

UNIT - I
BINARY SYSTEMS

Digital Systems:

A Digital system is defined as an interconnection of digital modules, that manipulate discrete elements of information which is represented internally in the binary form.

Digital systems are playing a vital role in our daily life because of which we are referring the present technological period as digital age.

Fast and reliable solutions using switching techniques proved the tremendous power and usefulness of digital systems.

Digital systems are used in communication, business transaction, traffic control, space craft guidance, medical treatment, weather monitoring, the internet and many other commercial, industrial and scientific enterprises. Today we are enjoying digital telephones, digital televisions, digital versatile discs, digital cameras, handheld devices and digital computers because of the development in digital technology.

These devices have Graphical user interfaces (GUI's), which makes the user to interact with the digital system easily by making the choices from the Menu (or) a group of icons.

A GUI is nothing but a program that is written by the user according to the application.

The characteristics of the digital systems is given below.

Characteristics of a digital system:

- * Digital system manipulates discrete elements of information.
- * Discrete elements of information is nothing but digits such as 10 decimal digits, 26 letters of alphabets and so on.
- * In most of the digital systems the signals contain only two values only, they are 0, 1. Since they are having only two values in number system, we can call the number system as binary number system.
- * Discrete elements of information are represented with a group of bits called binary codes, where a bit represents a binary digit (either '0' or '1'). For example the decimal digits 0 to 9 are represented in a digital system with a 4-bit code called BCD (Binary Coded Decimal).
- * Digital systems like a digital computer is a programmable device which can be programmed to perform a variety of tasks.
- * The digital system like a digital computer is an interconnection of digital modules. The major units of a digital computer are a central processing unit, memory unit and input output unit. The CPU performs the arithmetic and logical operations. The programs and data prepared by the user can be entered into the system through an input device called keyboard and stored in the memory. The monitor and printer are the examples for output devices.

Advantages of Digital systems over Analog systems:

- 1) Flexibility: Digital systems are more flexible to design as its design involves a set of logical steps. A digital system can be reconfigured for some other application simply by changing the software program. Where as in analog systems if we want to perform any other operation we have to change the hardware components.
- 2) Ease of design: Digital systems are easy to design than analog.
- 3) Accuracy: The accuracy of the digital systems is much higher than that of the analog systems. In analog systems the temperature variations and component tolerance etc are major problems due to which the high accuracy is not possible in analog systems.
- 4) Size and reliability: The digital systems are small in size, more reliable and less expensive when compare to analog systems.
- 5) Programmability: Now-a-days, digital design is carried out by writing programs in hardware description language (HDL). These languages allow to stimulate and test the performance of digital circuits. This feature is very useful in designing critical digital systems.
- 6) Reproducibility of the result: The output of analog systems vary with temperature, component aging, power supply voltage, component tolerance and other factors. So it is difficult to produce the same result everytime even with same set of inputs and circuits components. This is not the case with digital systems. They always produce exactly same results with same

set of inputs and circuit components.

→ Upgrading Technology: As digital technology is becoming more and more popular, more research is going on in this field. so, the technological upgrade is expected in the digital world.

Number Systems:

Number system is the basis of counting various items.

On hearing the term 'number' all of us are familiar with decimal number system that includes ten digits: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

Modern computers are operated and communicated with binary number system that includes only two digits: 0, 1.

For example let us take a decimal number $(24)_{10}$. This number can be represented in binary as $(11000)_2$. In this example, for decimal number $(24)_{10}$ we require two digits and its binary equivalent number $(11000)_2$ requires five digits. As we go on increasing the decimal number value the number of binary digits in its equivalent binary number will also be increasing.

This fact gives rise to three new number systems that represent a binary number in a compressed form. They are

1) Octal

2) Hexa Decimal

3) Binary Coded Decimal (BCD).

These number systems are widely used to compress the long strings of binary number.

(2)

Radix: (or) Base:

It specifies the number of symbols used for a particular number system.

Radix point:

In any number system, the radix point specifies the dividing line between the integer part and fractional part.

The Decimal Number System:

* The decimal number system contains ten unique symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. That means the radix (or) base of the decimal number system is 10.

* In decimal number system we can express any decimal number in terms of units, tens, hundreds, thousands --- etc.

* For example let us consider a decimal number $(5678.2)_{10}$. It can be represented as

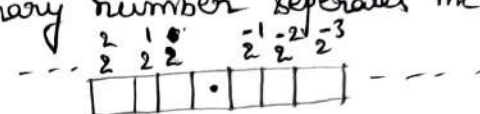
$$\begin{aligned}(5678.2)_{10} &= 5000 + 600 + 70 + 8 + 0.2 \\ &= 5 \times 10^3 + 6 \times 10^2 + 7 \times 10^1 + 8 \times 10^0 + 2 \times 10^{-1}\end{aligned}$$

* The position of a digit with reference to the decimal point determines the weight of that particular digit. The sum of all digits multiplied by the respective weights gives the total number.

* In a decimal number the left most digit which has the highest weight is called as Most Significant Digit (MSD) and the right most digit that has the lowest weight is called as Least Significant Digit (LSD). The decimal point separates the integer and fractional parts.

