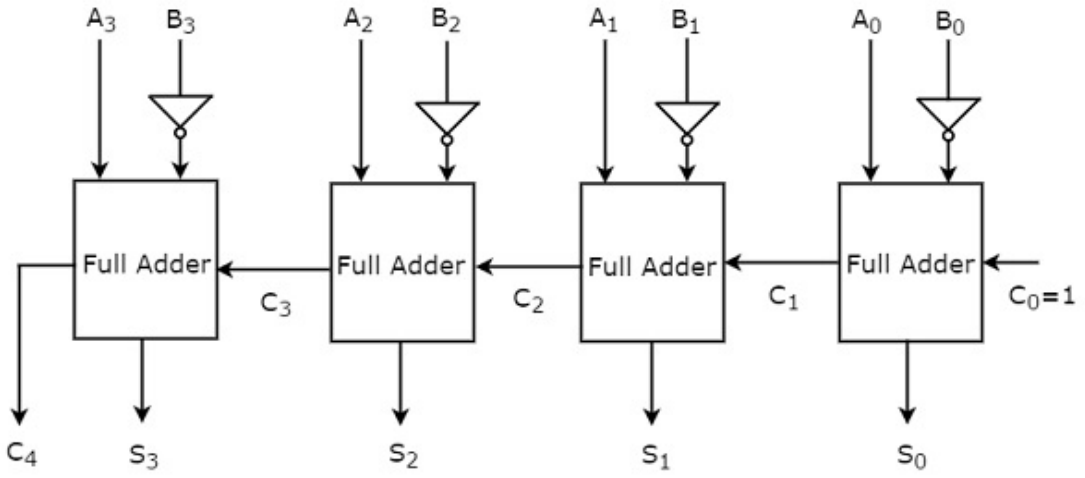
We know that the subtraction of two binary numbers A & B can be written as,

$A-B = A+\left({2}'s \: compliment \: of \: B \right)$

$\Rightarrow A-B = A+\left({1}'s \: compliment \: of \: B \right)+1$

4-bit Binary Subtractor

The 4-bit binary subtractor produces the **subtraction of two 4-bit numbers**. Let the 4bit binary numbers, $A=A_{3}A_{2}A_{1}A_{0}$ and $B= B_{3}B_{2}B_{1}B_{0}$. Internally, the operation of 4-bit Binary subtractor is similar to that of 4-bit Binary adder. If the normal bits of binary number A, complemented bits of binary number B and initial carry borrow, C_{in} as one are applied to 4-bit Binary adder, then it becomes 4-bit Binary subtractor. The **block diagram** of 4-bit binary subtractor is shown in the following figure.



This 4-bit binary subtractor produces an output, which is having at most 5 bits. If Binary number A is greater than Binary number B, then MSB of the output is zero and the remaining bits hold the magnitude of A-B. If Binary number A is less than Binary number B, then MSB of the output is one. So, take the 2’s complement of output in order to get the magnitude of A-B.

In this way, we can implement any higher order binary subtractor just by cascading the required number of Full adders with necessary modifications.

Binary Adder / Subtractor

The circuit, which can be used to perform either addition or subtraction of two binary numbers at any time is known as **Binary Adder / subtractor**. Both, Binary adder and Binary subtractor contain a set of Full adders, which are cascaded. The input bits of binary number A are directly applied in both Binary adder and Binary subtractor.

There are two differences in the inputs of Full adders that are present in Binary adder and Binary subtractor.

- The input bits of binary number B are directly applied to Full adders in Binary adder, whereas the complemented bits of binary number B are applied to Full adders in Binary subtractor.
- The initial carry, $C_0 = 0$ is applied in 4-bit Binary adder, whereas the initial carry borrow, $C_0 = 1$ is applied in 4-bit Binary subtractor.

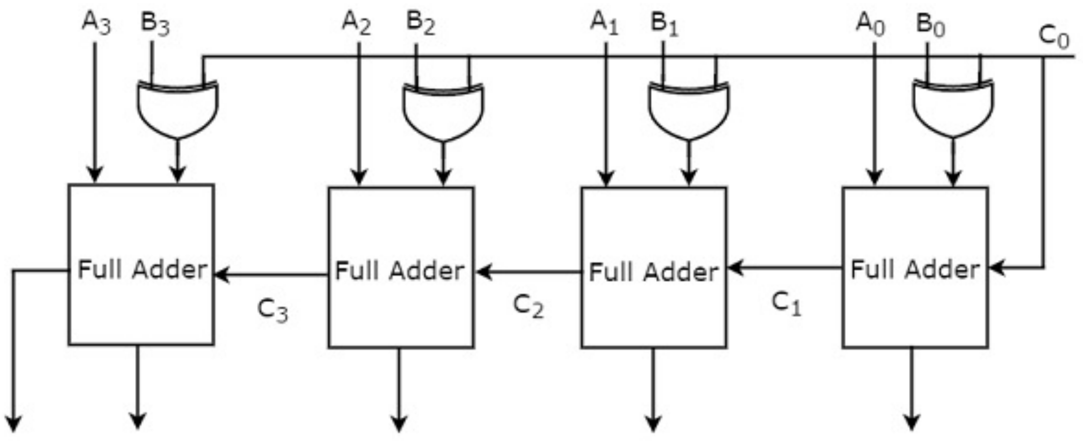
We know that a **2-input Ex-OR gate** produces an output, which is same as that of first input when other input is zero. Similarly, it produces an output, which is complement of first input when other input is one.

Therefore, we can apply the input bits of binary number B, to 2-input Ex-OR gates. The other input to all these Ex-OR gates is C_0. So, based on the value of C_0, the Ex-OR gates produce either the normal or complemented bits of binary number B.

4-bit Binary Adder / Subtractor

The 4-bit binary adder / subtractor produces either the addition or the subtraction of two 4-bit numbers based on the value of initial carry or borrow, C_0. Let the 4-bit binary numbers, A=A_{3}A_{2}A_{1}A_{0} and B= B_{3}B_{2}B_{1}B_{0}. The operation of 4-bit Binary adder / subtractor is similar to that of 4-bit Binary adder and 4-bit Binary subtractor.

Apply the normal bits of binary numbers A and B & initial carry or borrow, C_0 from externally to a 4-bit binary adder. The **block diagram** of 4-bit binary adder / subtractor is shown in the following figure.



If initial carry, C_0 is zero, then each full adder gets the normal bits of binary numbers A & B. So, the 4-bit binary adder / subtractor produces an output, which is the **addition of two binary numbers** A & B.

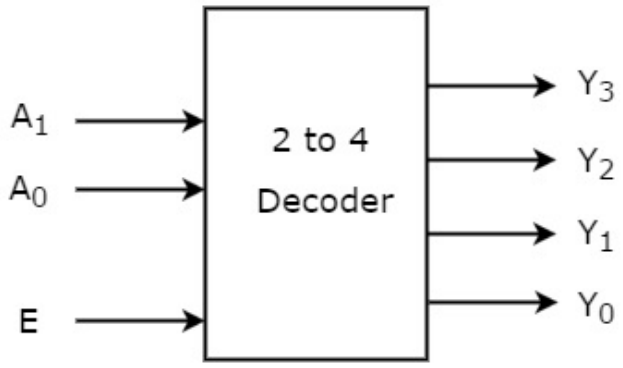
If initial borrow, C_0 is one, then each full adder gets the normal bits of binary number A & complemented bits of binary number B. So, the 4-bit binary adder / subtractor produces an output, which is the **subtraction of two binary numbers** A & B.

Therefore, with the help of additional Ex-OR gates, the same circuit can be used for both addition and subtraction of two binary numbers.

Decoder is a combinational circuit that has 'n' input lines and maximum of 2^n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing but the **min terms** of 'n' input variables lines, when it is enabled.

2 to 4 Decoder

Let 2 to 4 Decoder has two inputs A_1 & A_0 and four outputs Y_3, Y_2, Y_1 & Y_0. The **block diagram** of 2 to 4 decoder is shown in the following figure.



One of these four outputs will be '1' for each combination of inputs when enable, E is '1'. The **Truth table** of 2 to 4 decoder is shown below.

Enable	**Inputs**		**Outputs**			
E	**A_1**	**A_0**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

From Truth table, we can write the **Boolean functions** for each output as

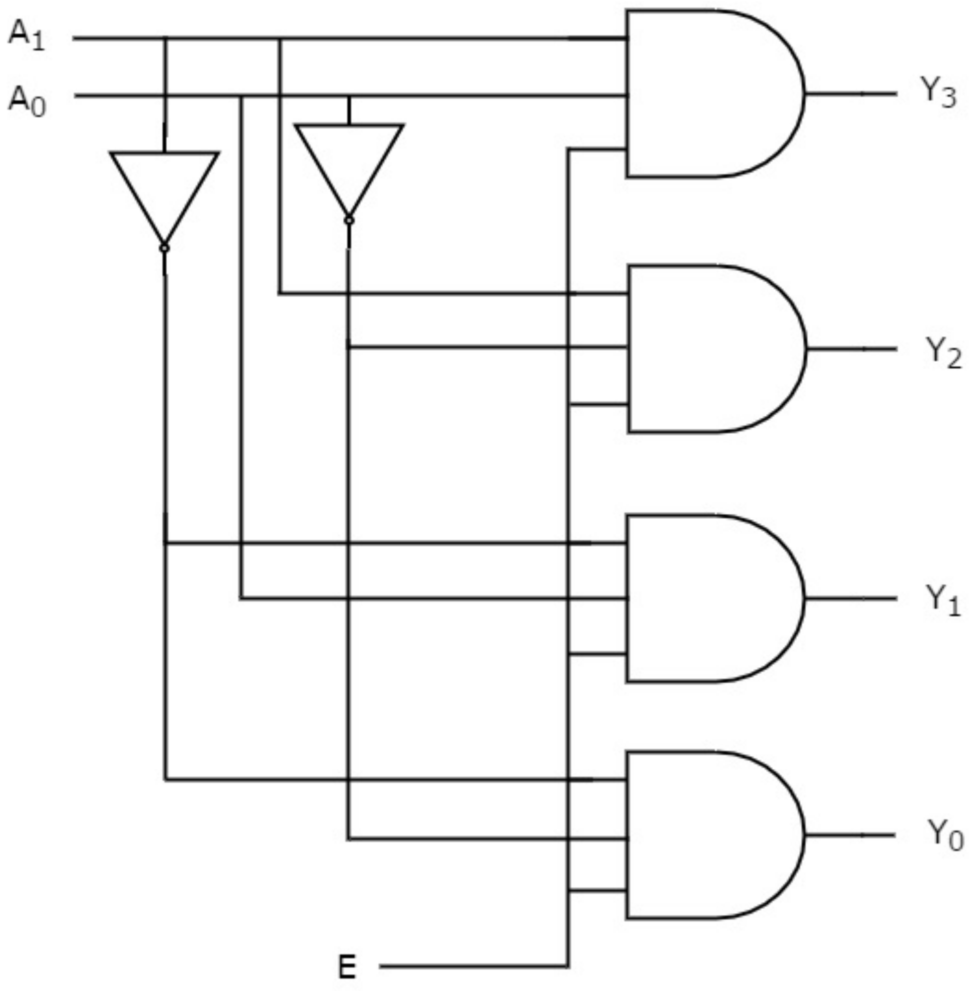
Y_{3}=E.A_{1}.A_{0}

Y_{2}=E.A_{1}.{A_{0}}'

Y_{1}=E.{A_{1}}'.A_{0}

Y_{0}=E.{A_{1}}'.{A_{0}}'

Each output is having one product term. So, there are four product terms in total. We can implement these four product terms by using four AND gates having three inputs each & two inverters. The **circuit diagram** of 2 to 4 decoder is shown in the following figure.



Therefore, the outputs of 2 to 4 decoder are nothing but the **min terms** of two input variables A_1 & A_0, when enable, E is equal to one. If enable, E is zero, then all the outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces eight min terms of three input variables A_2, A_1 & A_0 and 4 to 16 decoder produces sixteen min terms of four input variables A_3, A_2, A_1 & A_0.

Implementation of Higher-order Decoders

Now, let us implement the following two higher-order decoders using lower-order decoders.

- 3 to 8 decoder
- 4 to 16 decoder

3 to 8 Decoder

In this section, let us implement **3 to 8 decoder using 2 to 4 decoders**. We know that 2 to 4 Decoder has two inputs, A_1 & A_0 and four outputs, Y_3 to Y_0. Whereas, 3 to 8 Decoder has three inputs A_2, A_1 & A_0 and eight outputs, Y_7 to Y_0.

We can find the number of lower order decoders required for implementing higher order decoder using the following formula.

$Required \: number \: of \: lower \: order \: decoders=\frac{m_{2}}{m_{1}}$

Where,

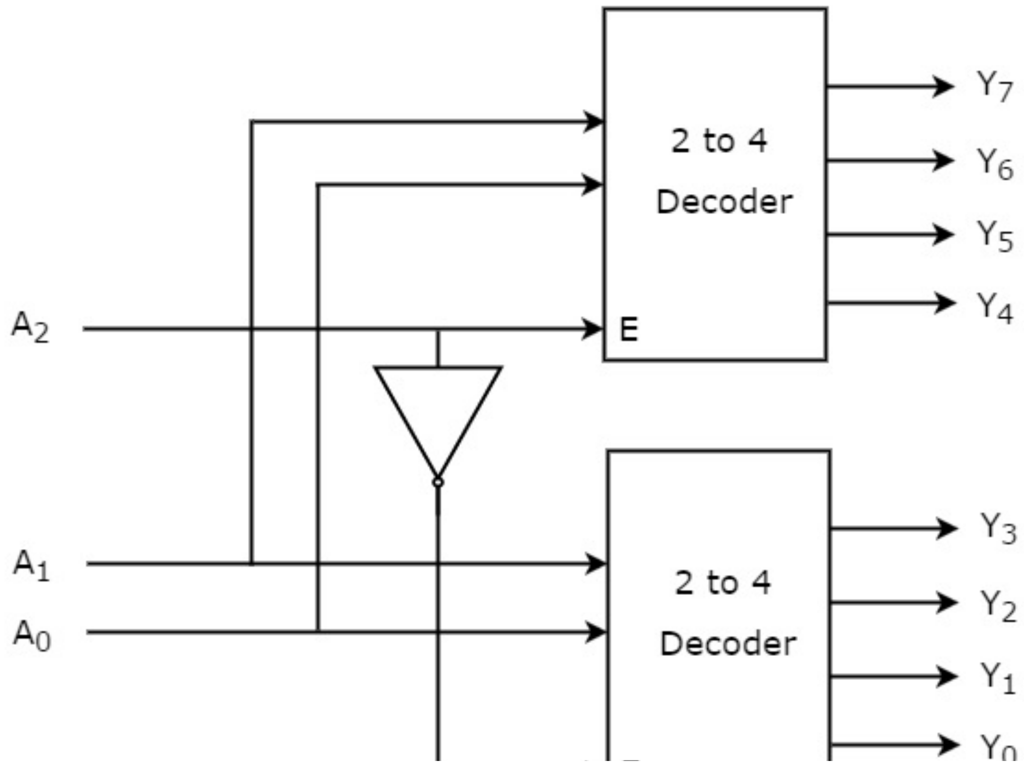
m_{1} is the number of outputs of lower order decoder.

m_{2} is the number of outputs of higher order decoder.

Here, m_{1} = 4 and m_{2} = 8. Substitute, these two values in the above formula.

$Required \: number \: of \: 2 \: to \: 4 \: decoders=\frac{8}{4}=2$

Therefore, we require two 2 to 4 decoders for implementing one 3 to 8 decoder. The **block diagram** of 3 to 8 decoder using 2 to 4 decoders is shown in the following figure.



The parallel inputs A_1 & A_0 are applied to each 2 to 4 decoder. The complement of input A_2 is connected to Enable, E of lower 2 to 4 decoder in order to get the outputs, Y_3 to Y_0. These are the **lower four min terms**. The input, A_2 is directly connected to Enable, E of upper 2 to 4 decoder in order to get the outputs, Y_7 to Y_4. These are the **higher four min terms**.

4 to 16 Decoder

In this section, let us implement **4 to 16 decoder using 3 to 8 decoders**. We know that 3 to 8 Decoder has three inputs A_2, A_1 & A_0 and eight outputs, Y_7 to Y_0. Whereas, 4 to 16 Decoder has four inputs A_3, A_2, A_1 & A_0 and sixteen outputs, Y_{15} to Y_0

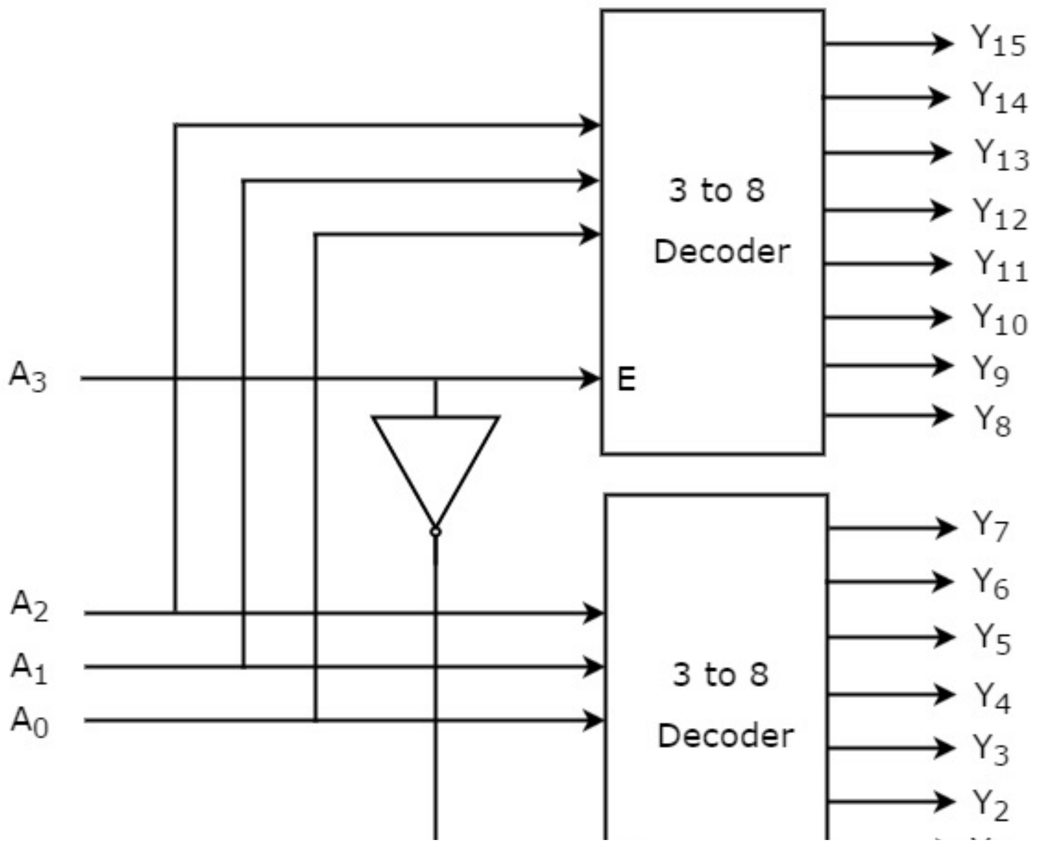
We know the following formula for finding the number of lower order decoders required.

Required \: number \: of \: lower \: order \: decoders=\frac{m_{2}}{m_{1}}

Substitute, m_{1} = 8 and m_{2} = 16 in the above formula.

Required \: number \: of \: 3 \: to \: 8 decoders=\frac{16}{8}=2

Therefore, we require two 3 to 8 decoders for implementing one 4 to 16 decoder. The **block diagram** of 4 to 16 decoder using 3 to 8 decoders is shown in the following figure.

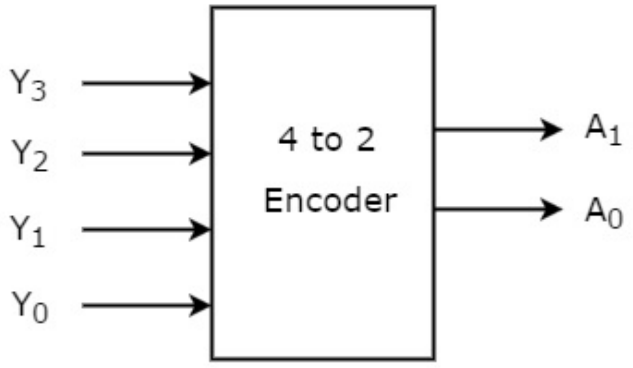


The parallel inputs A_2, A_1 & A_0 are applied to each 3 to 8 decoder. The complement of input, A3 is connected to Enable, E of lower 3 to 8 decoder in order to get the outputs, Y_7 to Y_0. These are the **lower eight min terms**. The input, A_3 is directly connected to Enable, E of upper 3 to 8 decoder in order to get the outputs, Y_{15} to Y_8. These are the **higher eight min terms**.

An **Encoder** is a combinational circuit that performs the reverse operation of Decoder. It has maximum of 2^n input lines and 'n' output lines. It will produce a binary code equivalent to the input, which is active High. Therefore, the encoder encodes 2^n input lines with 'n' bits. It is optional to represent the enable signal in encoders.

4 to 2 Encoder

Let 4 to 2 Encoder has four inputs Y_3, Y_2, Y_1 & Y_0 and two outputs A_1 & A_0. The **block diagram** of 4 to 2 Encoder is shown in the following figure.



At any time, only one of these 4 inputs can be '1' in order to get the respective binary code at the output. The **Truth table** of 4 to 2 encoder is shown below.

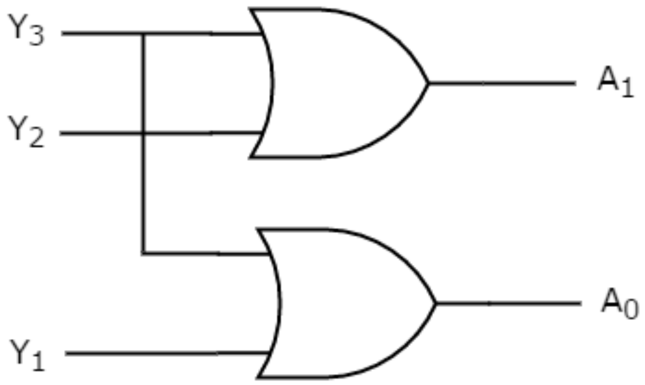
Inputs				**Outputs**	
Y_3	Y_2	Y_1	Y_0	A_1	A_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

From Truth table, we can write the **Boolean functions** for each output as

A_{1}=Y_{3}+Y_{2}

A_{0}=Y_{3}+Y_{1}

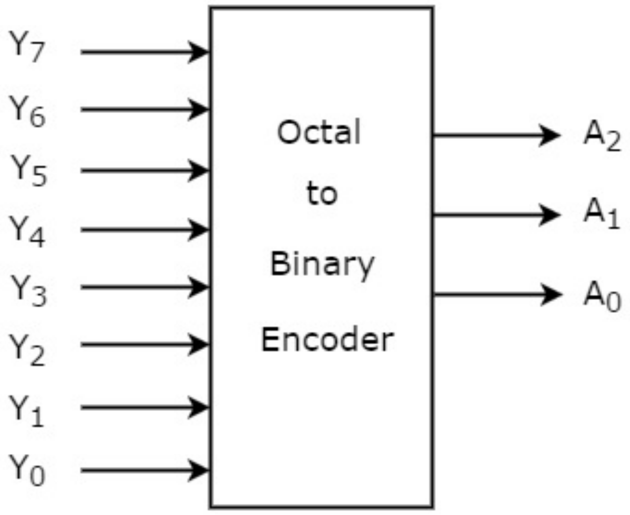
We can implement the above two Boolean functions by using two input OR gates. The **circuit diagram** of 4 to 2 encoder is shown in the following figure.



The above circuit diagram contains two OR gates. These OR gates encode the four inputs with two bits

Octal to Binary Encoder

Octal to binary Encoder has eight inputs, Y_7 to Y_0 and three outputs A_2, A_1 & A_0. Octal to binary encoder is nothing but 8 to 3 encoder. The **block diagram** of octal to binary Encoder is shown in the following figure.



At any time, only one of these eight inputs can be ‘1’ in order to get the respective binary code. The **Truth table** of octal to binary encoder is shown below.

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

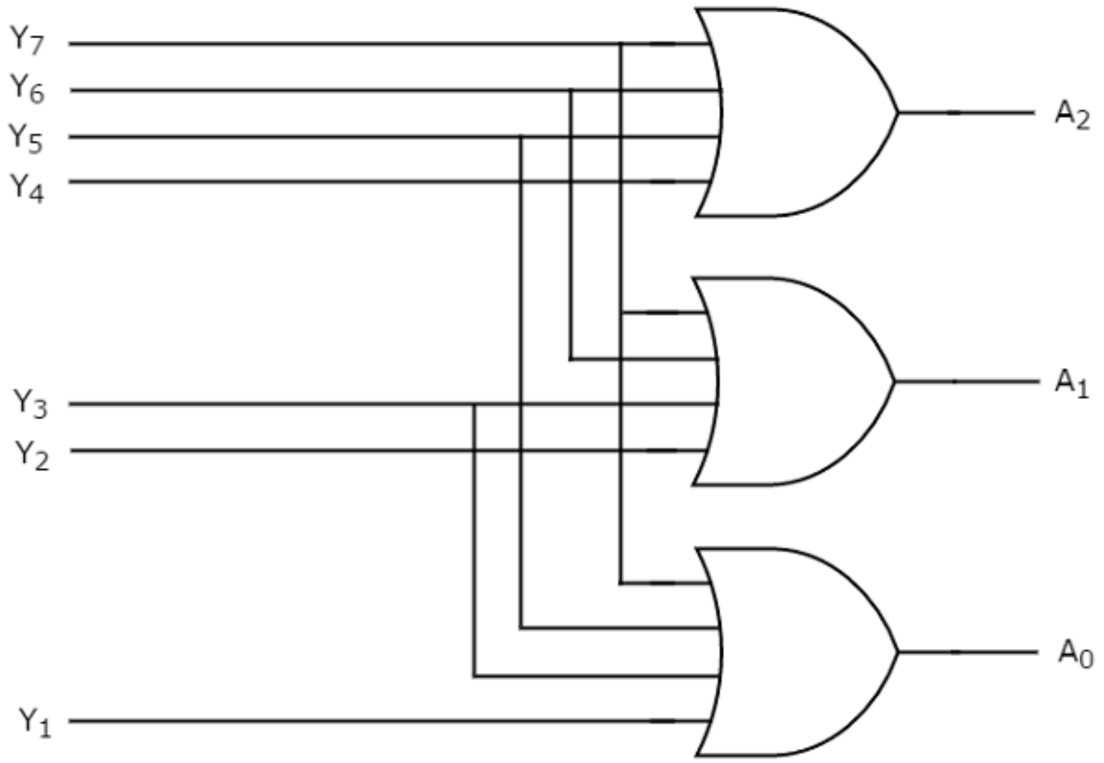
From Truth table, we can write the **Boolean functions** for each output as

A_{2}=Y_{7}+Y_{6}+Y_{5}+Y_{4}

A_{1}=Y_{7}+Y_{6}+Y_{3}+Y_{2}

A_{0}=Y_{7}+Y_{5}+Y_{3}+Y_{1}

We can implement the above Boolean functions by using four input OR gates. The **circuit diagram** of octal to binary encoder is shown in the following figure.



The above circuit diagram contains three 4-input OR gates. These OR gates encode the eight inputs with three bits.

Drawbacks of Encoder

Following are the drawbacks of normal encoder.

- There is an ambiguity, when all outputs of encoder are equal to zero.

Because, it could be the code corresponding to the inputs, when only least significant input is one or when all inputs are zero.

- If more than one input is active High, then the encoder produces an output, which may not be the correct code. For **example**, if both Y_3 and Y_6 are '1', then the encoder produces 111 at the output. This is neither equivalent code corresponding to Y_3, when it is '1' nor the equivalent code corresponding to Y_6, when it is '1'.

So, to overcome these difficulties, we should assign priorities to each input of encoder. Then, the output of encoder will be the binary code corresponding to the active High inputs, which has higher priority. This encoder is called as **priority encoder**.

Priority Encoder

A 4 to 2 priority encoder has four inputs Y_3, Y_2, Y_1 & Y_0 and two outputs A_1 & A_0. Here, the input, Y_3 has the highest priority, whereas the input, Y_0 has the lowest priority. In this case, even if more than one input is '1' at the same time, the output will be the binary code corresponding to the input, which is having **higher priority**.

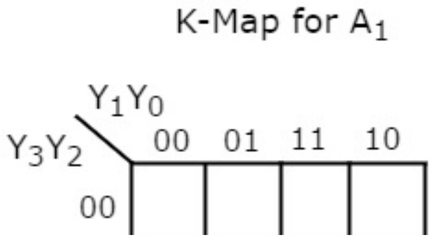
We considered one more **output, V** in order to know, whether the code available at outputs is valid or not.

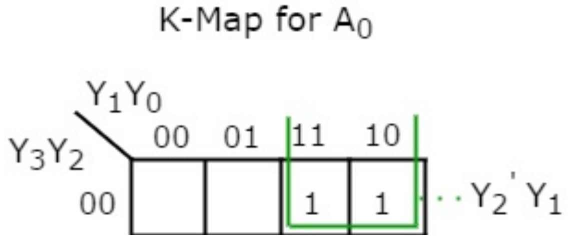
- If at least one input of the encoder is '1', then the code available at outputs is a valid one. In this case, the output, V will be equal to 1.
- If all the inputs of encoder are '0', then the code available at outputs is not a valid one. In this case, the output, V will be equal to 0.

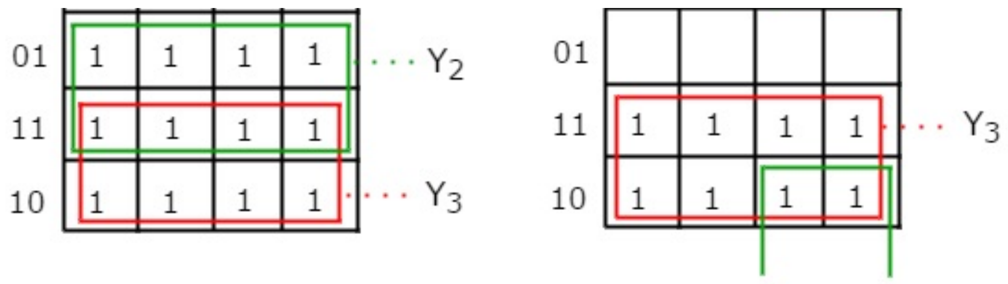
The **Truth table** of 4 to 2 priority encoder is shown below.

Inputs				**Outputs**		
Y_3	Y_2	Y_1	Y_0	A_1	A_0	V
0	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

Use **4 variable K-maps** for getting simplified expressions for each output.







The simplified **Boolean functions** are

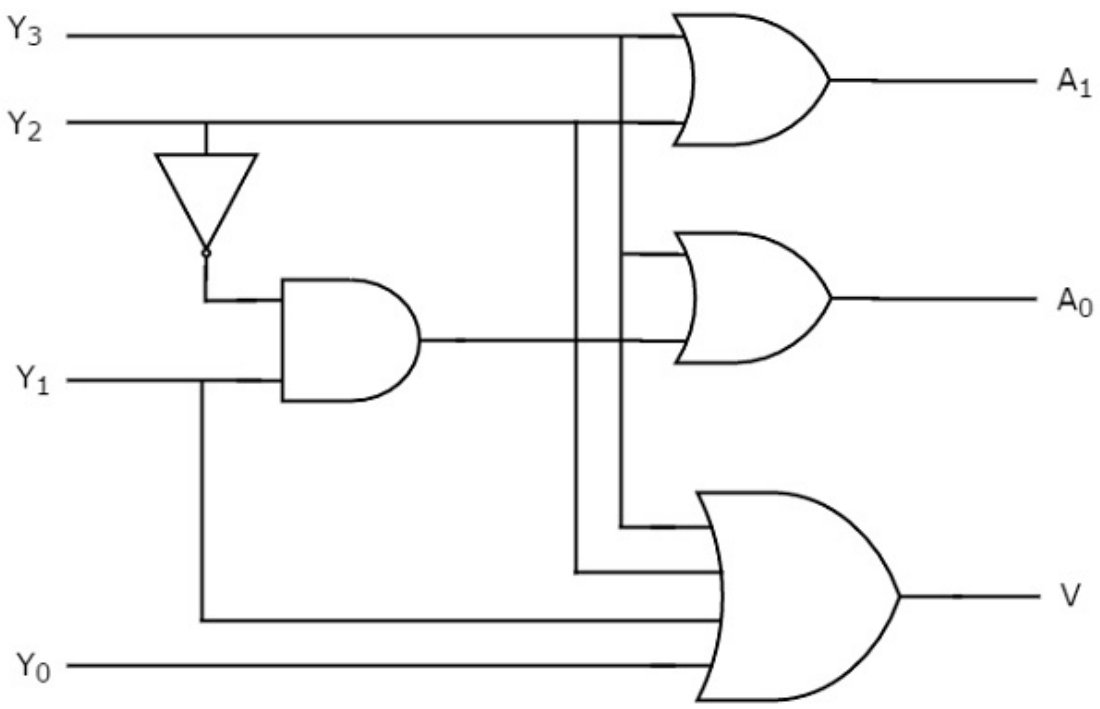
A_{1}=Y_{3}+Y_{2}

A_{0}=Y_{3}+{Y_{2}}'Y_{1}

Similarly, we will get the Boolean function of output, V as

V=Y_{3}+Y_{2}+Y_{1}+Y_{0}

We can implement the above Boolean functions using logic gates. The **circuit diagram** of 4 to 2 priority encoder is shown in the following figure.



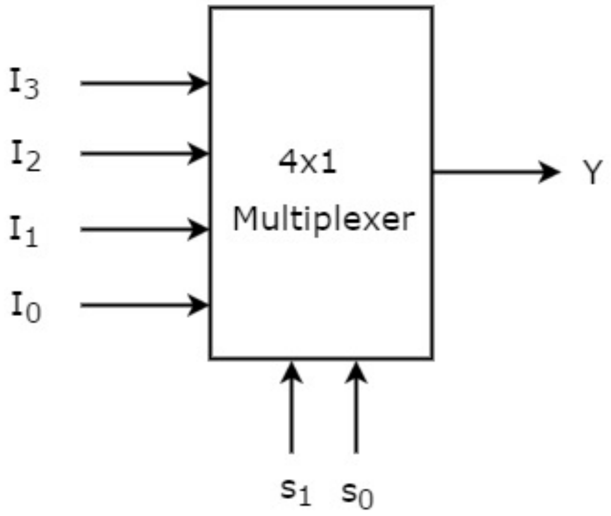
The above circuit diagram contains two 2-input OR gates, one 4-input OR gate, one 2input AND gate & an inverter. Here AND gate & inverter combination are used for producing a valid code at the outputs, even when multiple inputs are equal to ‘1’ at the same time. Hence, this circuit encodes the four inputs with two bits based on the **priority** assigned to each input.

Multiplexer is a combinational circuit that has maximum of 2^n data inputs, ‘n’ selection lines and single output line. One of these data inputs will be connected to the output based on the values of selection lines.

Since there are ‘n’ selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination will select only one data input. Multiplexer is also called as **Mux**.

4x1 Multiplexer

4x1 Multiplexer has four data inputs I_3, I_2, I_1 & I_0, two selection lines s_1 & s_0 and one output Y. The **block diagram** of 4x1 Multiplexer is shown in the following figure.



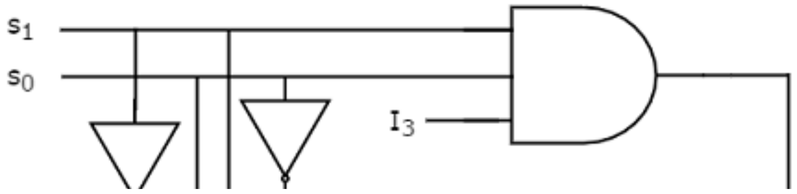
One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. **Truth table** of 4x1 Multiplexer is shown below.

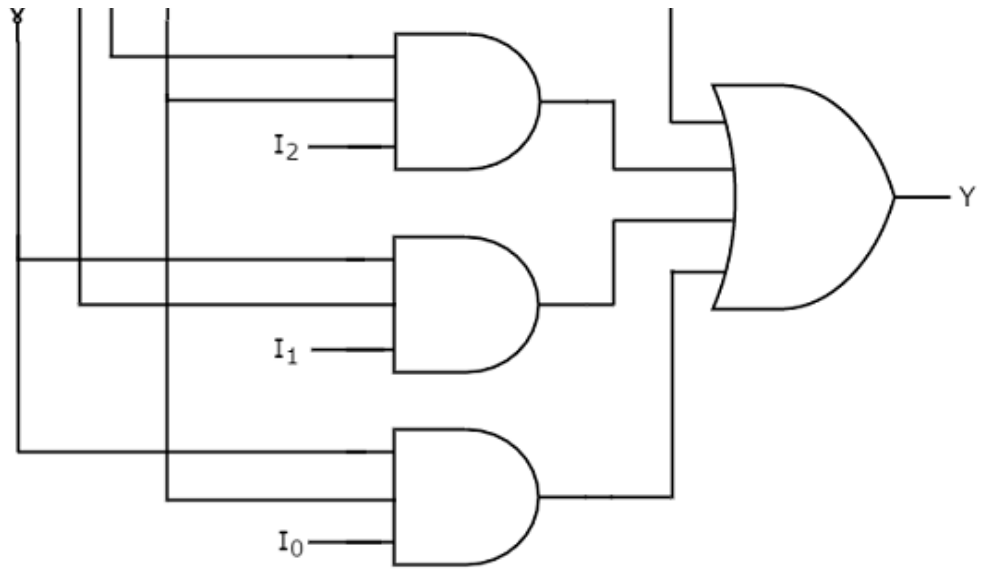
Selection Lines		**Output**
S_1	**S_0**	**Y**
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

From Truth table, we can directly write the **Boolean function** for output, Y as

Y={S_{1}}'{S_{0}}'I_{0}+{S_{1}}'S_{0}I_{1}+S_{1}{S_{0}}'I_{2}+S_{1}S_{0}I_{3}

We can implement this Boolean function using Inverters, AND gates & OR gate. The **circuit diagram** of 4x1 multiplexer is shown in the following figure.





We can easily understand the operation of the above circuit. Similarly, you can implement 8x1 Multiplexer and 16x1 multiplexer by following the same procedure.

Implementation of Higher-order Multiplexers.

Now, let us implement the following two higher-order Multiplexers using lower-order Multiplexers.

- 8x1 Multiplexer
- 16x1 Multiplexer

8x1 Multiplexer

In this section, let us implement 8x1 Multiplexer using 4x1 Multiplexers and 2x1 Multiplexer. We know that 4x1 Multiplexer has 4 data inputs, 2 selection lines and one output. Whereas, 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output.

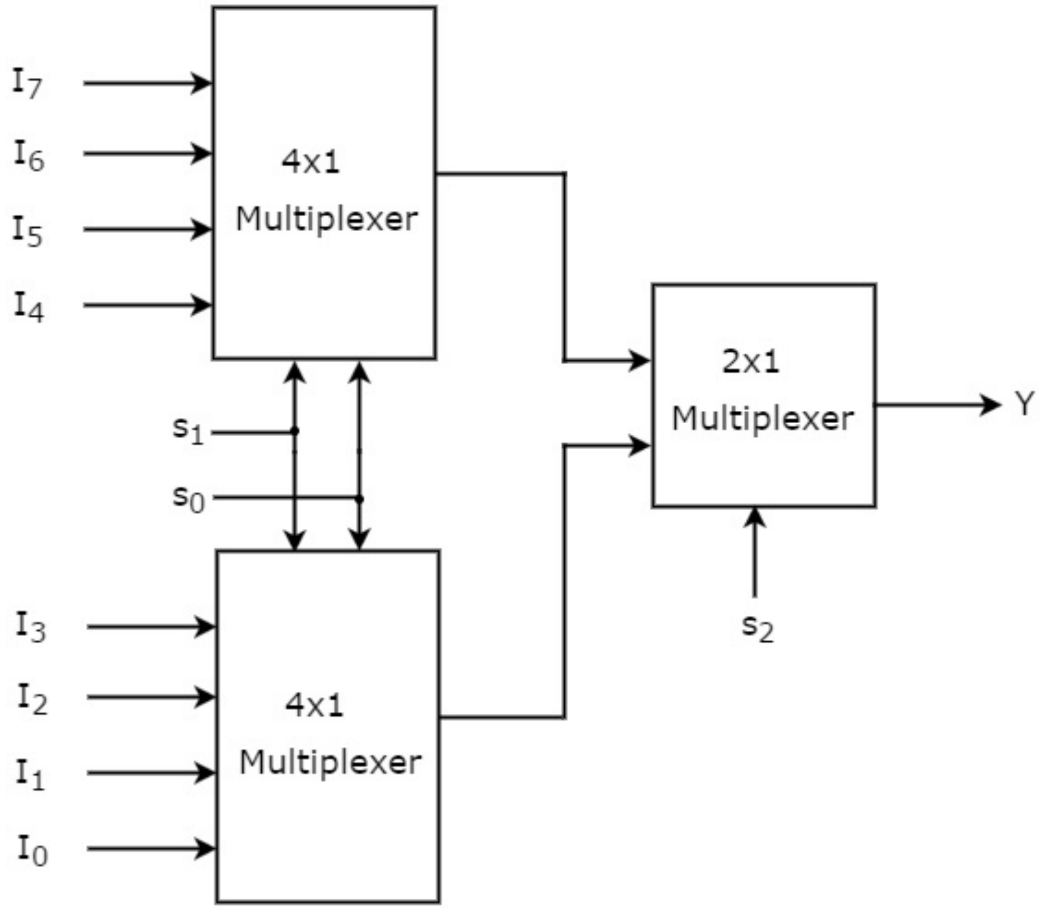
So, we require two **4x1 Multiplexers** in first stage in order to get the 8 data inputs. Since, each 4x1 Multiplexer produces one output, we require a **2x1 Multiplexer** in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 8x1 Multiplexer has eight data inputs I_7 to I_0, three selection lines s_2, s_1 & s0 and one output Y. The **Truth table** of 8x1 Multiplexer is shown below.

Selection Inputs			**Output**
S_2	**S_1**	**S_0**	**Y**
0	0	0	I_0
0	0	1	I_1

0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

We can implement 8x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 8x1 Multiplexer is shown in the following figure.



The same **selection lines, s_1 & s_0** are applied to both 4x1 Multiplexers. The data inputs of upper 4x1 Multiplexer are I_7 to I_4 and the data inputs of lower 4x1 Multiplexer are I_3 to I_0. Therefore, each 4x1 Multiplexer produces an output based on the values of selection lines, s_1 & s_0.

The outputs of first stage 4x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_2** is applied to 2x1 Multiplexer.

- If s_2 is zero, then the output of 2x1 Multiplexer will be one of the 4 inputs I_3 to I_0 based on the values of selection lines s_1 & s_0.
- If s_2 is one, then the output of 2x1 Multiplexer will be one of the 4 inputs I_7 to I_4 based on the values of selection lines s_1 & s_0.

Therefore, the overall combination of two 4x1 Multiplexers and one 2x1 Multiplexer performs as one 8x1 Multiplexer.

16x1 Multiplexer

In this section, let us implement 16x1 Multiplexer using 8x1 Multiplexers and 2x1 Multiplexer. We know that 8x1 Multiplexer has 8 data inputs, 3 selection lines and one output. Whereas, 16x1 Multiplexer has 16 data inputs, 4 selection lines and one output.