* The decimal number system can also be called as radix-10 number system or base-10 number system.

Binary Number System:

- * The base (or) radix of this number system is 2.
Hence it has two independent symbols, they are 0 and 1.
- * A binary digit present in a binary number is called as a bit. A bit can be either 0 (or) 1.
- * The binary number system can also be called as radix-2 number system (or) base-2 number system.
- * The binary point in a binary number separates the integer part and fractional part. 
The diagram shows a sequence of boxes representing digits. Above the first four boxes are weights $2^2, 2^1, 2^0, 2^{-1}$. Above the next three boxes are weights $2^{-2}, 2^{-3}, 2^{-4}$. A dot is placed between the fourth and fifth boxes, representing the binary point.
- * In binary number system the weights are expressed in terms of the powers of 2.
- * By adding each digit multiplied by its respective weight in the given binary number, we can obtain the decimal equivalent number.

Ex: $(1101.101)_2$ Convert into decimal equivalent.

Sol

$$\begin{aligned} (1101.101)_2 &= 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ &\quad + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1 + 1 \times 0.5 + 0 \times 0.25 \\ &\quad + 1 \times 0.125 \\ &= 8 + 4 + 0 + 1 + 0.5 + 0 + 0.125 \\ &= (13.625)_{10} \end{aligned}$$

Practice problems: 1) Convert $(10111.1001)_2$ into its decimal equivalent.

2) Convert $(100111.10111)_2$ into its decimal equivalent.

3) Convert $(101011.11011)_2$ into its decimal equivalent.

Octal Number System :

* The base (or) radix of the octal number system is 8.

Hence this number system has 8 independent symbols : 0, 1, 2, 3, 4, 5, 6 and 7.

* This number system is also called as base-8 (or) radix-8 number system.

* In the octal number system the weights are represented in terms of powers of 8.

---	3	2	1	0	-1	-2	-3	-4	---
---	8	8	8	8	8	8	8	8	---
---									---
---									---

* By adding each digit of an octal number multiplied by its respective weight we can obtain the decimal equivalent of the given octal number.

Example : Convert $(2763.45)_8$ into its equivalent decimal.

Sol \rightarrow

$$\begin{array}{ccccccc} 8^3 & 8^2 & 8^1 & 8^0 & 8^{-1} & 8^{-2} & \\ (2 & 7 & 6 & 3 & . & 4 & 5)_8 \end{array} = 2 \times 8^3 + 7 \times 8^2 + 6 \times 8^1 + 3 \times 8^0 + 4 \times 8^{-1} + 5 \times 8^{-2}$$
$$= 1024 + 448 + 48 + 3 + 0.5 + 0.078125$$
$$= (1523.578125)_{10}$$

Practice problems : convert the following octal numbers into their respective decimal equivalent numbers

i) $(762.345)_8$ ii) $(3467.25)_8$ iii) $(675.36)_8$

Hexa decimal Number system :

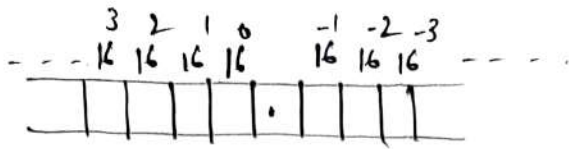
* The base (or) radix of the hexadecimal number system is 16.

Hence this number system has 16 independent symbols : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.

* This number system is also called as base-16 number system (or)

Radix-16 number system.

- * In the hexadecimal number system the weights are represented in terms of powers of 16.



- * By adding each digit in the given hexadecimal number multiplied by its respective weight we can obtain the decimal equivalent for the given hexadecimal number.

Example: Convert $(9DEB.7A)_{16}$ into its equivalent decimal number.

Sol^y

$$\begin{aligned} & \left(\begin{array}{ccccccc} 16^3 & 16^2 & 16^1 & 16^0 & 16^{-1} & 16^{-2} & \\ 9 & D & E & B & . & 7 & A \end{array} \right)_{16} = 9 \times 16^3 + 13 \times 16^2 + 14 \times 16^1 + \\ & \quad 11 \times 16^0 + 7 \times 16^{-1} + 10 \times 16^{-2} \\ & = 36864 + 3328 + 224 + 11 + 0.4375 + 0.0390625 \\ & = (40427.4765625)_{10} \end{aligned}$$

Practice Problems: Convert the following hexadecimal numbers into their respective decimal equivalent numbers.

i) $(6B7.A2)_{16}$ ii) $(96E.2F)_{16}$ iii) $(479.BD)_{16}$

Note: * A group of 4 binary bits is called as a 'Nibble'.

* A group of 8 binary bits is called as a 'Byte'.

* A group of 16 binary bits is called as a 'word'.

* A group of 32 binary bits is called as a 'double word'.

*** For Radix-16 number system the weights are in terms of powers of 16. By adding each digit multiplied by its respective weight we can obtain its decimal equivalent.

The following table shows the relationship among all the number-systems. In this table the first 16 numbers present in the decimal, binary, octal and hexadecimal number systems are listed.