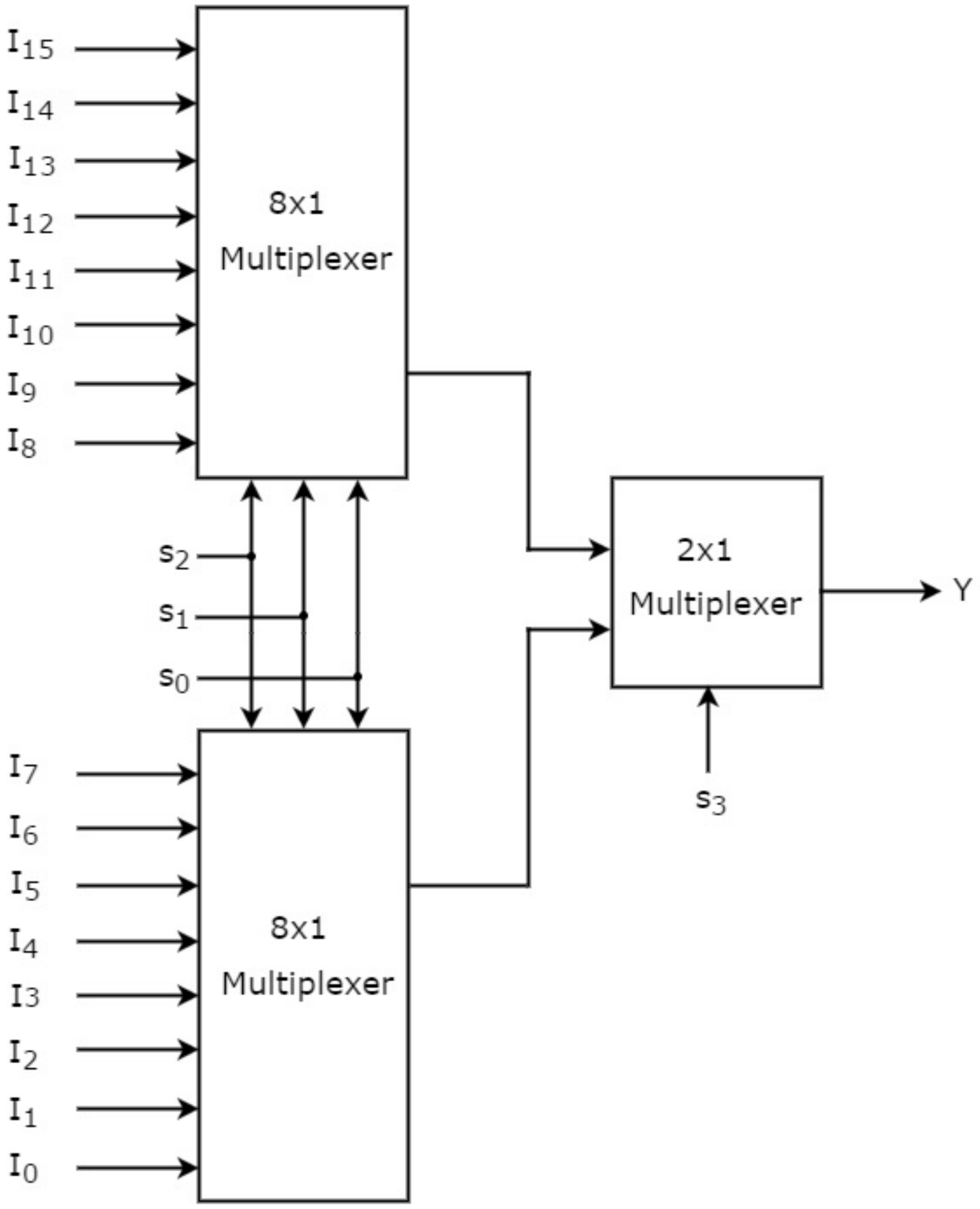
So, we require two **8x1 Multiplexers** in first stage in order to get the 16 data inputs. Since, each 8x1 Multiplexer produces one output, we require a 2x1 Multiplexer in second stage by considering the outputs of first stage as inputs and to produce the final output.

Let the 16x1 Multiplexer has sixteen data inputs I_{15} to I_0, four selection lines s_3 to s_0 and one output Y. The **Truth table** of 16x1 Multiplexer is shown below.

Selection Inputs				Output
S_3	S_2	S_1	S_0	Y
0	0	0	0	I_0
0	0	0	1	I_1
0	0	1	0	I_2
0	0	1	1	I_3
0	1	0	0	I_4
0	1	0	1	I_5
0	1	1	0	I_6
0	1	1	1	I_7
1	0	0	0	I_8
1	0	0	1	I_9
1	0	1	0	I_{10}
1	0	1	1	I_{11}
1	1	0	0	I_{12}

1	1	0	1	I_{13}
1	1	1	0	I_{14}
1	1	1	1	I_{15}

We can implement 16x1 Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 16x1 Multiplexer is shown in the following figure.



The **same selection lines, s_2, s_1 & s_0** are applied to both 8x1 Multiplexers. The data inputs of upper 8x1 Multiplexer are I_{15} to I_8 and the data inputs of lower 8x1 Multiplexer are I_7 to I_0. Therefore, each 8x1 Multiplexer produces an output based on the values of selection lines, s_2, s_1 & s_0.

The outputs of first stage 8x1 Multiplexers are applied as inputs of 2x1 Multiplexer that is present in second stage. The other **selection line, s_3** is applied to 2x1 Multiplexer.

- If s_3 is zero, then the output of 2x1 Multiplexer will be one of the 8 inputs Is_7 to I_0 based on the values of selection lines s_2, s_1 & s_0.
- If s_3 is one, then the output of 2x1 Multiplexer will be one of the 8 inputs I_{15} to I_8 based on the values of selection lines s_2, s_1 & s_0.

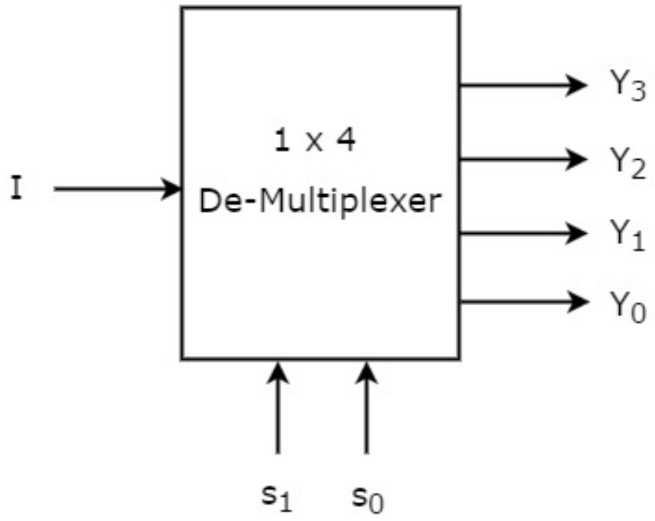
Therefore, the overall combination of two 8x1 Multiplexers and one 2x1 Multiplexer performs as one 16x1 Multiplexer.

De-Multiplexer is a combinational circuit that performs the reverse operation of Multiplexer. It has single input, ‘n’ selection lines and maximum of 2^n outputs. The input will be connected to one of these outputs based on the values of selection lines.

Since there are ‘n’ selection lines, there will be 2^n possible combinations of zeros and ones. So, each combination can select only one output. De-Multiplexer is also called as **De-Mux**.

1x4 De-Multiplexer

1x4 De-Multiplexer has one input I, two selection lines, s_1 & s_0 and four outputs Y_3, Y_2, Y_1 &Y_0. The **block diagram** of 1x4 De-Multiplexer is shown in the following figure.



The single input ‘I’ will be connected to one of the four outputs, Y_3 to Y_0 based on the values of selection lines s_1 & s0. The **Truth table** of 1x4 De-Multiplexer is shown below.

Selection Inputs		Outputs			
S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	**I**
0	1	0	0	**I**	0
1	0	0	**I**	0	0
1	1	**I**	0	0	0

From the above Truth table, we can directly write the **Boolean functions** for each output as

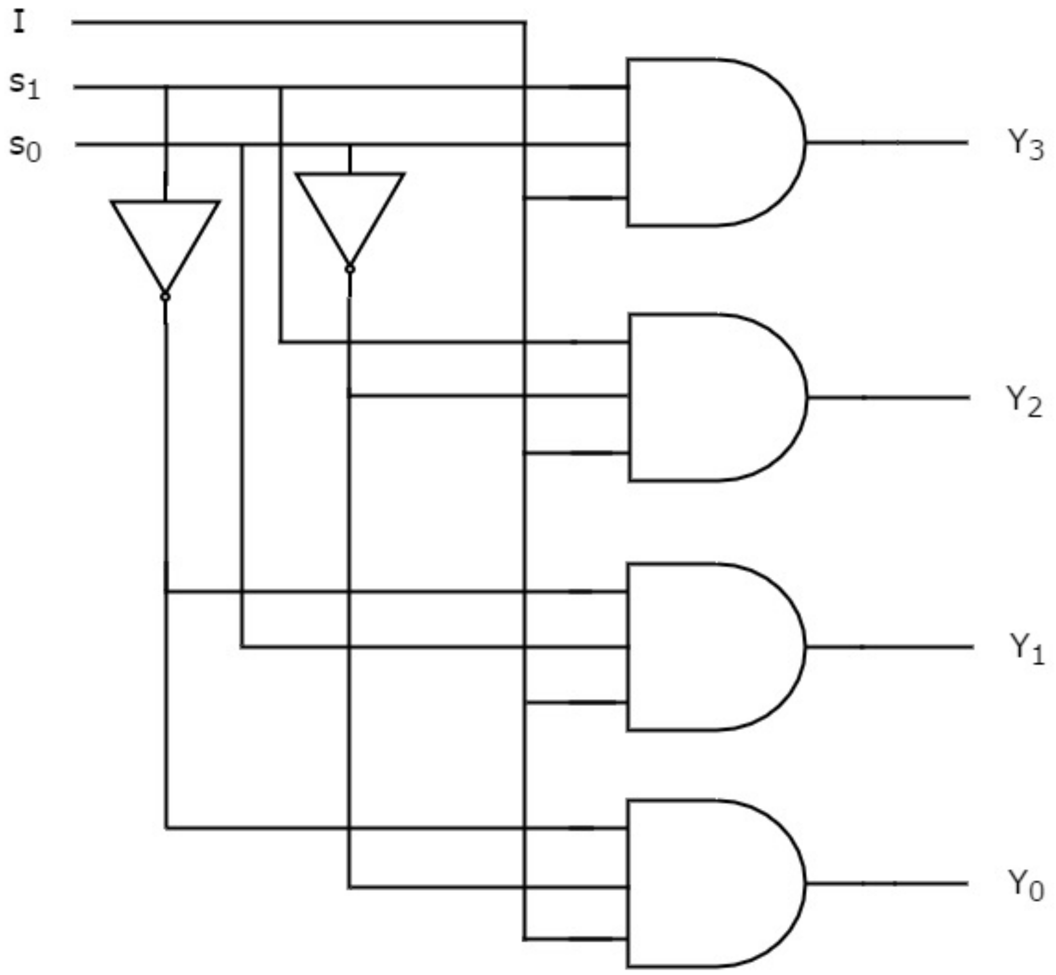
Y_{3}=s_{1}s_{0}I

Y_{2}=s_{1}{s_{0}}'I

Y_{1}={s_{1}}'s_{0}I

Y_{0}={s_1}'{s_{0}}'I

We can implement these Boolean functions using Inverters & 3-input AND gates. The **circuit diagram** of 1x4 De-Multiplexer is shown in the following figure.



We can easily understand the operation of the above circuit. Similarly, you can implement 1x8 De-Multiplexer and 1x16 De-Multiplexer by following the same procedure.

Implementation of Higher-order De-Multiplexers

Now, let us implement the following two higher-order De-Multiplexers using lower-order De-Multiplexers.

- 1x8 De-Multiplexer
- 1x16 De-Multiplexer

1x8 De-Multiplexer

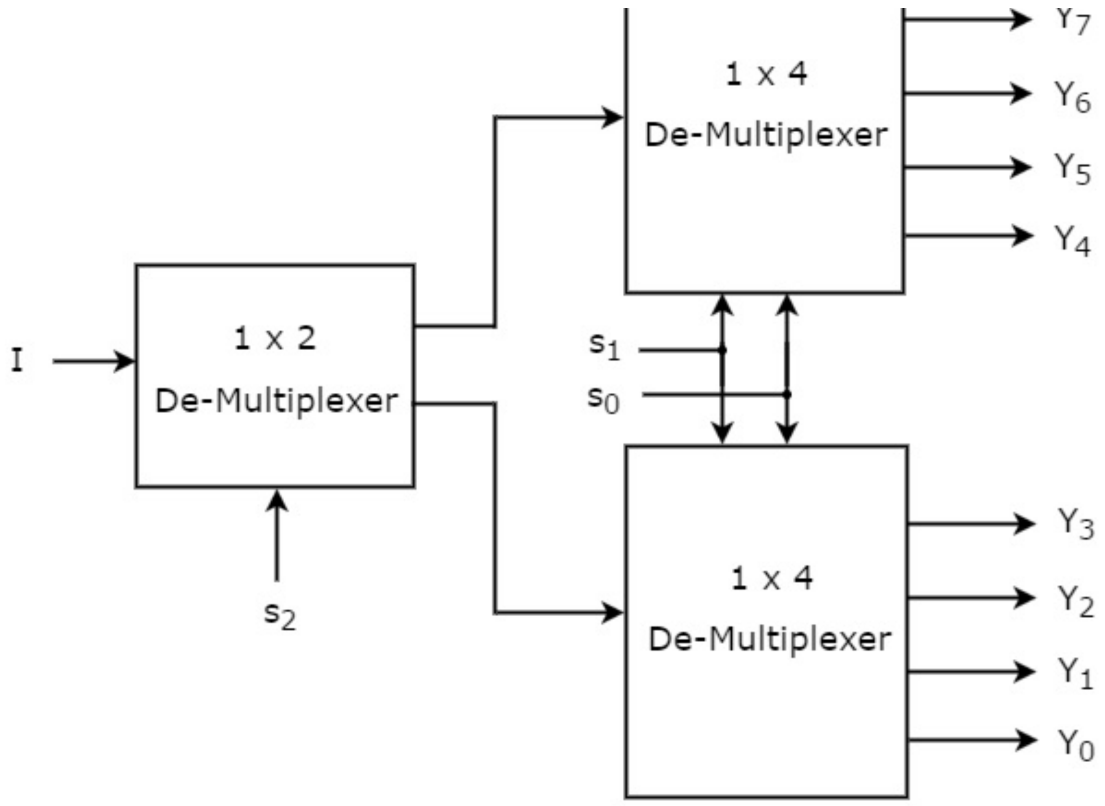
In this section, let us implement 1x8 De-Multiplexer using 1x4 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x4 De-Multiplexer has single input, two selection lines and four outputs. Whereas, 1x8 De-Multiplexer has single input, three selection lines and eight outputs.

So, we require two **1x4 De-Multiplexers** in second stage in order to get the final eight outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x8 De-Multiplexer.

Let the 1x8 De-Multiplexer has one input **I**, three selection lines s_2, s_1 & s_0 and outputs Y_7 to Y_0. The **Truth table** of 1x8 De-Multiplexer is shown below.

Selection Inputs			**Outputs**							
s_2	**s_1**	**s_0**	**Y_7**	**Y_6**	**Y_5**	**Y_4**	**Y_3**	**Y_2**	**Y_1**	**Y_0**
0	0	0	0	0	0	0	0	0	0	**I**
0	0	1	0	0	0	0	0	0	**I**	0
0	1	0	0	0	0	0	0	**I**	0	0
0	1	1	0	0	0	0	**I**	0	0	0
1	0	0	0	0	0	**I**	0	0	0	0
1	0	1	0	0	**I**	0	0	0	0	0
1	1	0	0	**I**	0	0	0	0	0	0
1	1	1	**I**	0	0	0	0	0	0	0

We can implement 1x8 De-Multiplexer using lower order Multiplexers easily by considering the above Truth table. The **block diagram** of 1x8 De-Multiplexer is shown in the following figure.



The common **selection lines, s_1 & s_0** are applied to both 1x4 De-Multiplexers. The outputs of upper 1x4 De-Multiplexer are Y_7 to Y_4 and the outputs of lower 1x4 De-Multiplexer are Y_3 to Y_0.

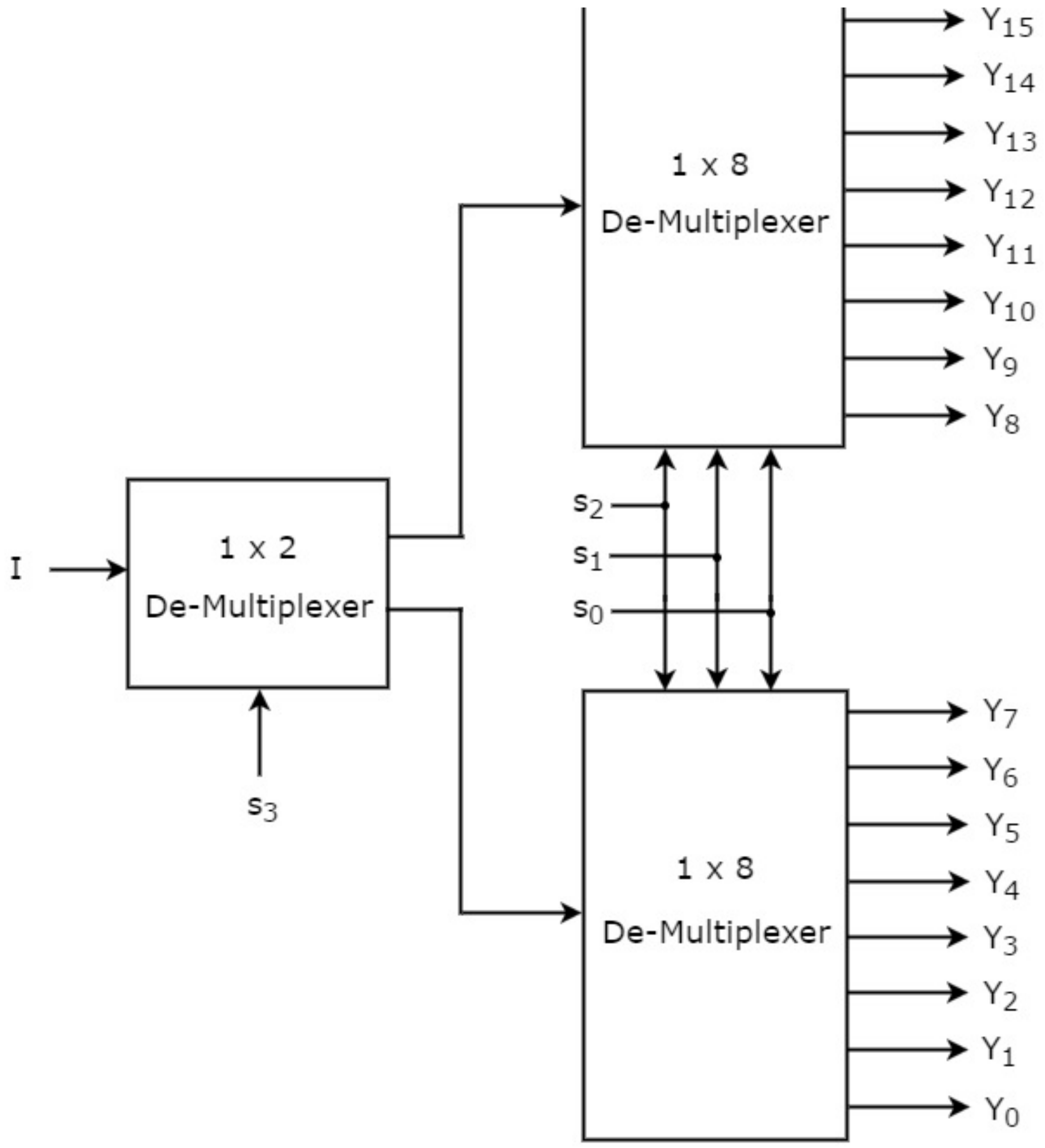
The other **selection line, s_2** is applied to 1x2 De-Multiplexer. If s_2 is zero, then one of the four outputs of lower 1x4 De-Multiplexer will be equal to input, I based on the values of selection lines s_1 & s_0. Similarly, if s_2 is one, then one of the four outputs of upper 1x4 DeMultiplexer will be equal to input, I based on the values of selection lines s_1 & s_0.

1x16 De-Multiplexer

In this section, let us implement 1x16 De-Multiplexer using 1x8 De-Multiplexers and 1x2 De-Multiplexer. We know that 1x8 De-Multiplexer has single input, three selection lines and eight outputs. Whereas, 1x16 De-Multiplexer has single input, four selection lines and sixteen outputs.

So, we require two **1x8 De-Multiplexers** in second stage in order to get the final sixteen outputs. Since, the number of inputs in second stage is two, we require **1x2 DeMultiplexer** in first stage so that the outputs of first stage will be the inputs of second stage. Input of this 1x2 De-Multiplexer will be the overall input of 1x16 De-Multiplexer.

Let the 1x16 De-Multiplexer has one input I, four selection lines s_3, s_2, s_1 & s_0 and outputs Y_{15} to Y_0. The **block diagram** of 1x16 De-Multiplexer using lower order Multiplexers is shown in the following figure.



The common **selection lines s_2, s_1 & s_0** are applied to both 1x8 De-Multiplexers. The outputs of upper 1x8 De-Multiplexer are Y_{15} to Y_8 and the outputs of lower 1x8 DeMultiplexer are Y_7 to Y_0.

The other **selection line, s_3** is applied to 1x2 De-Multiplexer. If s_3 is zero, then one of the eight outputs of lower 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0. Similarly, if s3 is one, then one of the 8 outputs of upper 1x8 De-Multiplexer will be equal to input, I based on the values of selection lines s_2, s_1 & s_0.

Programmable Logic Devices **PLDs** are the integrated circuits. They contain an array of AND gates & another array of OR gates. There are three kinds of PLDs based on the type of arrays, which has programmable feature.

- Programmable Read Only Memory

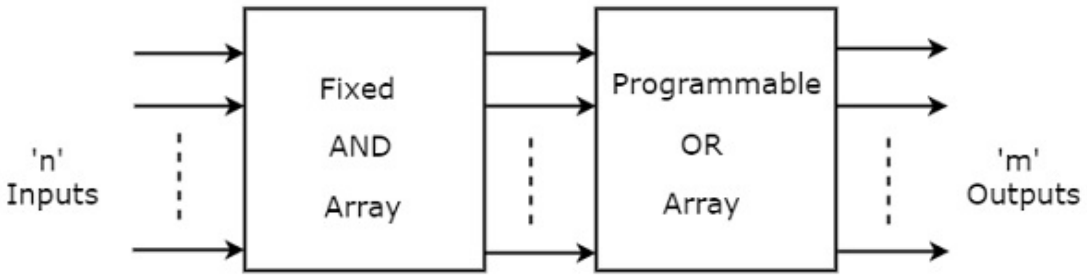
- Programmable Array Logic
- Programmable Logic Array

The process of entering the information into these devices is known as **programming**. Basically, users can program these devices or ICs electrically in order to implement the Boolean functions based on the requirement. Here, the term programming refers to hardware programming but not software programming.

Programmable Read Only Memory PROM

Read Only Memory ROM is a memory device, which stores the binary information permanently. That means, we can’t change that stored information by any means later. If the ROM has programmable feature, then it is called as **Programmable ROM PROM**. The user has the flexibility to program the binary information electrically once by using PROM programmer.

PROM is a programmable logic device that has fixed AND array & Programmable OR array. The **block diagram** of PROM is shown in the following figure.



Here, the inputs of AND gates are not of programmable type. So, we have to generate 2^n product terms by using 2^n AND gates having n inputs each. We can implement these product terms by using nx2^n decoder. So, this decoder generates ‘n’ **min terms**.

Here, the inputs of OR gates are programmable. That means, 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 PROM will be in the form of **sum of min terms**.

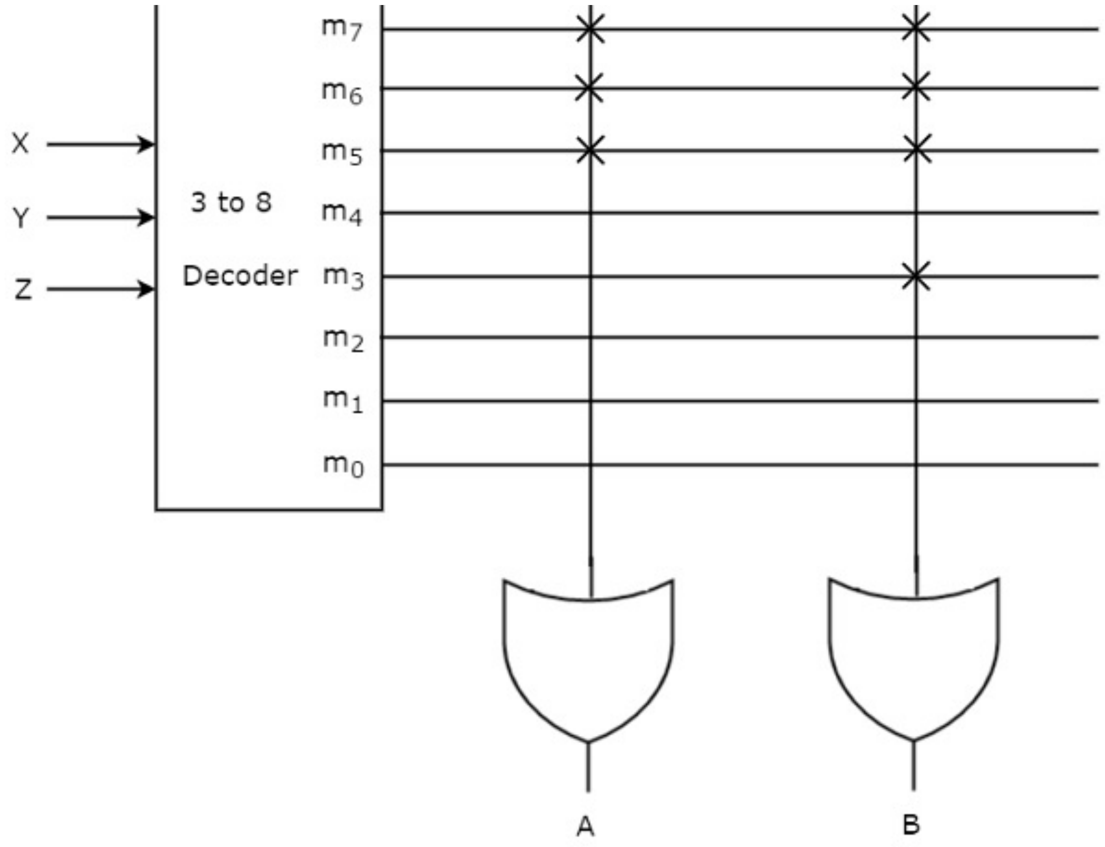
Example

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

A(X,Y,Z)=\sum m\left (5,6,7 \right)

B(X,Y,Z)=\sum m\left (3,5,6,7 \right)

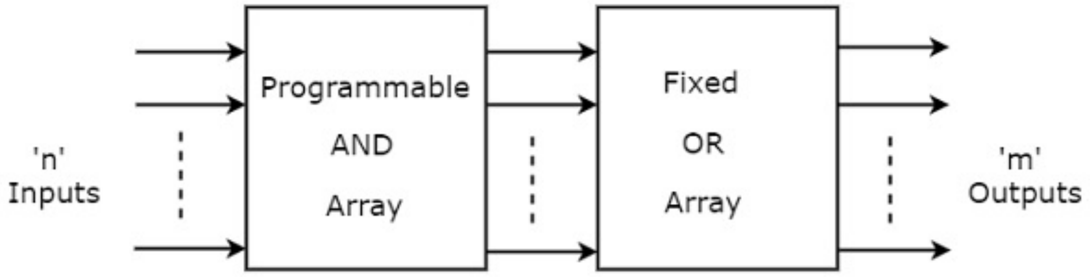
The given two functions are in sum of min terms form and each function is having three variables X, Y & Z. So, we require a 3 to 8 decoder and two programmable OR gates for producing these two functions. The corresponding **PROM** is shown in the following figure.



Here, 3 to 8 decoder generates eight min terms. The two programmable OR gates have the access of all these min terms. But, only the required min terms are programmed in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

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**.

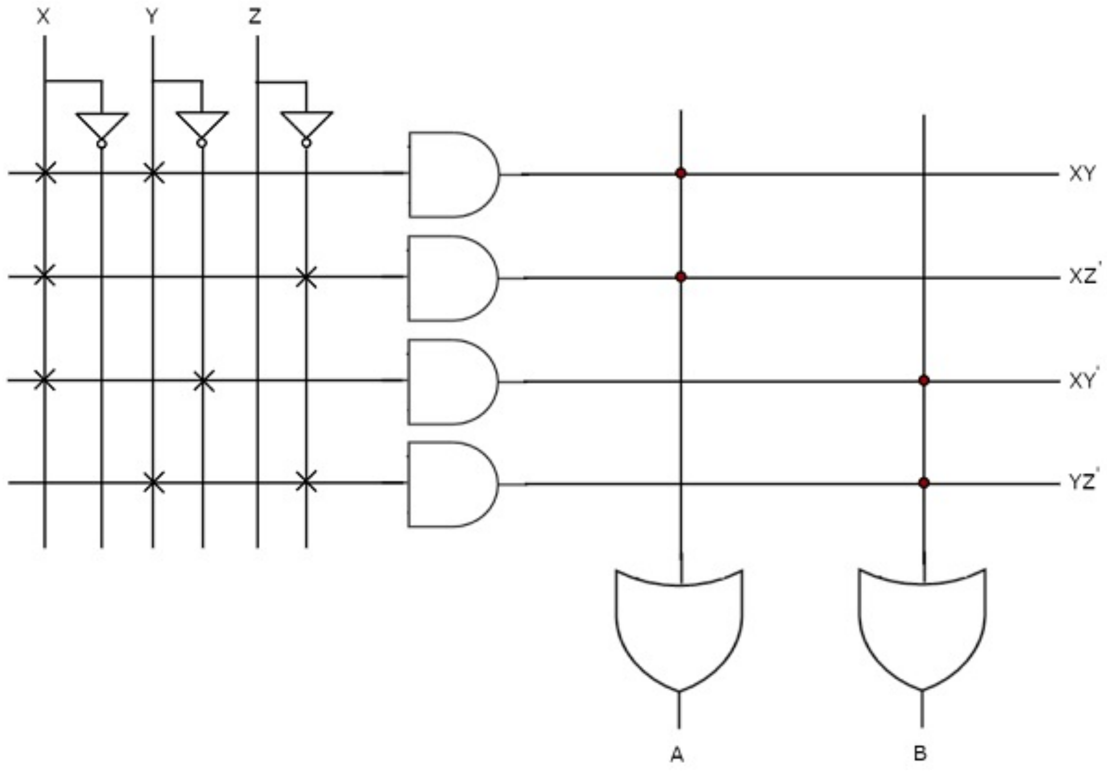
Example

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

A=XY+X{Z}'

A=X{Y}'+Y{Z}'

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.

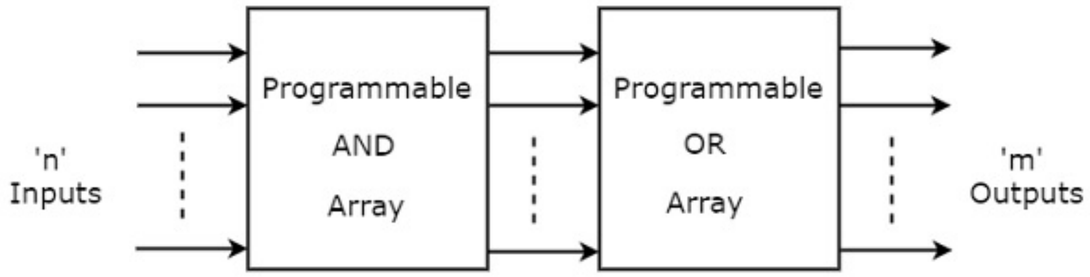


The **programmable AND gates** have the access of both normal and complemented inputs of variables. In the above figure, the inputs X, {X}', Y, {Y}', Z & {Z}', are available at the inputs of each AND gate. So, program only the required literals in order to generate one product term by each AND gate.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

Here, the inputs of OR gates are of fixed type. So, the necessary product terms are connected to inputs of each **OR gate**. So that the OR gates produce the respective Boolean functions. The symbol ‘.’ is used for fixed connections.

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. The **block diagram** of PLA 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 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

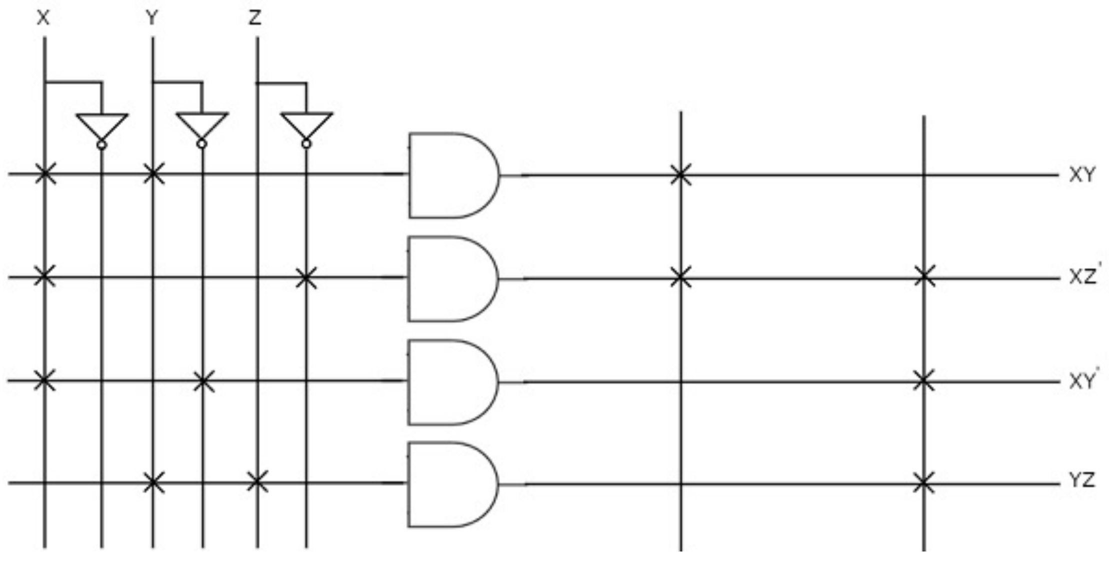
Let us implement the following **Boolean functions** using PLA.

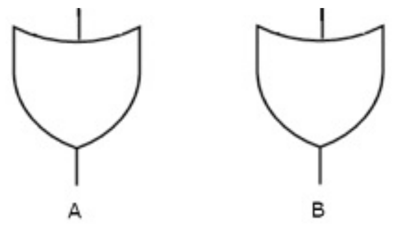
A=XY+X{Z}'

B=X{Y}'+YZ+X{Z}'

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}'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.

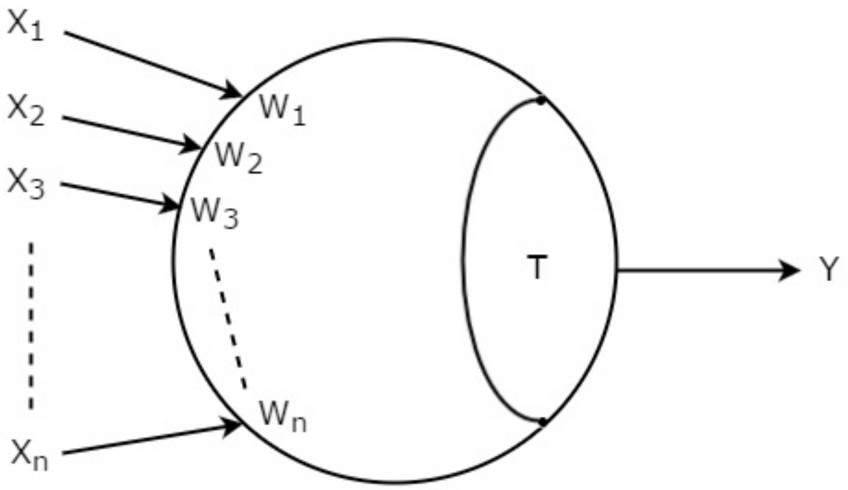
All these product terms are available at the inputs of each **programmable OR gate**. But, only program the required product terms in order to produce the respective Boolean functions by each OR gate. The symbol 'X' is used for programmable connections.

In previous chapters, we have implemented various combinational circuits using logic gates. Except NOT gate, the remaining all logic gates have at least two inputs and single output. Similarly, the **threshold gate** also contains at least one input and only one output.

Additionally, it contains the respective weights to each input and a threshold value. The values of these weights and threshold could be of any finite real number.

Basics of Threshold gate

Let the inputs of threshold gate are X_1, X_2, X_3,..., X_n. The corresponding weights of these inputs are W_1, W_2, W_3,..., W_n. The **symbol** of Threshold gate is shown in the following figure.



Threshold gate is represented with a circle and it is having 'n' inputs, X_1 to X_n and single output, Y. This circle is made into two parts. One part represents the weights corresponding to the inputs and other part represents Threshold value, T.

The sum of products of inputs with corresponding weights is known as **weighted sum**. If this weighted sum is greater than or equal to Threshold value, T then only the output, Y will be equal to one. Otherwise, the output, Y will be equal to zero.

Mathematically, we can write this relationship between inputs and output of Threshold gate as below.

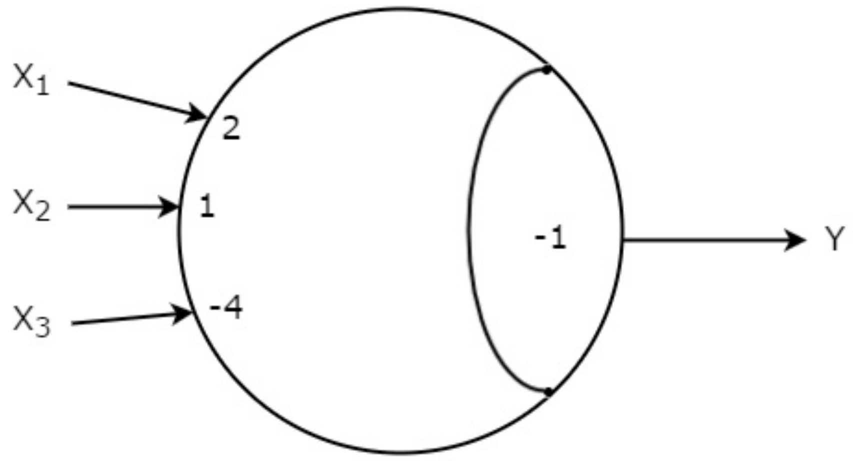
$Y=1, \text{ if } \: \: W_{1}X_{1}+W_{2}X_{2}+W_{3}X_{3}+...W_{n}X_{n}\geq T$

Y = 0, otherwise.

Therefore, we can implement various logic gates and Boolean functions just by changing the values of weights and / or Threshold value, T.

Example

Let us find the **simplified Boolean function** for the following Threshold gate.



This Threshold gate is having three inputs X_1, X_2, X_3 and one output Y.

The weights corresponding to the inputs X_1, X_2 & X_3 are W_1 = 2, W_2 = 1 & W_3 = -4 respectively.

The value of Threshold gate is T = -1.

The **weighted sum** of Threshold gate is

$W=W_{1}X_{1}+W_{2}X_{2}+W_{3}X_{3}$

Substitute the given weights in the above equation.

$\Rightarrow W=2X_{1}+X_{2}-4X_{3}$

Output of Threshold gate, Y will be ‘1’ if W ≥ −1, otherwise it will be ‘0’.

The following **table** shows the relationship between the input and output for all possible combination of inputs.

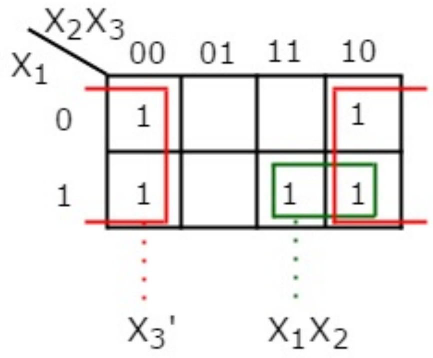
Inputs			**Weighted sum**	**Output**
X_{1}	X_{2}	X_{3}	$W=2X_{1}+X_{2}-4X_{3}$	Y

0	0	0	0	1
0	0	1	-4	0
0	1	0	1	1
0	1	1	-3	0
1	0	0	2	1
1	0	1	-2	0
1	1	0	3	1
1	1	1	-1	1

From the above table, we can write the **Boolean function** for output, Y as

Y= \sum m\left (0,2,4,6,7 \right)

The simplification of this Boolean function using **3 variable K-Map** is shown in the following figure.



Therefore, the **simplified Boolean function** for given Threshold gate is Y={X_{3}'}+X_{1}X_{2}.

Synthesis of Threshold Functions

Threshold gate is also called as **universal gate** because we can implement any Boolean function using Threshold gates. Some-times, it may not possible to implement few logic gates and Boolean functions by using single Threshold gate. In that case, we may require multiple Threshold gates.

Follow these **steps** for implementing a Boolean function using single Threshold gate.

Step 1 – Formulate a **Truth table** for given Boolean function.

Step 2 – In the above Truth table, add include one more column, which gives the relation between **weighted sums** and **Threshold value**.

Step 3 – Write the relation between weighted sums and threshold for each combination of inputs as mentioned below.

- If the output of Boolean function is 1, then the weighted sum will be greater than or equal to Threshold value for those combination of inputs.

- If the output of Boolean function is 0, then the weighted sum will be less than Threshold value for those combination of inputs.

Step 4 – Choose the values of weights & Threshold in such a way that they should satisfy all the relations present in last column of the above table.

step 5 – Draw the **symbol** of Threshold gate with those weights and Threshold value.

Example

Let us implement the following **Boolean function** using single Threshold gate.

$Y\left (X_{1},X_{2},X_{3} \right)=\sum m\left (0,2,4,6,7 \right)$

The given Boolean function is a three variable function, which is represented in sum of min terms form. The **Truth table** of this function is shown below.

Inputs			**Output**
X_1	**X_2**	**X_3**	**Y**
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	1

Now, let us add include one more column to the above Truth table. This last column contains the relations between **weighted sums W and Threshold** value T for each combination of inputs.

Inputs			**Output**	**Relations between W & T**
X_1	**X_2**	**X_3**	**Y**	
0	0	0	1	$0 \geq T$
0	0	1	0	$W_3 < T$
0	1	0	1	$W_2 \geq T$
0	1	1	0	$W_2 + W_3 < T$
1	0	0	1	$W_1 \geq T$

1	0	1	0	$W_1 + W_3 < T$
1	1	0	1	$W_1 + W_2 \geq T$
1	1	1	1	$W_1 + W_2 + W_3 \geq T$

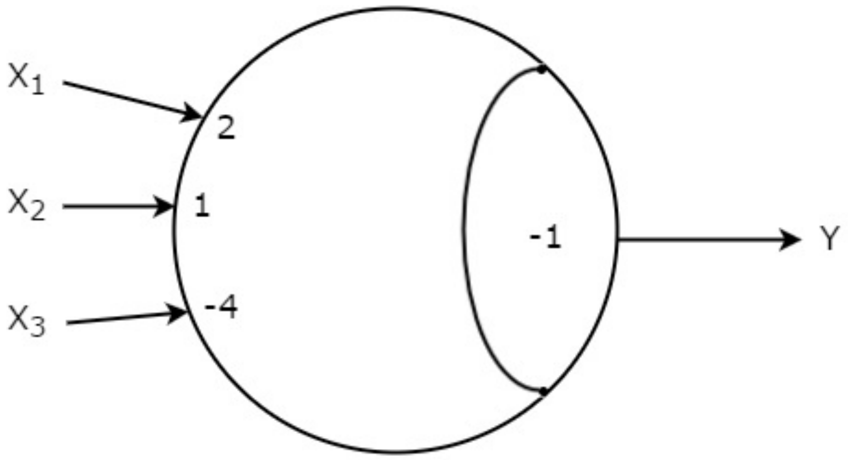
Following are the conclusions from the above table.

- The value of Threshold should be either zero or negative based on first relation.
- The value of W_3 should be negative based on first and second relations.
- The values of W_1 and W_2 should be greater than or equal Threshold value based on fifth and third relations.
- W_2 should be greater than W_3 based on fourth relation.

We can choose the following values for weights and Threshold based on the above conclusions.

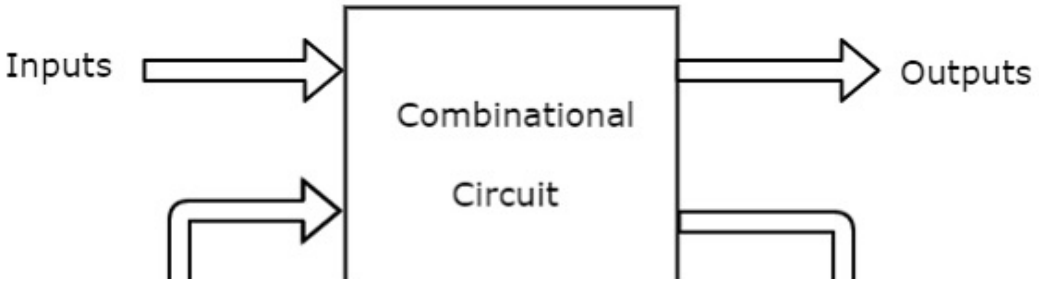
W_1=2, W_2=1, W_3= -4 & T = -1

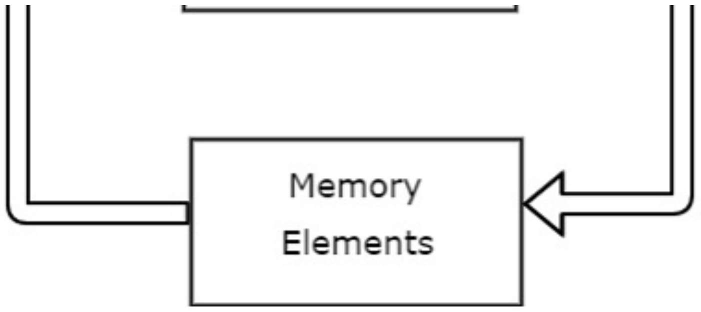
The **symbol** of Threshold gate with the above values is shown below.