Decimal (Base-10)	Binary (Base-2)	Octal (Base-8)	Hexa decimal (Base-16)
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Counting in radix- n (or) base- n :

In previous discussion we have seen the number systems with radix equal to 10, 2, 8 and 16. Each number system has n set of symbols. For example decimal number system has radix 10 and 10 set of symbols from 0 to 9. In binary system n is equal to 2 and it has 2 symbols from 0 to 1 (i.e 0, 1).

In general we can say that, a number represented in radix- n , has n symbols in its set and n can be any value.

For a particular radix value, the respective symbol set is shown in the below table.

Radix- r (or) Base- r	Symbols in set
2 (Binary)	0, 1
3	0, 1, 2
4	0, 1, 2, 3
5	0, 1, 2, 3, 4
6	0, 1, 2, 3, 4, 5
7	0, 1, 2, 3, 4, 5, 6
8 (Octal)	0, 1, 2, 3, 4, 5, 6, 7
9	0, 1, 2, 3, 4, 5, 6, 7, 8
10 (Decimal)	0, 1, 2, 3, 4, 5, 6, 7, 8, 9
11	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A
12	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B
13	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C
14	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D
15	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E
16	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Converting any radix number to decimal:

Consider a general number with radix- r . Let the number be

$$(A_{m-1} A_{m-2} A_{m-3} \dots A_2 A_1 A_0 \cdot A_{-1} A_{-2} A_{-3} \dots A_{-n+2} A_{-n+1} A_{-n})_r$$

Assigning weights to each digit in the above number in terms of powers of r , we get weights

$$\begin{array}{cccccccccccccccc} r^{m-1} & r^{m-2} & r^{m-3} & & r^2 & r^1 & r^0 & r^{-1} & r^{-2} & r^{-3} & & r^{-n+2} & r^{-n+1} & r^{-n} \\ (A_{m-1} & A_{m-2} & A_{m-3} & \dots & A_2 & A_1 & A_0 \cdot & A_{-1} & A_{-2} & A_{-3} & \dots & A_{-n+2} & A_{-n+1} & A_{-n}) \end{array}$$

(6)

Decimal equivalent number = $A_{m-1}r^{m-1} + A_{m-2}r^{m-2} + \dots + A_2r^2 + A_1r + A_0r^0 + A_{-1}r^{-1} + A_{-2}r^{-2} + \dots + A_{-n}r^{-n}$.

Example: Convert $(3102.12)_4$ to its equivalent decimal.

Sol) Given $(3102.12)_4$ which has radix 4. Hence the weights are assigned in terms of powers of 4.

$$\begin{array}{ccccccc} 4^3 & 4^2 & 4^1 & 4^0 & 4^{-1} & 4^{-2} & \\ (3 & 1 & 0 & 2 & . & 1 & 2) \\ & & & & & & 4 \end{array}$$

The decimal equivalent number for $(3102.12)_4$ is

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

$$= (210.375)_{10}$$

$$\therefore (3102.12)_4 = (210.375)_{10}$$

Practice problems: Convert the following into their equivalent decimal.

i) $(231.23)_4$ ii) $(310.21)_5$ iii) $(123.15)_6$

Example Convert $(654.32)_7$ to its equivalent decimal.

Sol) Since the given number $(654.32)_7$ has the radix - 7, the weights are assigned in terms of powers of 7.

$$\begin{array}{ccccccc} 7^2 & 7^1 & 7^0 & 7^{-1} & 7^{-2} & & \\ (6 & 5 & 4 & . & 3 & 2) & \\ & & & & & & 7 \end{array} = 6 \times 7^2 + 5 \times 7^1 + 4 \times 7^0 + 3 \times 7^{-1} + 2 \times 7^{-2}$$

$$= (333.4693878)_{10}$$

Practice problems Convert i) $(A98.7B)_{12}$ to its equivalent decimal

ii) $(786.35)_9$ to its equivalent decimal.

Number base conversions:

Binary to octal conversion:

We know that the base for octal number system is 8 and the base for binary number system is 2. The base for octal number is the third power of the base for binary, (i.e. $8 = 2^3$). Therefore by grouping 3 digits of binary numbers and then converting each group to its octal equivalent, we can convert the given binary number to its octal equivalent.

Example: Convert $(10101101.0111)_2$ to octal equivalent.

Solution: Given binary $(10101101.0111)_2$.

To convert a binary number to octal equivalent we need to follow the steps given below.

Step 1: Make groups of 3-bits starting from right most bit for integer part and left most bit for fractional part, append 0's at the end if necessary.

Step 2: Write the equivalent octal number for each group of 3-bits.

Step 1: $010 \mid 101 \mid 101 \mid .011 \mid 100$

Step 2: 2 5 5 . 3 4

$$\therefore (10101101.0111)_2 = (255.34)_8$$

Practice problems:

Convert the following binary numbers to their equivalent octal numbers

i) $(110011011.01011)_2$ ii) $(1010111011.101111)_2$

Octal to Binary Conversion:

The octal to binary conversion is a reverse process of the conversion of binary to octal. In this octal to binary conversion, each octal digit in the given octal number is replaced by its 3 bit binary equivalent to get the binary equivalent of it.

Example: convert $(367.04)_8$ to its equivalent binary number.

Sol) Given the octal number $(367.04)_8$.

To convert the given octal number to binary we need to follow the steps given below.

Step 1: write the equivalent 3-bit binary number for each octal digit in the given octal number.

Step 2: Remove leading and trailing 0's from the integer part and fractional parts respectively (if any)

3 6 7 . 0 4

step 1: 011 110 111 . 000 100

step 2: 11110111 . 0001

$$\therefore (367.04)_8 = (11110111.0001)_2$$

Practice problems: convert the following octal numbers to their equivalent binary numbers. i) $(73.465)_8$ ii) $(125.62)_8$

Example: convert $(35.764)_8$ to its equivalent binary.

Sol Given octal number is $(35.764)_8$.

step 1: 3 5 . 7 6 4
011 101 . 111 110 100

step 2: 11101 . 1111101

$$\therefore (35.764)_8 = (11101.1111101)_2$$

Binary to Hexadecimal Conversion:

We know that the base of the hexadecimal number system is 16 and the base of the binary number system is 2. The base of the hexadecimal number is the fourth power of the base for the binary numbers. Therefore by grouping 4 digits of binary numbers and then converting each group to its hexadecimal equivalent, we can convert binary number to its hexadecimal equivalent.

Example: Convert $(1101110110011.1010110110)_2$ to its hexadecimal equivalent.

Sol) Given binary number $(1101110110011.1010110110)_2$.

To convert a binary number to its hexadecimal equivalent we need to follow the following steps.

Step 1: make the groups of 4 bits starting from right most bit for integer part and left most bit for fractional part, append 0's at the end if necessary.

Step 2: write the equivalent hexadecimal number for each group of 4-bits.

Step 1: $0001 \mid 1011 \mid 1011 \mid 0011 \mid . \mid 1010 \mid 1101 \mid 1000$

Step 2: 1 B B 3 . A D 8

$$\therefore (1101110110011.1010110110)_2 = (1BB3.AD8)_{16}$$

Practice Problems:

Convert the following binary numbers to their hexadecimal equivalent.

i) $(010011011110.01111)_2$ ii) $(1010111011.100101)_2$

iii) $(1101101110.10010101)_2$ iv) $(101010001001.1011010)_2$

Hexa Decimal to Binary Conversion:

The hexadecimal to binary conversion is the reverse process of the conversion of binary to hexadecimal. In this hexadecimal to binary conversion, each hexadecimal digit in the given hexadecimal number is replaced by its 4-bit binary equivalent to get the binary equivalent of the given number.

Example: Convert $(7C6.BA)_{16}$ to its equivalent binary number.

sol > Given the hexadecimal number $(7C6.BA)_{16}$.

To convert the given hexadecimal number to binary we need to follow the steps given below.

Step 1: Write the equivalent 4-bit binary number for each digit in the given hexadecimal number.

Step 2: Remove the leading and trailing 0's from the integer part and fractional parts respectively (if any).

Step 1: 7 C 6 . B A
 0111 1100 0110 1011 1010

Step 2: $(11111000110.1011101)_2$

$$\therefore (7C6.BA)_{16} = (11111000110.1011101)_2$$

Example 2: Convert $(976.DAC)_{16}$ to its equivalent binary number.

sol Given hexadecimal number $(976.DAC)_{16}$

Step 1: 9 7 6 . D A C
 1001 0111 0110 1101 1010 1100

Step 2: $(100101110110.1101101011)_2$

$$\therefore (976.DAC)_{16} = (100101110110.1101101011)_2$$

Practice problems: Convert i) $(6EA5.BE)_{16}$ ii) $(986.BED)_{16}$ to binary.

Octal to Hexa decimal Conversion:

In the process to convert a given octal number to it's hexa decimal equivalent, the following steps are to be followed.

- 1) Convert the given octal number to it's binary equivalent.
- 2) Convert this binary equivalent obtained in step 1 to it's hexa decimal equivalent

Example: Convert $(617.25)_8$ to it's hexadecimal equivalent.

Sol: Given octal number is $(617.25)_8$

Step 1: write the equivalent 3 bit binary number for each octal digit in the given octal number.

Step 2: Make group of 4-bits starting from right most bit for integer part and left most bit for fractional part and append 0's at the end if necessary.

Step 3: write the equivalent hexadecimal number for each group of 4 bits.

6 1 7 . 2 5

Step 1: 110 001 111 010 101

Step 2: 0001 | 1000 | 1111 . 0101 | 0100

Step 3: 1 8 F . 5 4

$$\therefore (617.25)_8 = (18F.54)_{16}$$

Practice problems: Convert the following octal numbers to their equivalent Hexadecimal numbers.

- i) $(732.56)_8$ ii) $(265.44)_8$ iii) $(365.24)_8$

Hexadecimal to octal conversion:

To convert a hexadecimal number into octal, the following steps are to be followed.