Therefore, this Threshold gate implements the given **Boolean function**, Y\left (X_{1}, X_{2},X_{3} \right)=\sum m\left (0,2,4,6,7 \right).

We discussed various combinational circuits in earlier chapters. All these circuits have a set of outputs, which depends only on the combination of present inputs. The following figure shows the **block diagram** of sequential circuit.





This sequential circuit contains a set of inputs and outputs. The outputs of sequential circuit depends not only on the combination of present inputs but also on the previous outputs. Previous output is nothing but the **present state**. Therefore, sequential circuits contain combinational circuits along with memory storage elements. Some sequential circuits may not contain combinational circuits, but only memory elements.

Following table shows the **differences** between combinational circuits and sequential circuits.

Combinational Circuits	Sequential Circuits
Outputs depend only on present inputs.	Outputs depend on both present inputs and present state.
Feedback path is not present.	Feedback path is present.
Memory elements are not required.	Memory elements are required.
Clock signal is not required.	Clock signal is required.
Easy to design.	Difficult to design.

Types of Sequential Circuits

Following are the two types of sequential circuits –

- Asynchronous sequential circuits
- Synchronous sequential circuits

Asynchronous sequential circuits

If some or all the outputs of a sequential circuit do not change affect with respect to active transition of clock signal, then that sequential circuit is called as **Asynchronous sequential circuit**. That means, all the outputs of asynchronous sequential circuits do not change affect at the same time. Therefore, most of the outputs of asynchronous sequential circuits are **not in synchronous** with either only positive edges or only negative edges of clock signal.

Synchronous sequential circuits

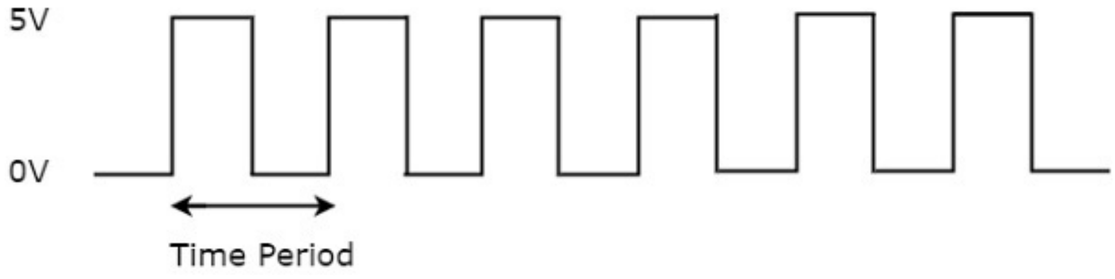
If all the outputs of a sequential circuit change affect with respect to active transition of clock signal, then that sequential circuit is called as **Synchronous sequential circuit**. That means, all the outputs of synchronous sequential circuits change affect at the same time. Therefore, the outputs of synchronous sequential circuits are in synchronous with either only positive edges or only negative edges of clock signal.

Clock Signal and Triggering

In this section, let us discuss about the clock signal and types of triggering one by one.

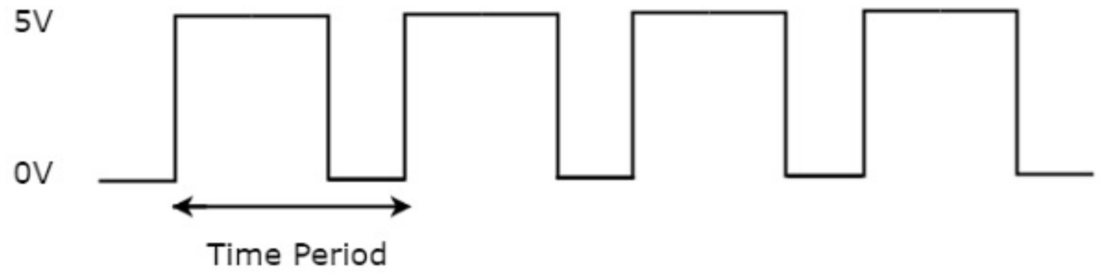
Clock signal

Clock signal is a periodic signal and its ON time and OFF time need not be the same. We can represent the clock signal as a **square wave**, when both its ON time and OFF time are same. This clock signal is shown in the following figure.



n the above figure, square wave is considered as clock signal. This signal stays at logic High 5V for some time and stays at logic Low 0V for equal amount of time. This pattern repeats with some time period. In this case, the **time period** will be equal to either twice of ON time or twice of OFF time.

We can represent the clock signal as **train of pulses**, when ON time and OFF time are not same. This clock signal is shown in the following figure.



In the above figure, train of pulses is considered as clock signal. This signal stays at logic High 5V for some time and stays at logic Low 0V for some other time. This pattern repeats with some time period. In this case, the **time period** will be equal to sum of ON time and OFF time.

The reciprocal of the time period of clock signal is known as the **frequency** of the clock signal. All sequential circuits are operated with clock signal. So, the frequency at which the sequential circuits can be operated accordingly the clock signal frequency has to be chosen.

Types of Triggering

Following are the two possible types of triggering that are used in sequential circuits.

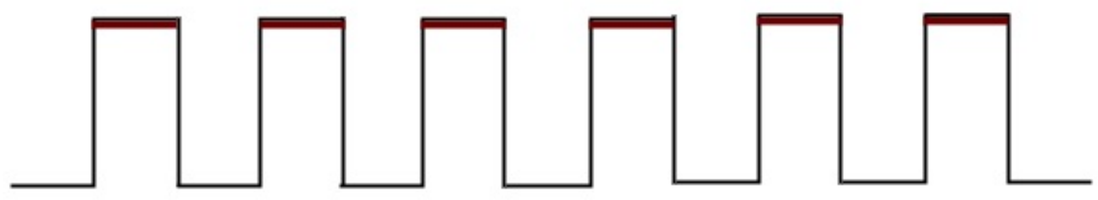
- Level triggering
- Edge triggering

Level triggering

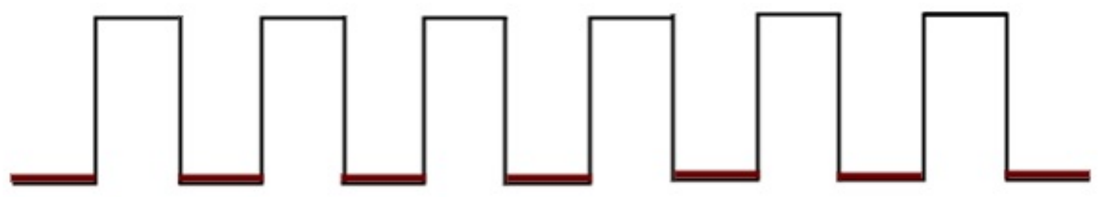
There are two levels, namely logic High and logic Low in clock signal. Following are the two **types of level triggering**.

- Positive level triggering
- Negative level triggering

If the sequential circuit is operated with the clock signal when it is in **Logic High**, then that type of triggering is known as **Positive level triggering**. It is highlighted in below figure.



If the sequential circuit is operated with the clock signal when it is in **Logic Low**, then that type of triggering is known as **Negative level triggering**. It is highlighted in the following figure.



Edge triggering

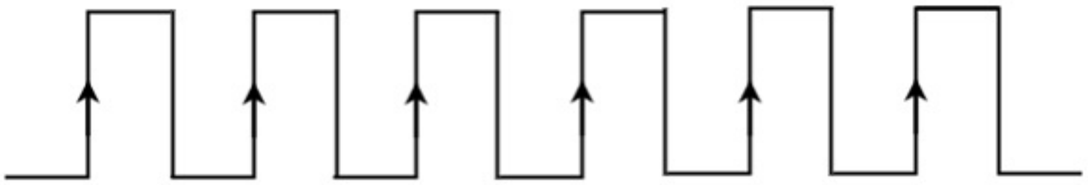
There are two types of transitions that occur in clock signal. That means, the clock signal transitions either from Logic Low to Logic High or Logic High to Logic Low.

Following are the two **types of edge triggering** based on the transitions of clock signal.

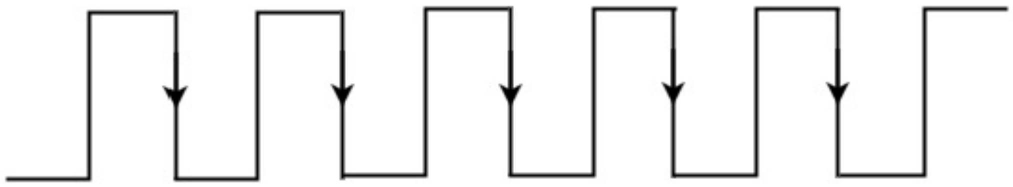
- Positive edge triggering

- Negative edge triggering

  If the sequential circuit is operated with the clock signal that is transitioning from Logic Low to Logic High, then that type of triggering is known as **Positive edge triggering**. It is also called as rising edge triggering. It is shown in the following figure.



  If the sequential circuit is operated with the clock signal that is transitioning from Logic High to Logic Low, then that type of triggering is known as **Negative edge triggering**. It is also called as falling edge triggering. It is shown in the following figure.



  In coming chapters, we will discuss about various sequential circuits based on the type of triggering that can be used in it.

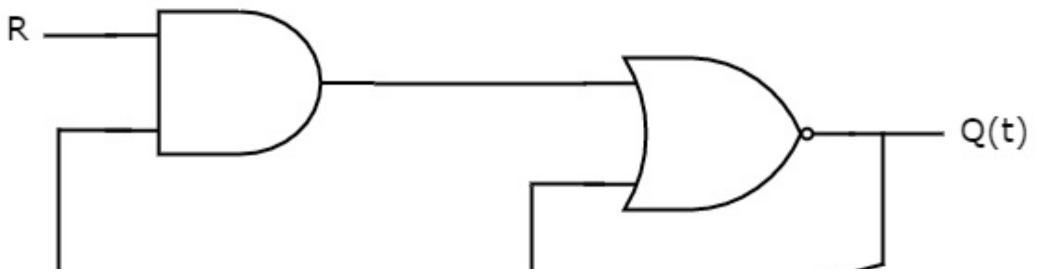
  There are two types of memory elements based on the type of triggering that is suitable to operate it.

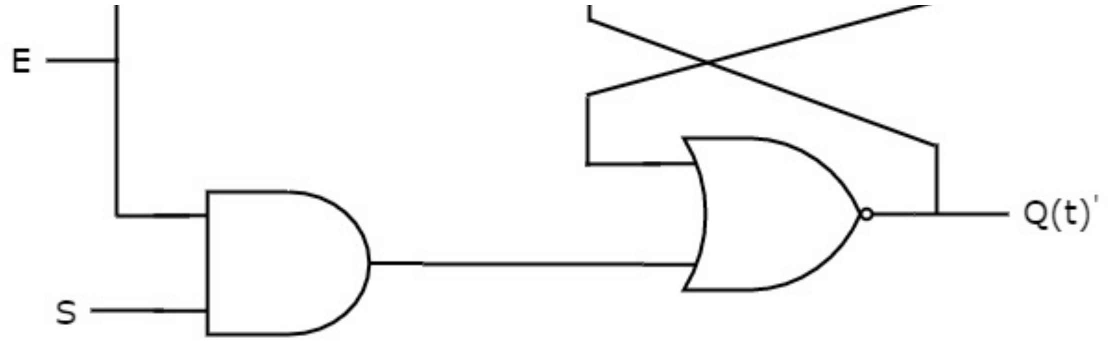
- Latches
- Flip-flops

  Latches operate with enable signal, which is **level sensitive**. Whereas, flip-flops are edge sensitive. We will discuss about flip-flops in next chapter. Now, let us discuss about SR Latch & D Latch one by one.

SR Latch

SR Latch is also called as **Set Reset Latch**. This latch affects the outputs as long as the enable, E is maintained at ‘1’. The **circuit diagram** of SR Latch is shown in the following figure.





This circuit has two inputs S & R and two outputs Qt & Qt’. The **upper NOR gate** has two inputs R & complement of present state, Qt’ and produces next state, Qt+1 when enable, E is ‘1’.

Similarly, the **lower NOR gate** has two inputs S & present state, Qt and produces complement of next state, Qt+1’ when enable, E is ‘1’.

We know that a **2-input NOR gate** produces an output, which is the complement of another input when one of the input is ‘0’. Similarly, it produces ‘0’ output, when one of the input is ‘1’.

- If S = 1, then next state Qt + 1 will be equal to ‘1’ irrespective of present state, Qt values.
- If R = 1, then next state Qt + 1 will be equal to ‘0’ irrespective of present state, Qt values.

At any time, only of those two inputs should be ‘1’. If both inputs are ‘1’, then the next state Qt + 1 value is undefined.

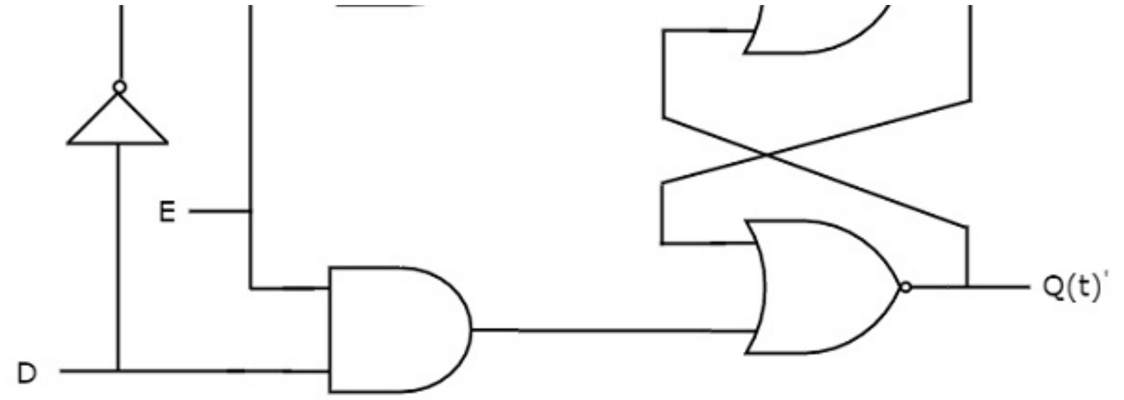
The following table shows the **state table** of SR latch.

S	R	Qt + 1
0	0	Qt
0	1	0
1	0	1
1	1	-

Therefore, SR Latch performs three types of functions such as Hold, Set & Reset based on the input conditions.

D Latch

There is one drawback of SR Latch. That is the next state value can’t be predicted when both the inputs S & R are one. So, we can overcome this difficulty by D Latch. It is also called as Data Latch. The **circuit diagram** of D Latch is shown in the following figure.



This circuit has single input D and two outputs Qt & Qt'. D Latch is obtained from SR Latch by placing an inverter between S amp;& R inputs and connect D input to S. That means we eliminated the combinations of S & R are of same value.

- If D = 0 → S = 0 & R = 1, then next state Qt + 1 will be equal to '0' irrespective of present state, Qt values. This is corresponding to the second row of SR Latch state table.
- If D = 1 → S = 1 & R = 0, then next state Qt + 1 will be equal to '1' irrespective of present state, Qt values. This is corresponding to the third row of SR Latch state table.

The following table shows the **state table** of D latch.

D	Qt + 1
0	0
1	1

Therefore, D Latch Hold the information that is available on data input, D. That means the output of D Latch is sensitive to the changes in the input, D as long as the enable is High.

In this chapter, we implemented various Latches by providing the cross coupling between NOR gates. Similarly, you can implement these Latches using NAND gates.

In previous chapter, we discussed about Latches. Those are the basic building blocks of flip-flops. We can implement flip-flops in two methods.

In first method, **cascade two latches** in such a way that the first latch is enabled for every positive clock pulse and second latch is enabled for every negative clock pulse. So that the combination of these two latches become a flip-flop.

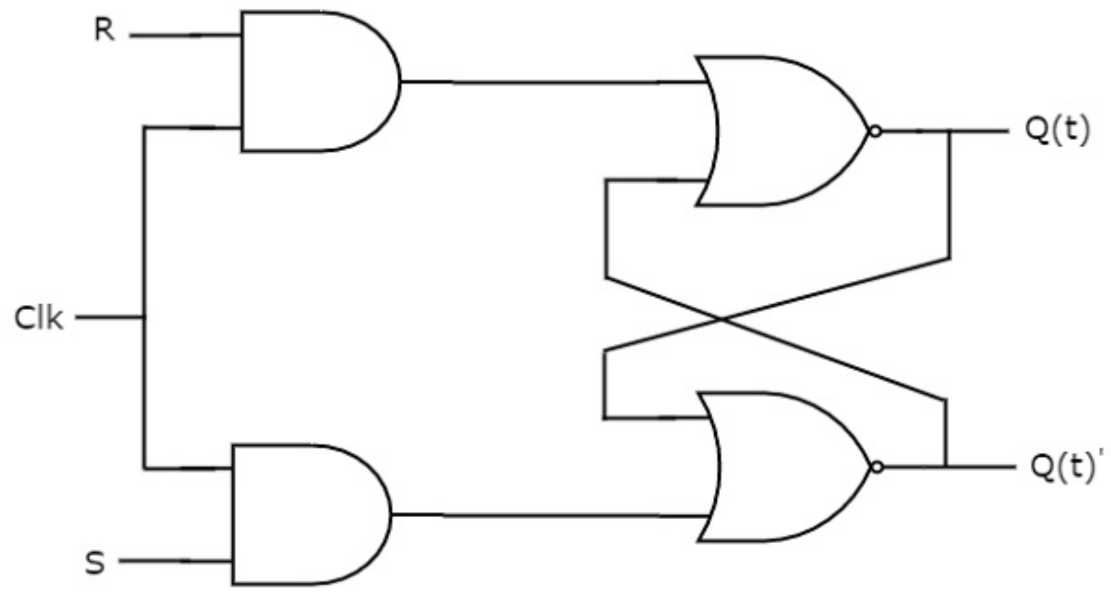
In second method, we can directly implement the flip-flop, which is edge sensitive. In this chapter, let us discuss the following **flip-flops** using second method.

- SR Flip-Flop
- D Flip-Flop
- JK Flip-Flop

- T Flip-Flop

SR Flip-Flop

SR flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, SR latch operates with enable signal. The **circuit diagram** of SR flip-flop is shown in the following figure.



This circuit has two inputs S & R and two outputs Qt & Qt'. The operation of SR flipflop is similar to SR Latch. But, this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table shows the **state table** of SR flip-flop.

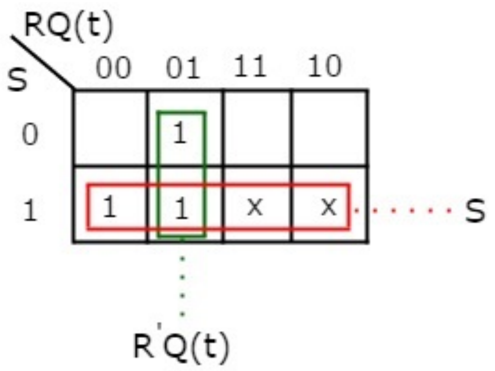
S	R	Qt + 1
0	0	Qt
0	1	0
1	0	1
1	1	-

Here, Qt & Qt + 1 are present state & next state respectively. So, SR flip-flop can be used for one of these three functions such as Hold, Reset & Set based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of SR flip-flop.

Present Inputs		Present State	Next State
S	**R**	**Qt**	**Qt + 1**
0	0	0	0
0	0	1	1

0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

By using three variable K-Map, we can get the simplified expression for next state, Qt + 1. The **three variable K-Map** for next state, Qt + 1 is shown in the following figure.

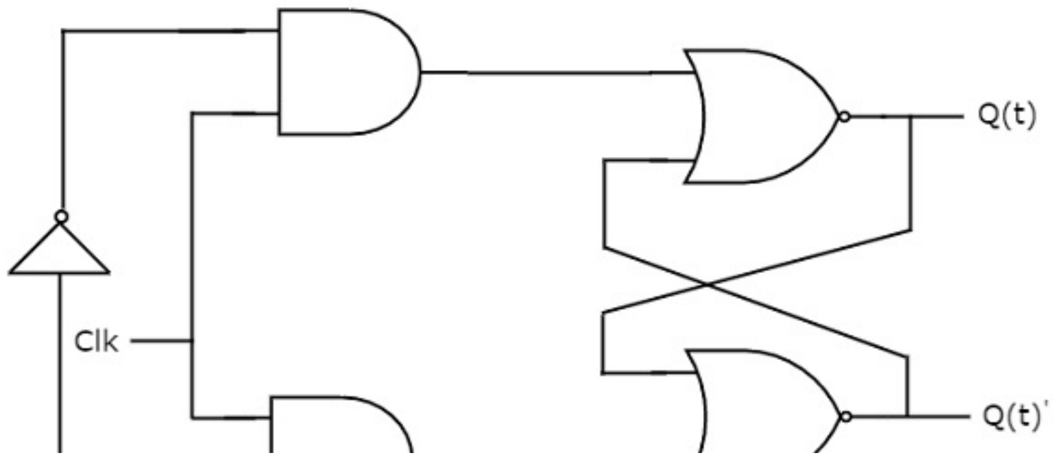


The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the **simplified expression** for next state Qt + 1 is

Q\left (t+1 \right)=S+{R}'Q\left (t \right)

D Flip-Flop

D flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, D latch operates with enable signal. That means, the output of D flip-flop is insensitive to the changes in the input, D except for active transition of the clock signal. The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit has single input D and two outputs Qt & Qt’. The operation of D flip-flop is similar to D Latch. But, this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table shows the **state table** of D flip-flop.