- 1) Convert the given hexadecimal number to its equivalent binary.
- 2) Convert the binary equivalent obtained in step 1 into its octal equivalent.

Example: Convert $(79D.AE)_{16}$ into its octal equivalent.

Sol. Given hexadecimal number = $(79D.AE)_{16}$

To convert hexadecimal to octal

- 1) Write the equivalent 4-bit binary number for each digit in the given hexadecimal number
- 2) Make the group of 3 bits starting from the right most bit for the integer part and the left most bit for the fractional part, append 0's at the end if required
- 3) Write the equivalent octal number for each group of 3 bits

7 9 D . A E

step 1: 0111 1001 1101 . 1010 1110

step 2: 011 | 110 | 011 | 101 . 101 | 011 | 100

step 3: 3 6 3 5 . 5 3 4

$$\therefore (79D.AE)_{16} = (3635.534)_8$$

Practice problems: Convert the following hexadecimal numbers into octal equivalent.

i) $(Bc66.FA)_{16}$ ii) $(CD8.9F)_{16}$

ii) $(98B.EC)_{16}$ iv) $(F786.CD)_{16}$

Conversion of Decimal to any radix :

The conversion of decimal number to any radix is carried out in two steps.

- 1) The integer part has to be converted into the required radix which is accomplished by using successive division method.
- 2) The fractional part is to be converted into the desired radix which is done by using successive multiplication method.

Successive division method for integer part conversion :

The integer part of the decimal number is divided by the new radix repeatedly until the quotient is zero. The remainders are taken from bottom to top to form the new radix number.

Successive multiplication for fractional part conversion :

The fractional part of the given decimal number is multiplied by the new radix that produces a product that has an integer part and a fractional part. The integer part is collected and the fractional part is again multiplied by the new radix. This process is continued until the fractional part is equal to zero (or) until new radix number have sufficient digits. The integer part of each product is to be taken from the top to bottom to form the fractional part of the new radix number.

Example: Find the octal equivalent for the following decimal number. $(659.825)_{10}$

Sol) Given decimal number $= (659.825)_{10}$
required radix is 8, as we need to find octal equivalent.

Integer Part Conversion: (using successive division method)

$$\begin{array}{r|l} 8 & 659 \\ \hline 8 & 82-3 \\ \hline 8 & 10-2 \\ \hline & 1-2 \end{array}$$

$$\therefore (659)_{10} = (1223)_8$$

Fractional Part Conversion using successive multiplication method:

$$\begin{array}{llll} 0.825 \times 8 & = & 6.6 & \Rightarrow 6 \\ 0.6 \times 8 & = & 4.8 & \Rightarrow 4 \\ 0.8 \times 8 & = & 6.4 & \Rightarrow 6 \\ 0.4 \times 8 & = & 3.2 & \Rightarrow 3 \\ 0.2 \times 8 & = & 1.6 & \Rightarrow 1 \end{array}$$

$$\therefore (0.825)_{10} = (0.64631)_8$$

$$\therefore (659.825)_{10} = (1223.64631)_8$$

Practice problems: i) Convert $(38475)_{10}$ to binary form.

ii) Convert $(9865.374)_{10}$ to its hexadecimal equivalent.

Example: convert the following i) $(102.67)_8 = (?)_{12}$

ii) $(345.21)_6 = (?)_7$ iii) $(A98B)_{12} = (?)_3$

Sol) i) $(102.67)_8 = (?)_{12}$

To get radix-12 number from the given octal number, firstly the octal number has to be converted into the decimal equivalent.

$$\begin{array}{ccccccc} & 2 & 1 & 0 & & & -2 \\ & 8 & 8 & 8 & 8 & 8 & \\ (102.67)_8 & = & 1 \times 8^2 & + & 0 \times 8^1 & + & 2 \times 8^0 & + & 6 \times 8^{-1} & + & 7 \times 8^{-2} \\ & = & (66.859375)_{10} \end{array}$$

Now Convert this decimal equivalent number into radix-12 number.

$$12 \overline{) 66} \\ \underline{5-} 6$$

$$(66)_{10} = (56)_{12}$$

$$\begin{array}{rcl} 0.859375 \times 12 & = & 10.3125 \Rightarrow A \\ 0.3125 \times 12 & = & 3.75 \Rightarrow 3 \\ 0.75 \times 12 & = & 9.00 \Rightarrow 9 \end{array} \quad \downarrow$$

$$\therefore (0.859375)_{10} = (A39)_{12}$$

$$\therefore (66.859375)_{10} = (56.A39)_{12}$$

ii) $(345.21)_6 = (?)_7$

Sol) To get radix-7 equivalent number from the given radix-6 number, firstly convert the given radix-6 number to decimal equivalent.

$$\begin{array}{ccccccc} & 2 & 1 & 0 & & -1 & -2 \\ & 6 & 6 & 6 & 6 & 6 & \\ (345.21)_6 & = & 3 \times 6^2 & + & 4 \times 6^1 & + & 5 \times 6^0 & + & 2 \times 6^{-1} & + & 1 \times 6^{-2} \\ & = & (137.361111)_{10} \end{array}$$