D	Qt + 1t + 1
0	0
1	1

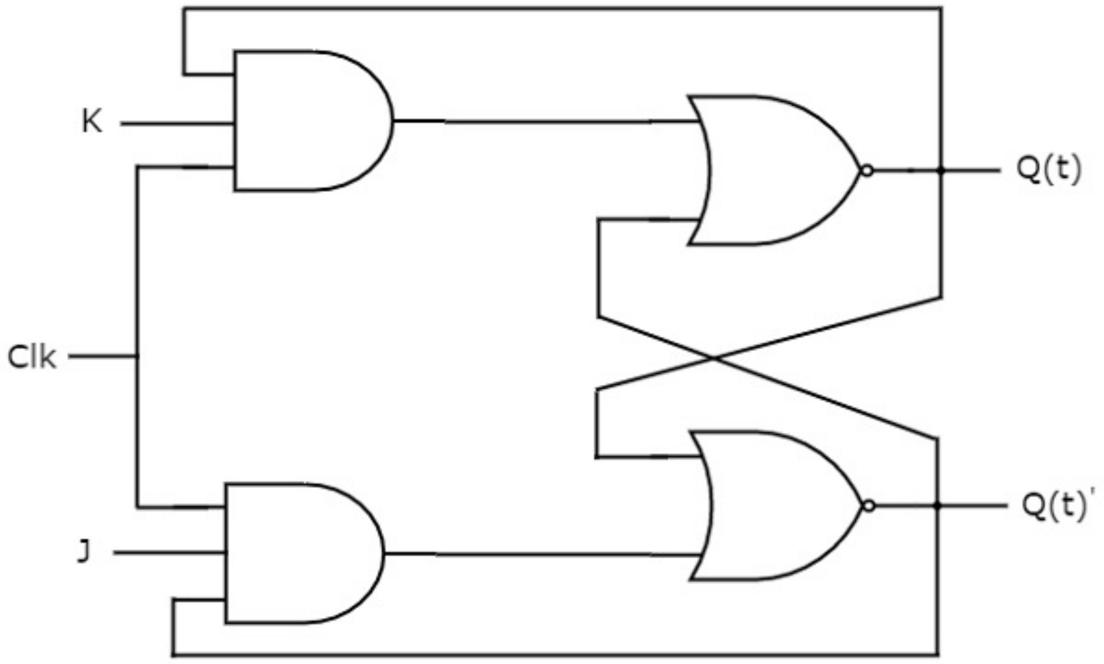
Therefore, D flip-flop always Hold the information, which is available on data input, D of earlier positive transition of clock signal. From the above state table, we can directly write the next state equation as

Qt + 1 = D

Next state of D flip-flop is always equal to data input, D for every positive transition of the clock signal. Hence, D flip-flops can be used in registers, **shift registers** and some of the counters.

JK Flip-Flop

JK flip-flop is the modified version of SR flip-flop. It operates with only positive clock transitions or negative clock transitions. The **circuit diagram** of JK flip-flop is shown in the following figure.



This circuit has two inputs J & K and two outputs Qt & Qt’. The operation of JK flip-flop is similar to SR flip-flop. Here, we considered the inputs of SR flip-flop as **S = J Qt'** and **R = KQt** in order to utilize the modified SR flip-flop for 4 combinations of inputs.

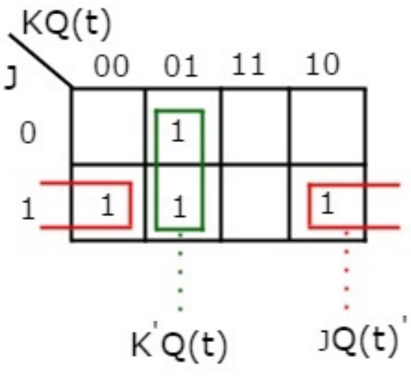
The following table shows the **state table** of JK flip-flop.

J	K	Qt + 1
0	0	Qt
0	1	0
1	0	1
1	1	Qt'

Here, Qt & Qt + 1 are present state & next state respectively. So, JK flip-flop can be used for one of these four functions such as Hold, Reset, Set & Complement of present state based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of JK flip-flop.

Present Inputs		Present State	Next State
J	**K**	**Qt**	**Qt+1**
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

By using three variable K-Map, we can get the simplified expression for next state, Qt + 1. **Three variable K-Map** for next state, Qt + 1 is shown in the following figure.

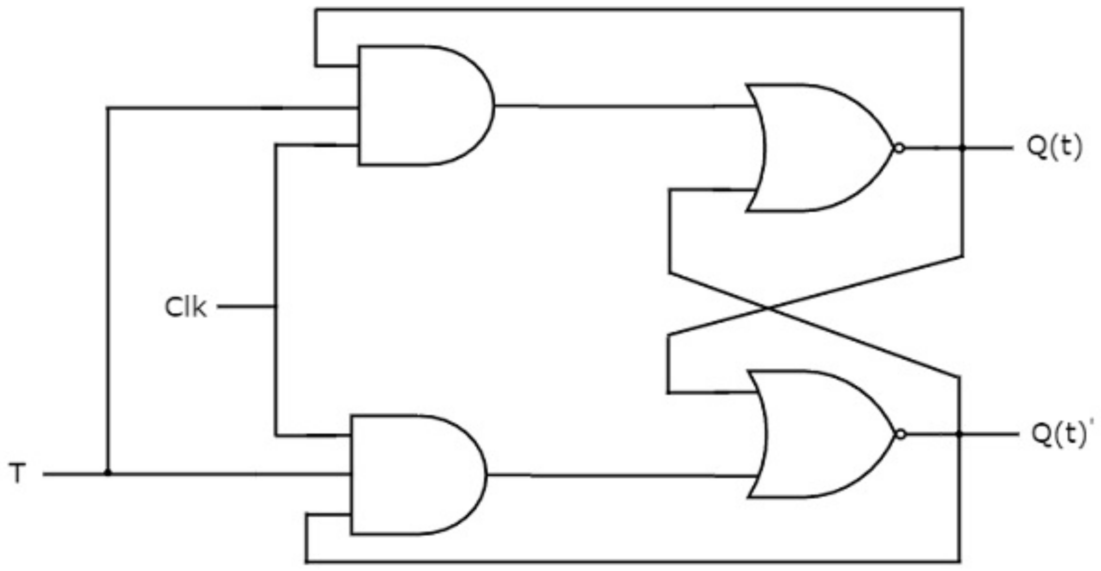


The maximum possible groupings of adjacent ones are already shown in the figure. Therefore, the **simplified expression** for next state Qt+1 is

Q\left (t+1 \right)=J{Q\left (t \right)}'+{K}'Q\left (t \right)

T Flip-Flop

T flip-flop is the simplified version of JK flip-flop. It is obtained by connecting the same input 'T' to both inputs of JK flip-flop. It operates with only positive clock transitions or negative clock transitions. The **circuit diagram** of T flip-flop is shown in the following figure.



This circuit has single input T and two outputs Qt & Qt'. The operation of T flip-flop is same as that of JK flip-flop. Here, we considered the inputs of JK flip-flop as **J = T** and **K = T** in order to utilize the modified JK flip-flop for 2 combinations of inputs. So, we eliminated the other two combinations of J & K, for which those two values are complement to each other in T flip-flop.

The following table shows the **state table** of T flip-flop.

D	Qt + 1
0	Qt
1	Qt'

Here, Qt & Qt + 1 are present state & next state respectively. So, T flip-flop can be used for one of these two functions such as Hold, & Complement of present state based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of T flip-flop.

Inputs	Present State	Next State
T	**Qt**	**Qt + 1**

0	0	0
0	1	1
1	0	1
1	1	0

From the above characteristic table, we can directly write the **next state equation** as

$Q\left (t+1 \right)={T}'Q\left (t \right)+TQ{\left (t \right)}'$

$\Rightarrow Q\left (t+1 \right)=T\oplus Q\left (t \right)$

The output of T flip-flop always toggles for every positive transition of the clock signal, when input T remains at logic High 1. Hence, T flip-flop can be used in **counters**.

In this chapter, we implemented various flip-flops by providing the cross coupling between NOR gates. Similarly, you can implement these flip-flops by using NAND gates.

In previous chapter, we discussed the four flip-flops, namely SR flip-flop, D flip-flop, JK flip-flop & T flip-flop. We can convert one flip-flop into the remaining three flip-flops by including some additional logic. So, there will be total of twelve **flip-flop conversions**.

Follow these **steps** for converting one flip-flop to the other.

- Consider the **characteristic table** of desired flip-flop.
- Fill the excitation values inputs of given flip-flop for each combination of present state and next state. The **excitation table** for all flip-flops is shown below.

Present State	Next State	SR flip-flop inputs		D flip-flop input	JK flip-flop inputs		T flip-flop input
Qt	**Qt+1**	**S**	**R**	**D**	**J**	**K**	**T**
0	0	0	x	0	0	x	0
0	1	1	0	1	1	x	1
1	0	0	1	0	x	1	1
1	1	x	0	1	x	0	0

- Get the **simplified expressions** for each excitation input. If necessary, use Kmaps for simplifying.
- Draw the **circuit diagram** of desired flip-flop according to the simplified expressions using given flip-flop and necessary logic gates.

Now, let us convert few flip-flops into other. Follow the same process for remaining flipflop conversions.

SR Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of SR flip-flop to other flip-flops.

- SR flip-flop to D flip-flop
- SR flip-flop to JK flip-flop
- SR flip-flop to T flip-flop

SR flip-flop to D flip-flop conversion

Here, the given flip-flop is SR flip-flop and the desired flip-flop is D flip-flop. Therefore, consider the following **characteristic table** of D flip-flop.

D flip-flop input	Present State	Next State
D	**Qt**	**Qt + 1**
0	0	0
0	1	0
1	0	1
1	1	1

We know that SR flip-flop has two inputs S & R. So, write down the excitation values of SR flip-flop for each combination of present state and next state values. The following table shows the characteristic table of D flip-flop along with the **excitation inputs** of SR flip-flop.

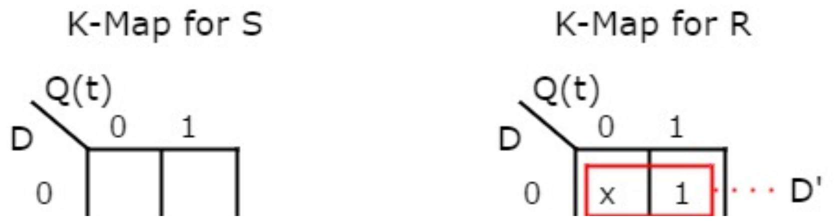
D flip-flop input	Present State	Next State	SR flip-flop inputs	
D	**Qt**	**Qt + 1**	**S**	**R**
0	0	0	0	x
0	1	0	0	1
1	0	1	1	0
1	1	1	x	0

From the above table, we can write the **Boolean functions** for each input as below.

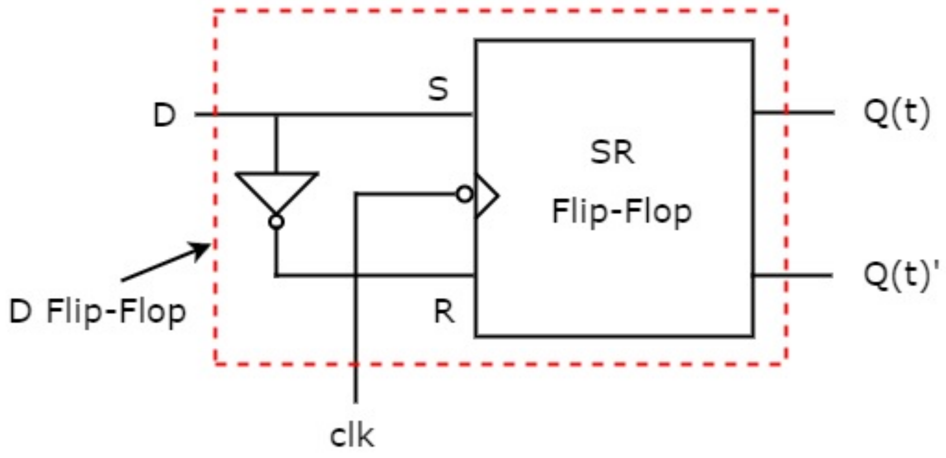
S=m_{2}+d_{3}

R=m_{1}+d_{0}

We can use 2 variable K-Maps for getting simplified expressions for these inputs. The **k-Maps** for S & R are shown below.



So, we got S = D & R = D' after simplifying. The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit consists of SR flip-flop and an inverter. This inverter produces an output, which is complement of input, D. So, the overall circuit has single input, D and two outputs Qt & Qt'. Hence, it is a **D flip-flop**. Similarly, you can do other two conversions.

D Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of D flip-flop to other flip-flops.

- D flip-flop to T flip-flop
- D flip-flop to SR flip-flop
- D flip-flop to JK flip-flop

D flip-flop to T flip-flop conversion

Here, the given flip-flop is D flip-flop and the desired flip-flop is T flip-flop. Therefore, consider the following **characteristic table** of T flip-flop.

T flip-flop input	Present State	Next State
T	**Qt**	**Qt + 1**
0	0	0
0	1	1
1	0	1
1	1	0

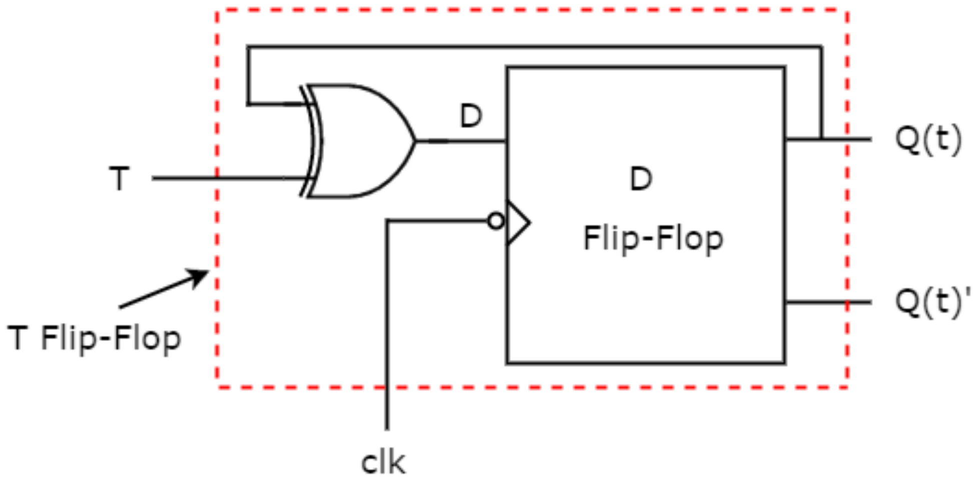
We know that D flip-flop has single input D. So, write down the excitation values of D flip-flop for each combination of present state and next state values. The following table shows the characteristic table of T flip-flop along with the **excitation input** of D flip-flop.

T flip-flop input	Present State	Next State	D flip-flop input
T	**Qt**	**Qt + 1**	**D**
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

From the above table, we can directly write the **Boolean function** of D as below.

D=T\oplus Q\left (t \right)

So, we require a two input Exclusive-OR gate along with D flip-flop. The **circuit diagram** of T flip-flop is shown in the following figure.



This circuit consists of D flip-flop and an Exclusive-OR gate. This Exclusive-OR gate produces an output, which is Ex-OR of T and Qt. So, the overall circuit has single input, T and two outputs Qt & Qt'. Hence, it is a **T flip-flop**. Similarly, you can do other two conversions.

JK Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of JK flip-flop to other flip-flops.

- JK flip-flop to T flip-flop
- JK flip-flop to D flip-flop
- JK flip-flop to SR flip-flop

JK flip-flop to T flip-flop conversion

Here, the given flip-flop is JK flip-flop and the desired flip-flop is T flip-flop. Therefore, consider the following **characteristic table** of T flip-flop.

T flip-flop input	**Present State**	**Next State**
T	**Qt**	**Qt + 1**
0	0	0
0	1	1
1	0	1
1	1	0

We know that JK flip-flop has two inputs J & K. So, write down the excitation values of JK flip-flop for each combination of present state and next state values. The following table shows the characteristic table of T flip-flop along with the **excitation inputs** of JK flipflop.

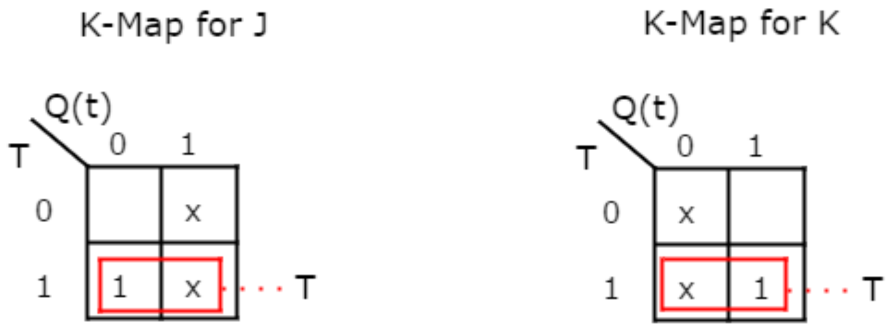
T flip-flop input	**Present State**	**Next State**	**JK flip-flop inputs**	
T	**Qt**	**Qt + 1**	**J**	**K**
0	0	0	0	x
0	1	1	x	0
1	0	1	1	x
1	1	0	x	1

From the above table, we can write the **Boolean functions** for each input as below.

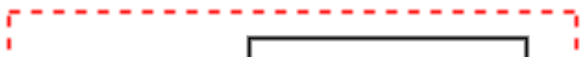
J=m_{2}+d_{1}+d_{3}

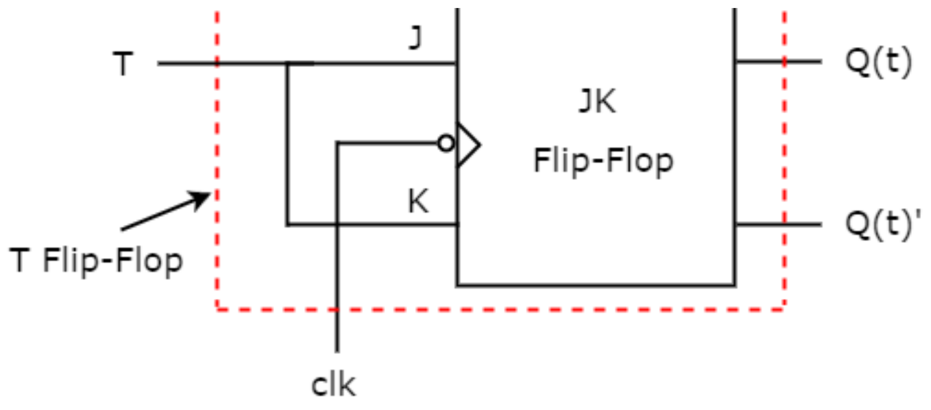
K=m_{3}+d_{0}+d_{2}

We can use 2 variable K-Maps for getting simplified expressions for these two inputs. The **k-Maps** for J & K are shown below.



So, we got, J = T & K = T after simplifying. The **circuit diagram** of T flip-flop is shown in the following figure.





This circuit consists of JK flip-flop only. It doesn’t require any other gates. Just connect the same input T to both J & K. So, the overall circuit has single input, T and two outputs Qt & Qt’. Hence, it is a **T flip-flop**. Similarly, you can do other two conversions.

T Flip-Flop to other Flip-Flop Conversions

Following are the three possible conversions of T flip-flop to other flip-flops.

- T flip-flop to D flip-flop
- T flip-flop to SR flip-flop
- T flip-flop to JK flip-flop

T flip-flop to D flip-flop conversion

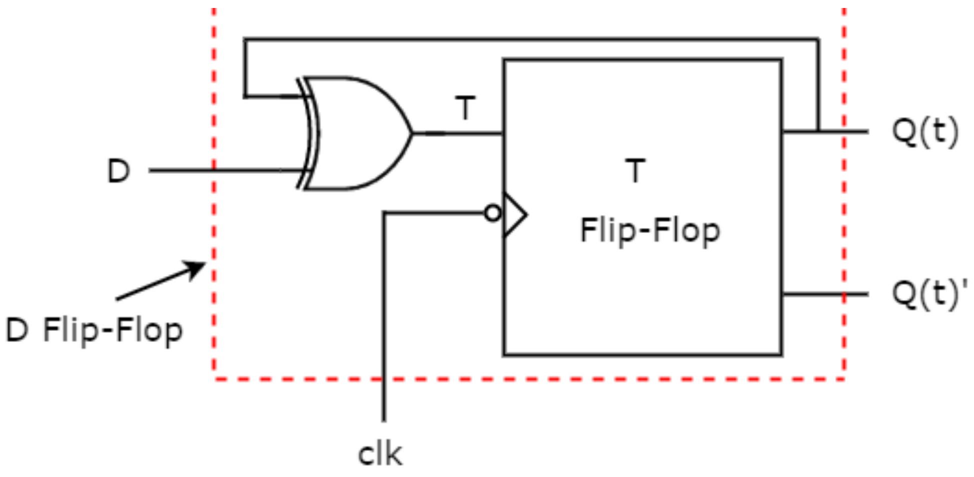
Here, the given flip-flop is T flip-flop and the desired flip-flop is D flip-flop. Therefore, consider the characteristic table of D flip-flop and write down the excitation values of T flip-flop for each combination of present state and next state values. The following table shows the **characteristic table** of D flip-flop along with the **excitation input** of T flip-flop.

D flip-flop input	**Present State**	**Next State**	**T flip-flop input**
D	**Qt**	**Qt + 1**	**T**
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

From the above table, we can directly write the Boolean function of T as below.

T=D\oplus Q\left (t \right)

So, we require a two input Exclusive-OR gate along with T flip-flop. The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit consists of T flip-flop and an Exclusive-OR gate. This Exclusive-OR gate produces an output, which is Ex-OR of D and Qt. So, the overall circuit has single input, D and two outputs Qt & Qt'. Hence, it is a **D flip-flop**. Similarly, you can do other two conversions.

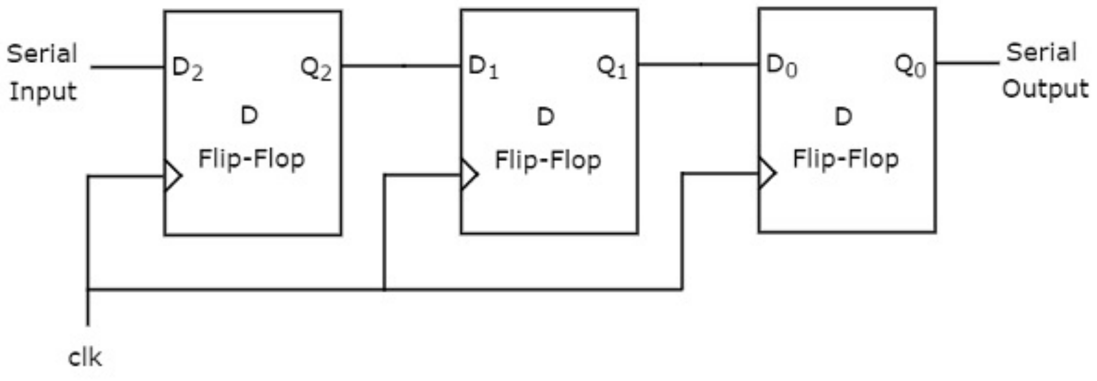
We know that one flip-flop can store one-bit of information. In order to store multiple bits of information, we require multiple flip-flops. The group of flip-flops, which are used to hold store the binary data is known as **register**.

If the register is capable of shifting bits either towards right hand side or towards left hand side is known as **shift register**. An 'N' bit shift register contains 'N' flip-flops. Following are the four types of shift registers based on applying inputs and accessing of outputs.

- Serial In – Serial Out shift register
- Serial In – Parallel Out shift register
- Parallel In – Serial Out shift register
- Parallel In – Parallel Out shift register

Serial In – Serial Out SISO Shift Register

The shift register, which allows serial input and produces serial output is known as Serial In – Serial Out **SISO** shift register. The **block diagram** of 3-bit SISO shift register is shown in the following figure.



This block diagram consists of three D flip-flops, which are **cascaded**. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we can receive the bits serially from the output of right most D flip-flop. Hence, this output is also called as **serial output**.

Example

Let us see the working of 3-bit SISO shift register by sending the binary information **“011”** from LSB to MSB serially at the input.

Assume, initial status of the D flip-flops from leftmost to rightmost is Q_{2}Q_{1}Q_{0}=000. We can understand the **working of 3-bit SISO shift register** from the following table.

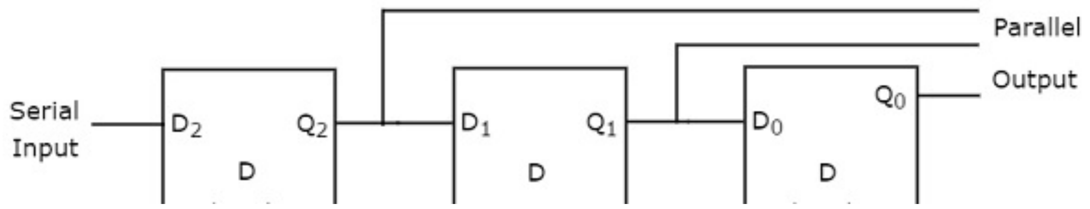
No of positive edge of Clock	Serial Input	Q_2	Q_1	Q_0
0	-	0	0	0
1	1LSB	1	0	0
2	1	1	1	0
3	0MSB	0	1	1LSB
4	-	-	0	1
5	-	-	-	0MSB

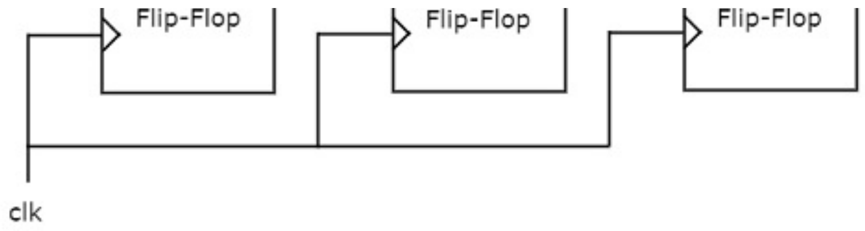
The initial status of the D flip-flops in the absence of clock signal is Q_{2}Q_{1}Q_{0}=000. Here, the serial output is coming from Q_{0}. So, the LSB 1 is received at 3rd positive edge of clock and the MSB 0 is received at 5th positive edge of clock.

Therefore, the 3-bit SISO shift register requires five clock pulses in order to produce the valid output. Similarly, the **N-bit SISO shift register** requires **2N-1** clock pulses in order to shift ‘N’ bit information.

Serial In - Parallel Out SIPO Shift Register

The shift register, which allows serial input and produces parallel output is known as Serial In – Parallel Out **SIPO** shift register. The **block diagram** of 3-bit SIPO shift register is shown in the following figure.





This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can send the bits serially from the input of left most D flip-flop. Hence, this input is also called as **serial input**. For every positive edge triggering of clock signal, the data shifts from one stage to the next. In this case, we can access the outputs of each D flip-flop in parallel. So, we will get **parallel outputs** from this shift register.

Example

Let us see the working of 3-bit SIPO shift register by sending the binary information **“011”** from LSB to MSB serially at the input.

Assume, initial status of the D flip-flops from leftmost to rightmost is Q_{2}Q_{1}Q_{0}=000. Here, Q_{2} & Q_{0} are MSB & LSB respectively. We can understand the **working of 3-bit SIPO shift register** from the following table.

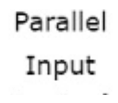
No of positive edge of Clock	Serial Input	Q_2MSB	Q_1	Q_0LSB
0	-	0	0	0
1	1LSB	1	0	0
2	1	1	1	0
3	0MSB	0	1	1

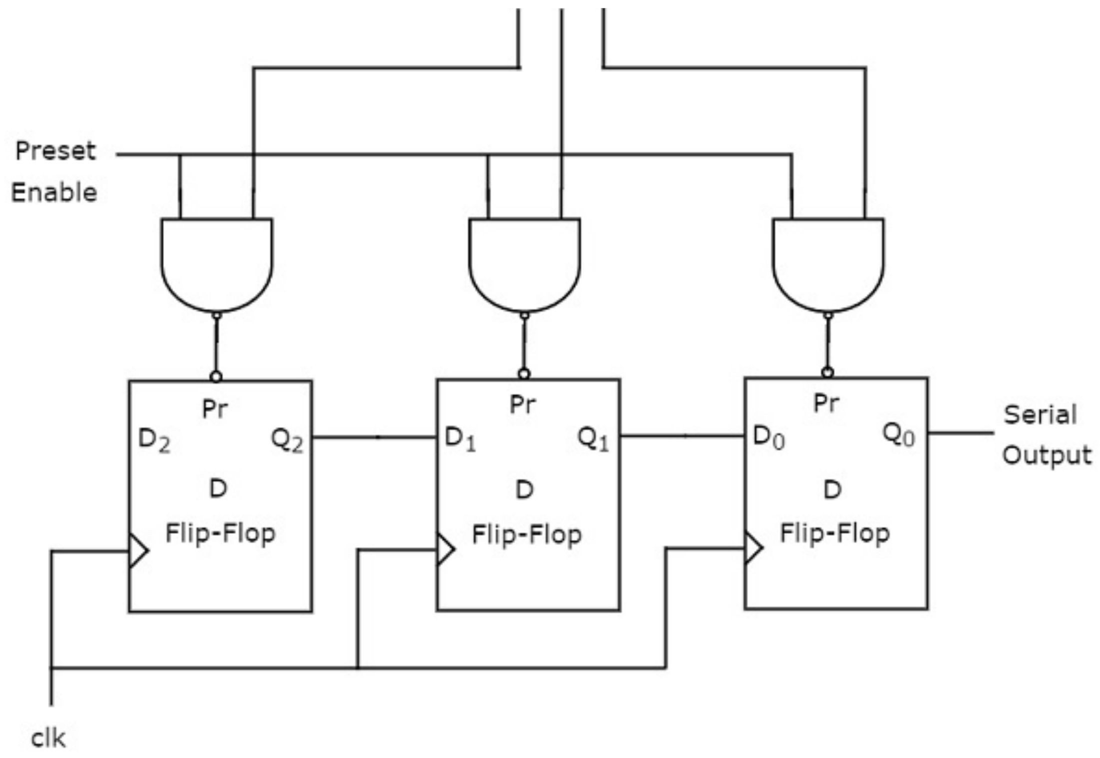
The initial status of the D flip-flops in the absence of clock signal is Q_{2}Q_{1}Q_{0}=000. The binary information **“011”** is obtained in parallel at the outputs of D flip-flops for third positive edge of clock.

So, the 3-bit SIPO shift register requires three clock pulses in order to produce the valid output. Similarly, the **N-bit SIPO shift register** requires **N** clock pulses in order to shift ‘N’ bit information.

Parallel In – Serial Out PISO Shift Register

The shift register, which allows parallel input and produces serial output is known as Parallel In – Serial Out **PISO** shift register. The **block diagram** of 3-bit PISO shift register is shown in the following figure.





This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1. For every positive edge triggering of clock signal, the data shifts from one stage to the next. So, we will get the **serial output** from the right most D flip-flop.

Example

Let us see the working of 3-bit PISO shift register by applying the binary information **"011"** in parallel through preset inputs.

Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be Q_{2}Q_{1}Q_{0}=011. We can understand the **working of 3-bit PISO shift register** from the following table.

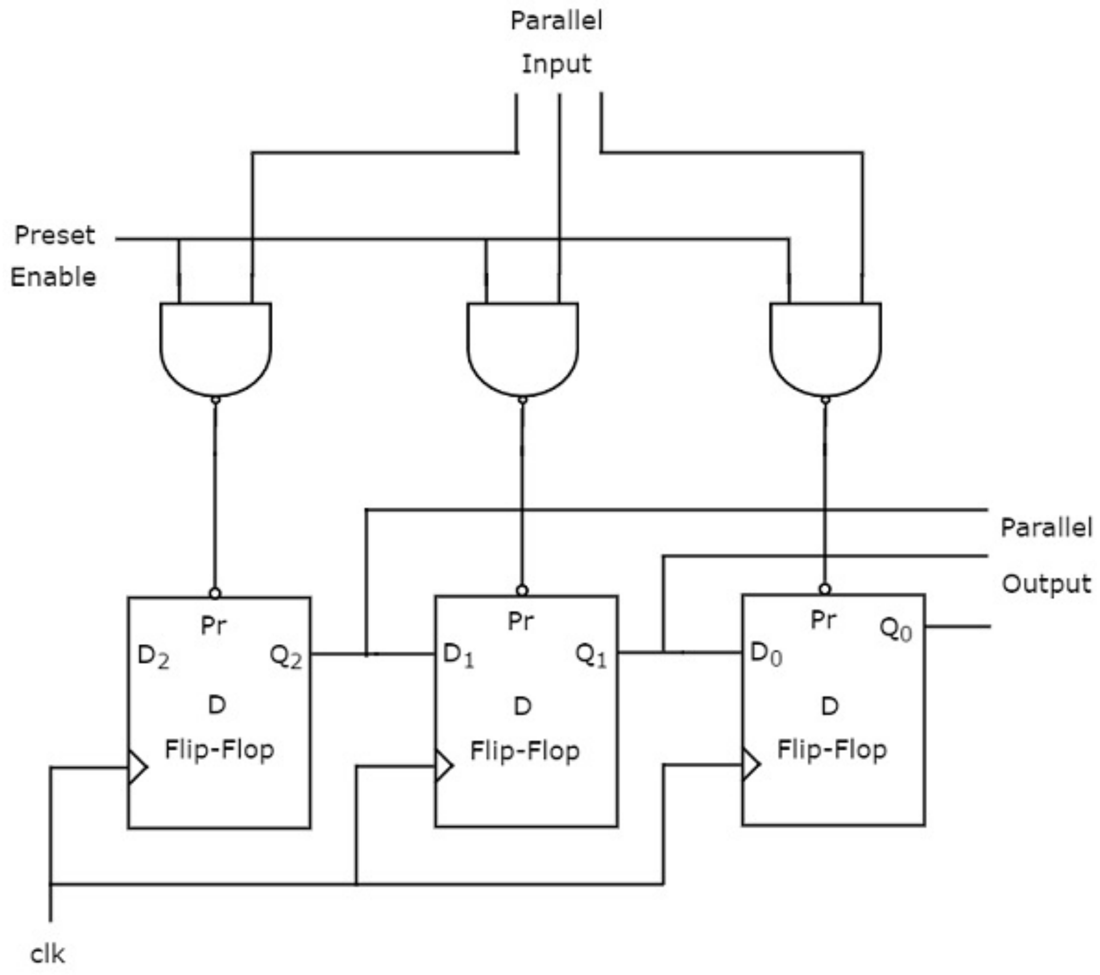
No of positive edge of Clock	Q_2	Q_1	Q_0
0	0	1	1LSB
1	-	0	1
2	-	-	0LSB

Here, the serial output is coming from Q_{0}. So, the LSB 1 is received before applying positive edge of clock and the MSB 0 is received at 2nd positive edge of clock.

Therefore, the 3-bit PISO shift register requires two clock pulses in order to produce the valid output. Similarly, the **N-bit PISO shift register** requires **N-1** clock pulses in order to shift 'N' bit information.

Parallel In - Parallel Out PIPO Shift Register

The shift register, which allows parallel input and produces parallel output is known as Parallel In – Parallel Out **PIPO** shift register. The **block diagram** of 3-bit PIPO shift register is shown in the following figure.



This circuit consists of three D flip-flops, which are cascaded. That means, output of one D flip-flop is connected as the input of next D flip-flop. All these flip-flops are synchronous with each other since, the same clock signal is applied to each one.

In this shift register, we can apply the **parallel inputs** to each D flip-flop by making Preset Enable to 1. We can apply the parallel inputs through preset or clear. These two are asynchronous inputs. That means, the flip-flops produce the corresponding outputs, based on the values of asynchronous inputs. In this case, the effect of outputs is independent of clock transition. So, we will get the **parallel outputs** from each D flip-flop.

Example

Let us see the working of 3-bit PIPO shift register by applying the binary information **“011”** in parallel through preset inputs.

Since the preset inputs are applied before positive edge of Clock, the initial status of the D flip-flops from leftmost to rightmost will be Q_{2}Q_{1}Q_{0}=011. So, the binary information **“011”** is obtained in parallel at the outputs of D flip-flops before applying positive edge of clock.

Therefore, the 3-bit PIPO shift register requires zero clock pulses in order to produce the valid output. Similarly, the **N-bit PIPO shift register** doesn’t require any clock pulse in order to shift ‘N’ bit information.

In previous chapter, we discussed four types of shift registers. Based on the requirement, we can use one of those shift registers. Following are the applications of shift registers.

- Shift register is used as **Parallel to serial converter**, which converts the parallel data into serial data. It is utilized at the transmitter section after Analog to Digital Converter ADC block.
- Shift register is used as **Serial to parallel converter**, which converts the serial data into parallel data. It is utilized at the receiver section before Digital to Analog Converter DAC block.
- Shift register along with some additional gates generate the sequence of zeros and ones. Hence, it is used as **sequence generator**.
- Shift registers are also used as **counters**. There are two types of counters based on the type of output from right most D flip-flop is connected to the serial input. Those are Ring counter and Johnson Ring counter.

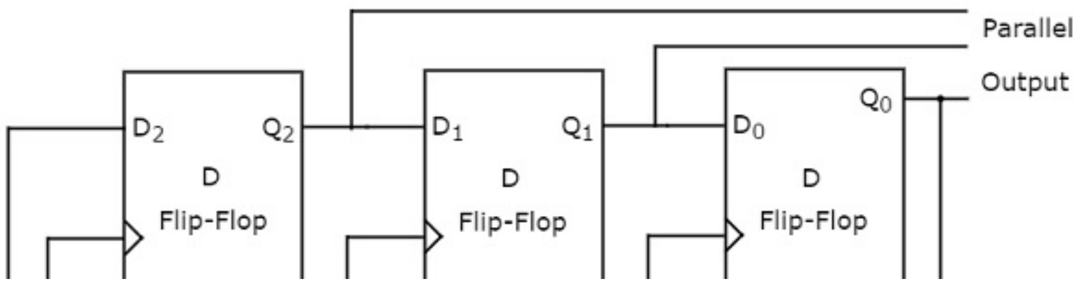
In this chapter, let us discuss about these two counters one by one.

Ring Counter

In previous chapter, we discussed the operation of Serial In - Parallel Out **SIPO** shift register. It accepts the data from outside in serial form and it requires ‘N’ clock pulses in order to shift ‘N’ bit data.

Similarly, **‘N’ bit Ring counter** performs the similar operation. But, the only difference is that the output of rightmost D flip-flop is given as input of leftmost D flip-flop instead of applying data from outside. Therefore, Ring counter produces a sequence of states pattern of zeros and ones and it repeats for every **‘N’ clock cycles**.

The **block diagram** of 3-bit Ring counter is shown in the following figure.



The 3-bit Ring counter contains only a 3-bit SIPO shift register. The output of rightmost D flip-flop is connected to serial input of left most D flip-flop.

Assume, initial status of the D flip-flops from leftmost to rightmost is Q_{2}Q_{1}Q_{0}=001. Here, Q_{2} & Q_{0} are MSB & LSB respectively. We can understand the **working of Ring counter** from the following table.

No of positive edge of Clock	Serial Input = Q_0	Q_2MSB	Q_1	Q_0LSB
0	-	0	0	1
1	1	1	0	0
2	0	0	1	0
3	0	0	0	1

The initial status of the D flip-flops in the absence of clock signal is Q_{2}Q_{1}Q_{0}=001. This status repeats for every three positive edge transitions of clock signal.

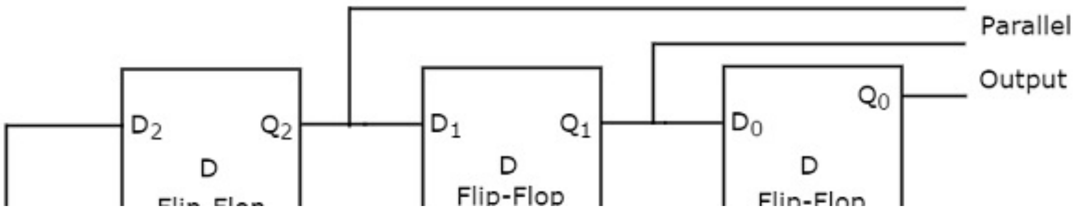
Therefore, the following **operations** take place for every positive edge of clock signal.

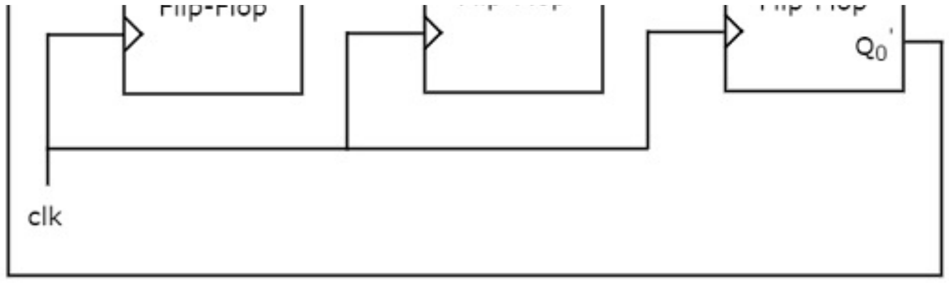
- Serial input of first D flip-flop gets the previous output of third flip-flop. So, the present output of first D flip-flop is equal to the previous output of third flip-flop.
- The previous outputs of first and second D flip-flops are right shifted by one bit. That means, the present outputs of second and third D flip-flops are equal to the previous outputs of first and second D flip-flops.

Johnson Ring Counter

The operation of **Johnson Ring counter** is similar to that of Ring counter. But, the only difference is that the complemented output of rightmost D flip-flop is given as input of leftmost D flip-flop instead of normal output. Therefore, 'N' bit Johnson Ring counter produces a sequence of states pattern of zeros and ones and it repeats for every **'2N' clock cycles**.

Johnson Ring counter is also called as **Twisted Ring counter** and switch tail Ring counter. The **block diagram** of 3-bit Johnson Ring counter is shown in the following figure.





The 3-bit Johnson Ring counter also contains only a 3-bit SIPO shift register. The complemented output of rightmost D flip-flop is connected to serial input of left most D flip-flop.

Assume, initially all the D flip-flops are cleared. So, Q_{2}Q_{1}Q_{0}=000. Here, Q_{2} & Q_{0} are MSB & LSB respectively. We can understand the **working** of Johnson Ring counter from the following table.

No of positive edge of Clock	Serial Input = Q_0	Q_2MSB	Q_1	Q_0LSB
0	-	0	0	0
1	1	1	0	0
2	1	1	1	0
3	1	1	1	1
4	0	0	1	1
5	0	0	0	1
6	0	0	0	0

The initial status of the D flip-flops in the absence of clock signal is Q_{2}Q_{1}Q_{0}=000. This status repeats for every six positive edge transitions of clock signal.

Therefore, the following **operations** take place for every positive edge of clock signal.

- Serial input of first D flip-flop gets the previous complemented output of third flip-flop. So, the present output of first D flip-flop is equal to the previous complemented output of third flip-flop.
- The previous outputs of first and second D flip-flops are right shifted by one bit. That means, the present outputs of second and third D flip-flops are equal to the previous outputs of first and second D flip-flops.

In previous two chapters, we discussed various shift registers & **counters using D flipflops**. Now, let us discuss various counters using T flip-flops. We know that T flip-flop toggles the output either for every positive edge of clock signal or for negative edge of clock signal.

An ‘N’ bit binary counter consists of ‘N’ T flip-flops. If the counter counts from 0 to 2^N – 1, then it is called as binary **up counter**. Similarly, if the counter counts down from 2^N – 1 to 0, then it is called as binary **down counter**.

There are two **types of counters** based on the flip-flops that are connected in synchronous or not.

- Asynchronous counters
- Synchronous counters

Asynchronous Counters

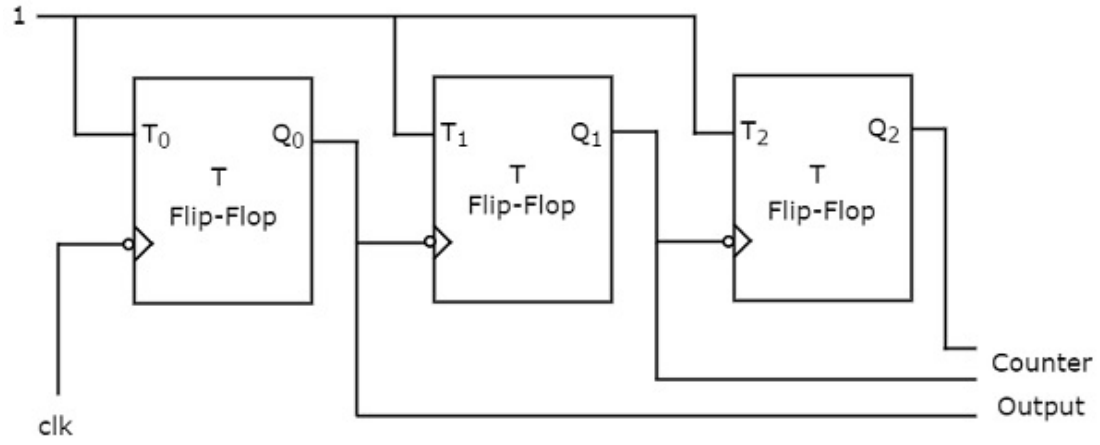
If the flip-flops do not receive the same clock signal, then that counter is called as **Asynchronous counter**. The output of system clock is applied as clock signal only to first flip-flop. The remaining flip-flops receive the clock signal from output of its previous stage flip-flop. Hence, the outputs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

Now, let us discuss the following two counters one by one.

- Asynchronous Binary up counter
- Asynchronous Binary down counter

Asynchronous Binary Up Counter

An ‘N’ bit Asynchronous binary up counter consists of ‘N’ T flip-flops. It counts from 0 to 2^N – 1. The **block diagram** of 3-bit Asynchronous binary up counter is shown in the following figure.