Now convert this decimal equivalent number into radix-7.

$$\begin{array}{r} 7 \overline{) 137} \\ 7 \overline{) 19-4} \uparrow \\ \underline{2-} 5 \end{array}$$

$$\therefore (137)_{10} = (254)_7$$

$$\begin{aligned}
 0.3611111 \times 7 &= 2.5277777 \Rightarrow 2 \\
 0.5277777 \times 7 &= 3.6944439 \Rightarrow 3 \\
 0.6944439 \times 7 &= 4.8611073 \Rightarrow 4 \\
 0.8611073 \times 7 &= 6.0277511 \Rightarrow 6 \downarrow
 \end{aligned}$$

$$\therefore (0.3611111)_{10} = (0.2346)_7$$

$$\therefore (137.3611111)_{10} = (254.2346)_7$$

$$\text{iii) } (A98B)_{12} = (?)_3$$

To convert radix-12 number into radix-3, firstly it has to be converted into radix-10 (i.e. decimal).

$$\begin{aligned}
 \begin{matrix} 3 & 2 & 1 & 0 \\ 12 & 12 & 12 & 12 \end{matrix} \\
 A \ 9 \ 8 \ B &= 10 \times 12^3 + 9 \times 12^2 + 8 \times 12^1 + 11 \times 12^0 \\
 &= 17280 + 1296 + 96 + 11 \\
 &= (18683)_{10}
 \end{aligned}$$

Now convert this decimal equivalent number into its equivalent radix-3 number.

$$\begin{array}{r}
 3 \overline{) 18683} \\
 \underline{3 6227} \uparrow \\
 3 \overline{) 2075} \\
 \underline{3 691} \\
 3 \overline{) 230} \\
 \underline{3 76} \\
 3 \overline{) 25} \\
 \underline{3 8} \\
 3 \overline{) 2} \\
 \underline{3 2}
 \end{array}$$

$$\therefore (A98B)_{12} = (221121222)_3$$

Example: Convert $(76.275)_8$ to decimal and then to binary.

Sol

$$(76.275)_8 = (?)_{10}$$

$$\begin{array}{ccccccc} 8^1 & 8^0 & 8^{-1} & 8^{-2} & 8^{-3} & & \\ 7 & 6 & . & 2 & 7 & 5 & \\ & & & & & & \end{array} = 7 \times 8^1 + 6 \times 8^0 + 2 \times 8^{-1} + 7 \times 8^{-2} + 5 \times 8^{-3}$$

$$= (62.369140625)_{10}$$

Now $(62.369140625)_{10}$ has to be converted to binary.

$$\begin{array}{r} 2 \overline{) 62} \\ 2 \overline{) 31 - 0} \\ 2 \overline{) 15 - 1} \\ 2 \overline{) 7 - 1} \\ 2 \overline{) 3 - 1} \\ 2 \overline{) 1 - 1} \end{array}$$

$$\therefore (62)_{10} = (111110)_2$$

$$\begin{array}{l} 0.369140625 \times 2 = 0.73828125 \Rightarrow 0 \\ 0.73828125 \times 2 = 1.4765625 \Rightarrow 1 \\ 0.4765625 \times 2 = 0.953125 \Rightarrow 0 \\ 0.953125 \times 2 = 1.90625 \Rightarrow 1 \\ 0.90625 \times 2 = 1.8125 \Rightarrow 1 \\ 0.8125 \times 2 = 1.625 \Rightarrow 1 \end{array}$$

$$\therefore (0.369140625)_{10} = (0.010111)_2$$

$$\therefore (62.369140625)_{10} = (111110.01011101)_2$$

Example: If $(79)_{10} = (117)_x = (142)_y$. Then find x, y .

Sol

$$(79)_{10} = (117)_x$$

Converting $(117)_x$ into decimal

$$\begin{array}{cccc} x^2 & x^1 & x^0 & \\ 1 & 1 & 7 & \\ & = & 1 \times x^2 + 1 \times x^1 + 7 \times x^0 \\ & = & x^2 + x + 7 \end{array}$$

$$\text{Given } (117)_x = (79)_{10}$$

$$\therefore x^2 + x + 7 = 79$$

$$x^2 + x + 7 - 79 = 0$$

$$x^2 + x - 72 = 0$$

$$x^2 + 9x - 8x - 72 = 0$$

$$x(x+9) - 8(x+9) = 0$$

$$(x+9)(x-8) = 0$$

$$x = -9, x = 8$$

Radix cannot be negative

$$\therefore x = 8$$

Similarly, from the given problem $(42)_y = (79)_{10}$

$$1 \cdot y^2 + 4 \cdot y^1 + 2 \cdot y^0 = 79$$

$$\Rightarrow y^2 + 4y - 79 = 0$$

$$y^2 + 11y - 7y - 79 = 0$$

$$y(y+11) - 7(y+11) = 0$$

$$(y-7)(y+11) = 0$$

$$y = 7 ; y = -11$$

Radix cannot be negative. So $y = 7$.

$$\therefore x = 8, y = 7$$

Example: $(143)_5 = (x)_6$ Then $x = ?$

Sol)

First of all convert $(143)_5$ to decimal.

$$(143)_5 = 1 \times 5^2 + 4 \times 5^1 + 3 \times 5^0$$

$$\Rightarrow (143)_5 = (48)_{10}$$

Now $(48)_{10} = (x)_6$, to get x value perform successive division with 6.

$$\begin{array}{r} 6 \overline{) 48} \\ \underline{6 \times 8 = 48} \\ 0 \end{array} \quad \begin{array}{c} \uparrow \\ 1-2 \end{array}$$

$$\therefore (48)_{10} = (120)_6, \quad x = 120.$$

Complements:

In digital computers, to simplify the subtraction operation and for logical operations complements are used.

In each radix system, there are two types of complements. They are
1) Radix Complement 2) Diminished radix Complement.

The radix complement is referred as n 's Complement and the diminished radix complement is referred as $n-1$'s Complement.

For example in binary system the radix is 2. So the two possible complements in the binary number system are

2's Complement and 1's Complement. In Decimal number system we have 10's Complement and 9's Complement.

In octal we have the 8's Complement and 7's Complement. In hexadecimal we have 16's Complement and 15's Complement.

1's Complement representation :

The 1's complement of a given binary number is obtained by replacing all 1's with 0's and all 0's with 1's.

(or)

The 1's Complement of a binary number is obtained by subtracting each bit of the given binary number from 1.

2's Complement representation :

The 2's Complement of a given binary number is obtained by adding 1 to the LSB of the 1's Complement of that number.

Example :

Find the 1's Complement and 2's Complement of the following binary numbers. i) 10110110 ii) 11001011

Sol) i) Given binary number = $(10110110)_2$

$$1's \text{ Complement} = 01001001$$

$$2's \text{ Complement} = \begin{array}{r} 01001001 \\ 1 \\ \hline \end{array}$$

$$\therefore 2's \text{ Complement} = \underline{\underline{01001010}}$$

ii) Given binary number = $(11001011)_2$

$$1's \text{ Complement} = (00110100)$$

$$2's \text{ Complement} = \begin{array}{r} 00110100 \\ 1 \\ \hline \end{array}$$

$$\therefore 2's \text{ Complement} = \underline{\underline{00110101}}$$

Binary Arithmetic:

The digital computers will perform various arithmetic operations and logical operations. The basic arithmetic operations in the binary are 1. Binary Addition 2. Binary subtraction 3. Binary Multiplication and 4. Binary Division.

Binary Addition:

The binary addition involves the four elementary operations given below.

$$i) 0+0 = 0$$

$$ii) 0+1 = 1$$

$$iii) 1+0 = 1$$

$$iv) 1+1 = (10)_2$$

↑
↑
 carry sum

Example: Perform the binary addition between $(11001011)_2$ and $(11011010)_2$

Sol) Given binary numbers are $(11001011)_2$ and $(11011010)_2$

$$\begin{array}{r}
 11001011 \\
 11011010 \\
 \hline
 11111001 \\
 11011010 \\
 \hline
 11010010
 \end{array}$$

$$\therefore (11001011)_2 + (11011010)_2 = 11010010$$

Example: Add $(28)_{10}$ and $(15)_{10}$ by converting them to binary.

Sol)

$$\begin{array}{r}
 2 \overline{) 28} \\
 \underline{14-0} \\
 2 \overline{) 14} \\
 \underline{7-0} \\
 2 \overline{) 7} \\
 \underline{3-1} \\
 2 \overline{) 3} \\
 \underline{1-1}
 \end{array}$$

$$\therefore (28)_{10} = (11100)_2$$

$$\begin{array}{r}
 2 \overline{) 15} \\
 \underline{7-1} \\
 2 \overline{) 7} \\
 \underline{3-1} \\
 2 \overline{) 3} \\
 \underline{1-1}
 \end{array}$$

$$\therefore (15)_{10} = (1111)_2$$

$$\text{Now } (28)_{10} + (15)_{10} = (11100)_2 + (1111)_2$$

$$\begin{array}{r}
 11100 \\
 1111 \\
 \hline
 11111 \\
 1111 \\
 \hline
 10111
 \end{array}$$

$$\therefore (28)_{10} \text{ and } (15)_{10} = (10111)_2 = (43)_{10}$$

Binary Subtraction:

The binary subtraction includes four basic operations.

- They are
- i) $0 - 0 = 0$
 - ii) $0 - 1 = 1$ with borrow 1
 - iii) $1 - 0 = 1$
 - iv) $1 - 1 = 0$

$$X - Y$$

$X = \text{Minuend}$, $Y = \text{Subtrahend}$

In all the four operations shown above subtrahend bit is subtracted from the minuend bit. In the second case (i.e. $0 - 1 = 1$ with borrow 1) the minuend bit is smaller than the subtrahend bit hence '1' is taken as borrow.