The 3-bit Asynchronous binary up counter contains three T flip-flops and the T-input of all the flip-flops are connected to ‘1’. All these flip-flops are negative edge triggered but the outputs change asynchronously. The clock signal is directly applied to the first T flip-flop. So, the output of first T flip-flop **toggles** for every negative edge of clock signal.

The output of first T flip-flop is applied as clock signal for second T flip-flop. So, the output of second T flip-flop toggles for every negative edge of output of first T flip-flop. Similarly, the output of third T flip-flop toggles for every negative edge of output of second T flip-flop, since the output of second T flip-flop acts as the clock signal for third T flip-flop.

Assume the initial status of T flip-flops from rightmost to leftmost is

Q_{2}Q_{1}Q_{0}=000. Here, Q_{2} & Q_{0} are MSB & LSB respectively. We can understand the **working** of 3-bit asynchronous binary counter from the following table.

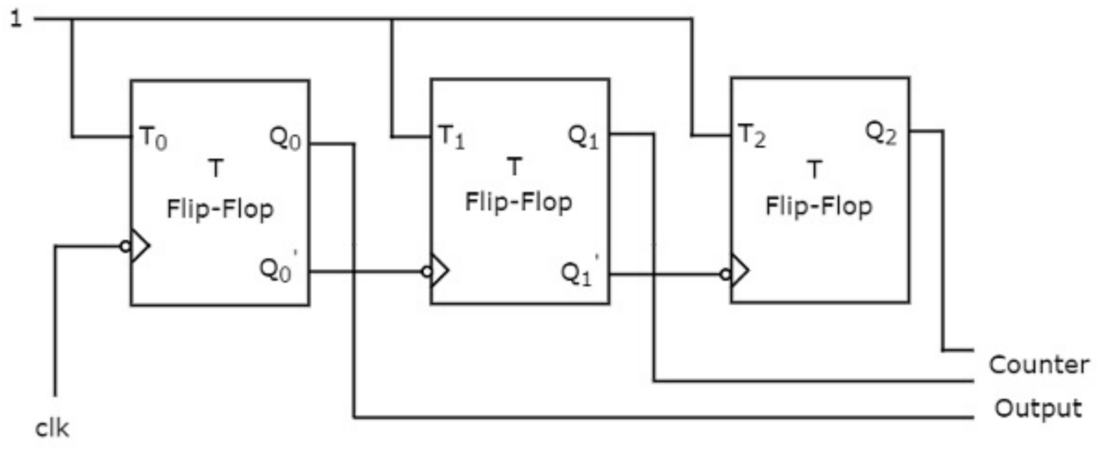
No of negative edge of Clock	Q_0LSB	Q_1	Q_2MSB
0	0	0	0
1	1	0	0
2	0	1	0
3	1	1	0
4	0	0	1
5	1	0	1
6	0	1	1
7	1	1	1

Here Q_{0} toggled for every negative edge of clock signal. Q_{1} toggled for every Q_{0} that goes from 1 to 0, otherwise remained in the previous state. Similarly, Q_{2} toggled for every Q_{1} that goes from 1 to 0, otherwise remained in the previous state.

The initial status of the T flip-flops in the absence of clock signal is Q_{2}Q_{1}Q_{0}=000. This is incremented by one for every negative edge of clock signal and reached to maximum value at 7th negative edge of clock signal. This pattern repeats when further negative edges of clock signal are applied.

Asynchronous Binary Down Counter

An 'N' bit Asynchronous binary down counter consists of 'N' T flip-flops. It counts from $2^N - 1$ to 0. The **block diagram** of 3-bit Asynchronous binary down counter is shown in the following figure.



The block diagram of 3-bit Asynchronous binary down counter is similar to the block diagram of 3-bit Asynchronous binary up counter. But, the only difference is that instead of connecting the normal outputs of one stage flip-flop as clock signal for next stage flip-flop, connect the **complemented outputs** of one stage flip-flop as clock signal for next stage flip-flop. Complemented output goes from 1 to 0 is same as the normal output goes from 0 to 1.

Assume the initial status of T flip-flops from rightmost to leftmost is Q_{2}Q_{1}Q_{0}=000. Here, Q_{2} & Q_{0} are MSB & LSB respectively. We can understand the **working** of 3-bit asynchronous binary down counter from the following table.

No of negative edge of Clock	Q_0LSB	Q_1	Q_2MSB
0	0	0	0
1	1	1	1
2	0	1	1
3	1	0	1
4	0	0	1
5	1	1	0
6	0	1	0
7	1	0	0

Here Q_{0} toggled for every negative edge of clock signal. Q_{1} toggled for every Q_{0} that goes from 0 to 1, otherwise remained in the previous state. Similarly, Q_{2} toggled for every Q_{1} that goes from 0 to 1, otherwise remained in the previous state.

The initial status of the T flip-flops in the absence of clock signal is Q_{2}Q_{1}Q_{0}=000. This is decremented by one for every negative edge of clock signal and reaches to the same value at 8^{th} negative edge of clock signal. This pattern repeats when further negative edges of clock signal are applied.

Synchronous Counters

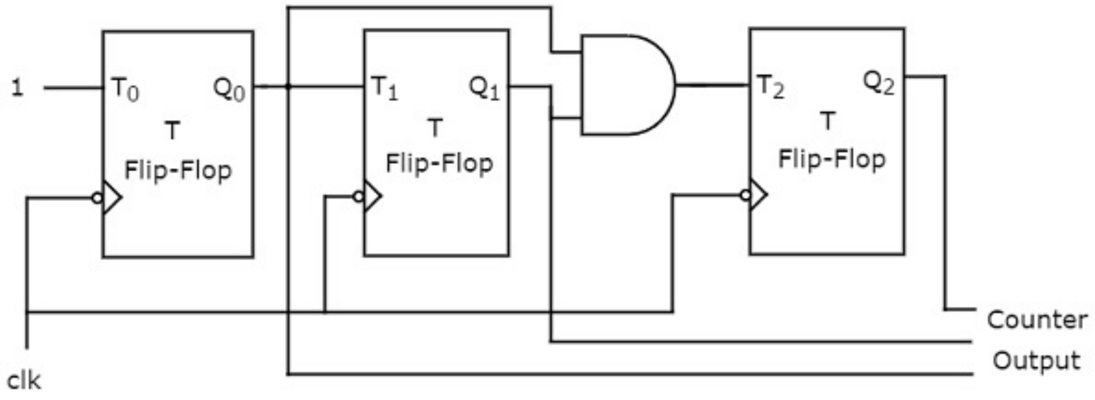
If all the flip-flops receive the same clock signal, then that counter is called as **Synchronous counter**. Hence, the outputs of all flip-flops change affect at the same time.

Now, let us discuss the following two counters one by one.

- Synchronous Binary up counter
- Synchronous Binary down counter

Synchronous Binary Up Counter

An ‘N’ bit Synchronous binary up counter consists of ‘N’ T flip-flops. It counts from 0 to $2^N - 1$. The **block diagram** of 3-bit Synchronous binary up counter is shown in the following figure.

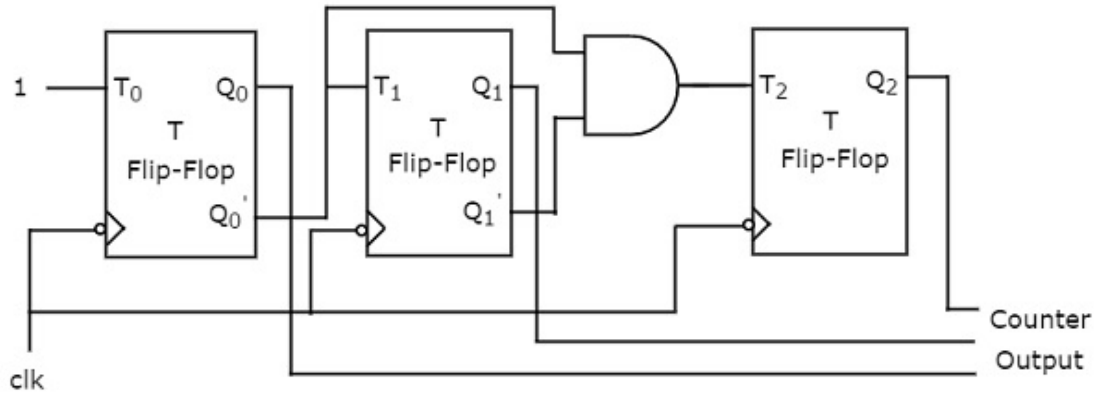


The 3-bit Synchronous binary up counter contains three T flip-flops & one 2-input AND gate. All these flip-flops are negative edge triggered and the outputs of flip-flops change affect synchronously. The T inputs of first, second and third flip-flops are 1, Q_{0} & Q_{1}Q_{0} respectively.

The output of first T flip-flop **toggles** for every negative edge of clock signal. The output of second T flip-flop toggles for every negative edge of clock signal if Q_{0} is 1. The output of third T flip-flop toggles for every negative edge of clock signal if both Q_{0} & Q_{1} are 1.

Synchronous Binary Down Counter

An ‘N’ bit Synchronous binary down counter consists of ‘N’ T flip-flops. It counts from $2^N - 1$ to 0. The **block diagram** of 3-bit Synchronous binary down counter is shown in the following figure.



The 3-bit Synchronous binary down counter contains three T flip-flops & one 2-input AND gate. All these flip-flops are negative edge triggered and the outputs of flip-flops change affect synchronously. The T inputs of first, second and third flip-flops are 1, {Q_{0}}' &' {Q_{1}}'{Q_{0}}' respectively.

The output of first T flip-flop **toggles** for every negative edge of clock signal. The output of second T flip-flop toggles for every negative edge of clock signal if {Q_{0}}' is 1. The output of third T flip-flop toggles for every negative edge of clock signal if both {Q_{1}}' & {Q_{0}}' are 1.

We know that synchronous sequential circuits change affect their states for every positive or negative transition of the clock signal based on the input. So, this behavior of synchronous sequential circuits can be represented in the graphical form and it is known as **state diagram**.

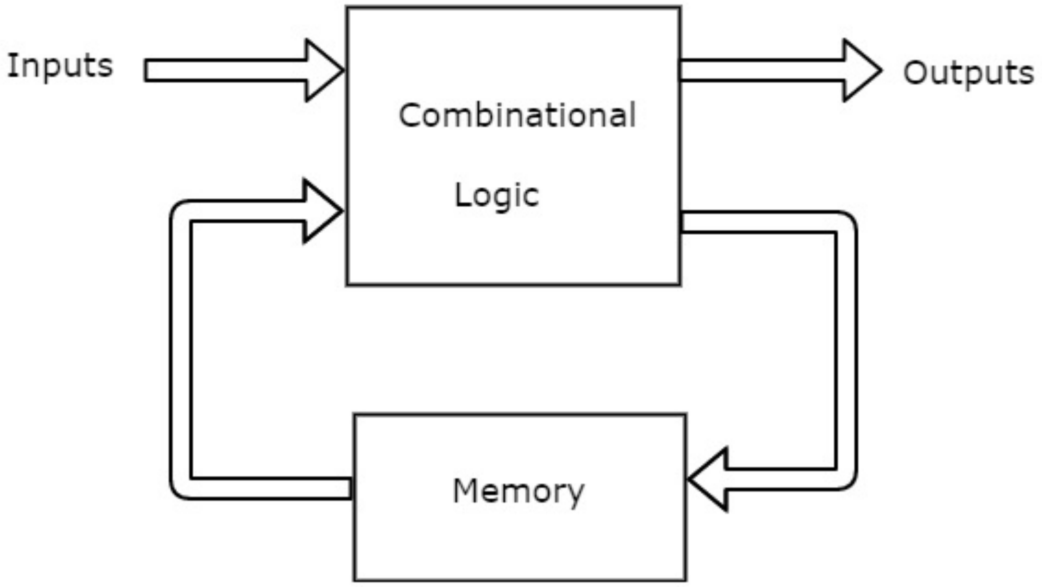
A synchronous sequential circuit is also called as **Finite State Machine** FSM, if it has finite number of states. There are two types of FSMs.

- Mealy State Machine
- Moore State Machine

Now, let us discuss about these two state machines one by one.

Mealy State Machine

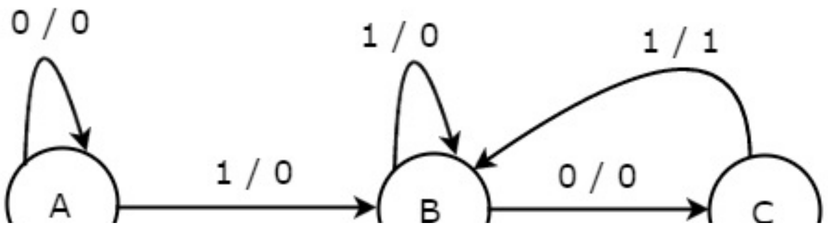
A Finite State Machine is said to be Mealy state machine, if outputs depend on both present inputs & present states. The **block diagram** of Mealy state machine is shown in the following figure.

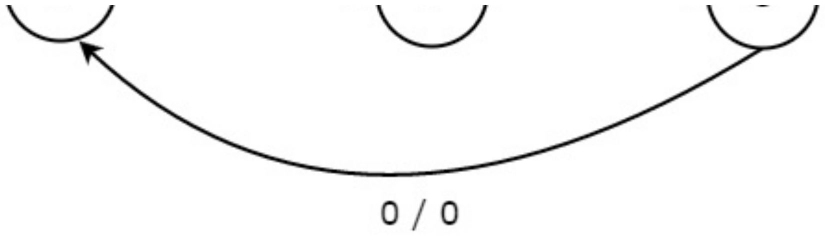


As shown in figure, there are two parts present in Mealy state machine. Those are combinational logic and memory. Memory is useful to provide some or part of previous outputs **present states** as inputs of combinational logic.

So, based on the present inputs and present states, the Mealy state machine produces outputs. Therefore, the outputs will be valid only at positive or negative transition of the clock signal.

The **state diagram** of Mealy state machine is shown in the following figure.



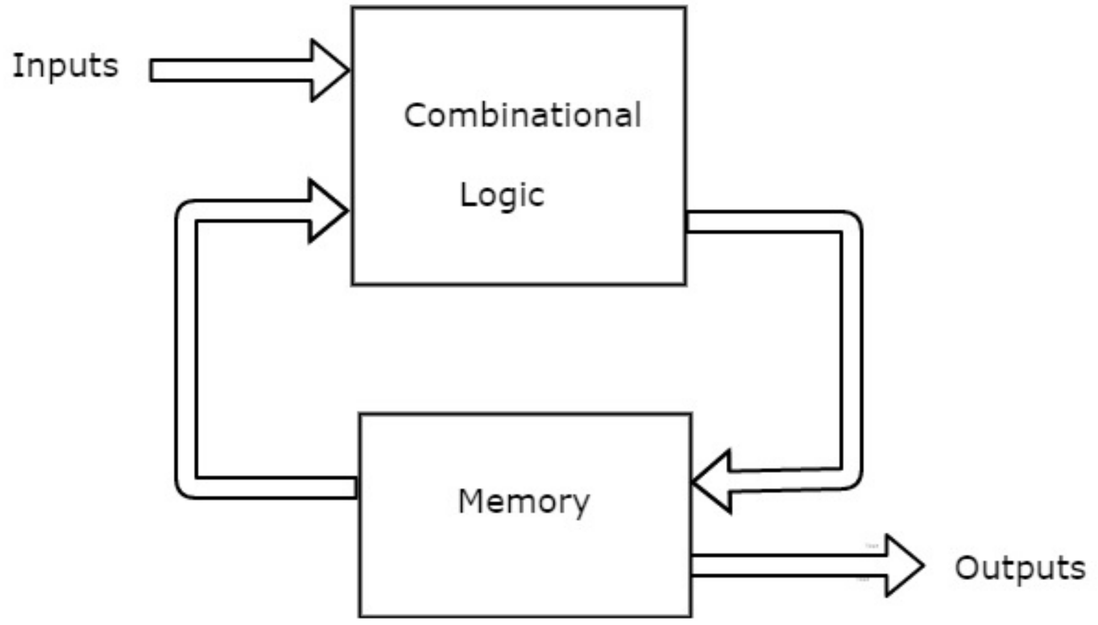


In the above figure, there are three states, namely A, B & C. These states are labelled inside the circles & each circle corresponds to one state. Transitions between these states are represented with directed lines. Here, 0 / 0, 1 / 0 & 1 / 1 denotes **input** / **output**. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Mealy state machine is less than or equal to the number of states required in Moore state machine. There is an equivalent Moore state machine for each Mealy state machine.

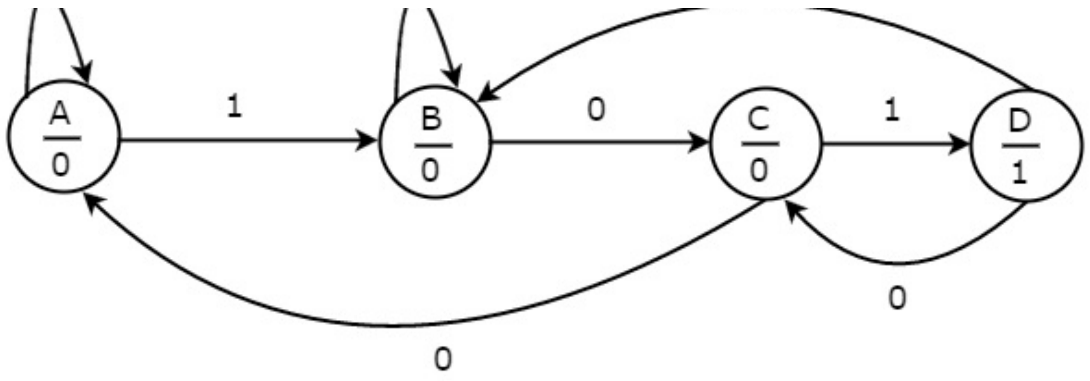
Moore State Machine

A Finite State Machine is said to be Moore state machine, if outputs depend only on present states. The **block diagram** of Moore state machine is shown in the following figure.



As shown in figure, there are two parts present in Moore state machine. Those are combinational logic and memory. In this case, the present inputs and present states determine the next states. So, based on next states, Moore state machine produces the outputs. Therefore, the outputs will be valid only after transition of the state.

The **state diagram** of Moore state machine is shown in the following figure.



In the above figure, there are four states, namely A, B, C & D. These states and the respective outputs are labelled inside the circles. Here, only the input value is labeled on each transition. In the above figure, there are two transitions from each state based on the value of input, x.

In general, the number of states required in Moore state machine is more than or equal to the number of states required in Mealy state machine. There is an equivalent Mealy state machine for each Moore state machine. So, based on the requirement we can use one of them.

Every **digital system** can be partitioned into two parts. Those are data path digital circuits and control circuits. Data path circuits perform the functions such as storing of binary information data and transfer of data from one system to the other system. Whereas, control circuits determine the flow of operations of digital circuits.

It is difficult to describe the behavior of large state machines using state diagrams. To overcome this difficulty, Algorithmic State Machine ASM charts can be used. **ASM charts** are similar to flow charts. They are used to represent the flow of tasks to be performed by data path circuits and control circuits.

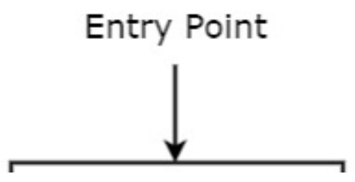
Basic Components of ASM charts

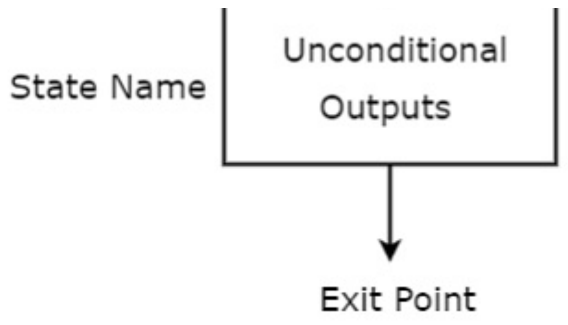
Following are the three basic components of ASM charts.

- State box
- Decision box
- Conditional output box

State box

State box is represented in rectangular shape. Each state box represents one state of the sequential circuit. The **symbol** of state box is shown in the following figure.

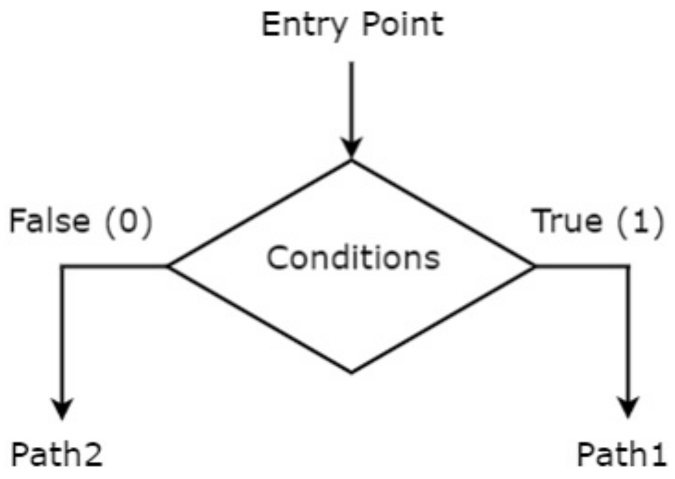




It is having one entry point and one exit point. Name of the state is placed to the left of state box. The unconditional outputs corresponding to that state can be placed inside state box. **Moore** state machine outputs can also be placed inside state box.

Decision box

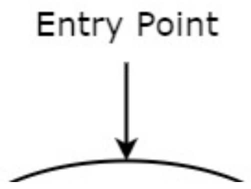
Decision box is represented in diamond shape. The **symbol** of decision box is shown in the following figure.

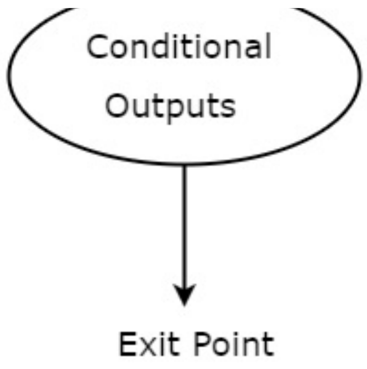


It is having one entry point and two exit paths. The inputs or Boolean expressions can be placed inside the decision box, which are to be checked whether they are true or false. If the condition is true, then it will prefer path1. Otherwise, it will prefer path2.

Conditional output box

Conditional output box is represented in oval shape. The **symbol** of conditional output box is shown in the following figure.





It is also having one entry point and one exit point similar to state box. The conditional outputs can be placed inside state box. In general, **Mealy** state machine outputs are represented inside conditional output box. So, based on the requirement, we can use the above components properly for drawing ASM charts.