Ex: Perform $(11101100)_2 - (00110010)_2$

Sol: Given $(11101100)_2 - (00110010)_2$

$$\begin{array}{r} 11101100 \\ 00110010 \\ \hline 10111010 \end{array}$$

$$\therefore (11101100)_2 - (00110010)_2 = (10111010)_2$$

NOTE: It is easy to perform the subtraction manually by a human being. But it is difficult to implement in a computer. So the computer uses complement methods to perform the subtraction operation.

Binary subtraction using 1's Complement method:

The operation $A - B$ is performed using the following steps.

1. Find the 1's Complement of B.
2. Add 1's complement of B with A.
3. If carry is generated the result is positive and in the true form. Add carry to the result to get the final result.
4. If carry is not generated the result is negative and in the 1's complement form. Find the 1's complement of the result to get the final result.

Ex: Perform $(28)_{10} - (15)_{10}$ using 1's complement method.

Sol) Given numbers are $(28)_{10}$ and $(15)_{10}$

$$(28)_{10} = (11100)_2$$

$$(15)_{10} = (1111)_2$$

The number of bits in the binary equivalent of $(28)_{10}$ is five where as in $(15)_{10}$ it is four. So append a 0 in the left side of the binary equivalent of $(15)_{10}$ to make it 5 bits.

$$\text{i.e. } (15)_{10} = (01111)_2$$

Now 1's complement of $(15)_{10} = 10000$

Adding 1's complement of $(15)_{10}$ with $(28)_{10}$ using binary addition.

$$\begin{array}{r} 11100 \\ + 10000 \\ \hline 1 \text{ carry} \quad 01100 \end{array}$$

Since carry is generated, the result is positive and in the true form. Now add carry to the result 01100 to get the final result.

$$\begin{array}{r} 01100 \\ + 1 \\ \hline 01101 \end{array}$$

$$01101 = (13)_{10}$$

$$\therefore (28)_{10} - (15)_{10} = (01101)_2 = (13)_{10} \text{ Ans}$$

Ex: Perform $(15)_{10} - (28)_{10}$ using 1's complement method.

Sol) Given numbers are $(15)_{10}$ and $(28)_{10}$

$$(15)_{10} = (1111)_2$$

$$(28)_{10} = (11100)_2$$

To make the number of bits equal in both the numbers, append a zero (0) in the left side of 15.

$$(15)_{10} = (01111)_2$$

Now 1's Complement of $(28)_{10} = (11100)_2$ is $(00011)_2$

Add the 1's Complement of $(28)_{10}$ with $(15)_{10}$ using binary addition

$$\begin{array}{r} 01111 \\ 00011 \\ \hline 10010 \end{array}$$

Here no carry is generated. Hence the result is negative and in 1's Complement form.

1's Complement of 10010 is $= 01101 = (13)_{10}$

$$\therefore (15)_{10} - (28)_{10} = -(01101)_2 = -(13)_{10}$$

Ex: Perform $(31)_{10} - (19)_{10}$ using 6 bit 1's complement method.

Sol: Given numbers are $(31)_{10}$ and $(19)_{10}$

$$(31)_{10} = (11111)_2, (19)_{10} = (10011)_2$$

Given 6 bit 1's Complement method. So the binary equivalent of $(31)_{10}$ and $(19)_{10}$ should have 6 bits each. So add one zero to the left side of $(31)_{10}$ and $(19)_{10}$ to make them 6 bit.

$$\therefore (31)_{10} = (011111)_2, (19)_{10} = (010011)_2$$

Now 1's Complement of $(19)_{10} = 101100$

Add 1's Complement of $(19)_{10}$ with $(31)_{10}$ using binary addition

$$\begin{array}{r} 011111 \\ 101100 \\ \hline 001011 \end{array}$$

①
Carry

Since carry is generated result is positive and in the true form, add carry to the result 001011 to get final result.

$$\begin{array}{r} 001011 \\ 111 \\ \hline 001100 \end{array}$$

$$(001100)_2 = (12)_{10}$$

$$\therefore (31)_{10} - (19)_{10} = +(001100)_2 = +12_{10}$$

Binary subtraction using 2's Complement method:

The operation $A-B$ using 2's complement method is performed by using the following steps.

- 1) Find the 2's Complement of B .
- 2) Add 2's Complement of B with A .
- 3) If the carry is generated the result is positive and in the true form. Ignore the carry to get the final result.
- 4) If the carry is generated the result is negative and in the 2's Complement form. Find the 2's Complement of the result to get the final result.

Ex ① Find $(37)_{10} - (25)_{10}$ using 2's Complement method.

Sol) Given numbers are $(37)_{10} = (100101)_2$
 $(25)_{10} = (11001)_2$

The number of bits in the binary of $(25)_{10}$ is one bit less than $(37)_{10}$. So append a zero to the left of $(25)_{10}$.

$$\therefore (25)_{10} = (011001)_2$$

$$2's \text{ Complement of } (25)_{10} = ?$$

$$1's \text{ Complement of } (25)_{10} = 100110$$

$$2's \text{ Complement of } (25)_{10} = \begin{array}{r} 100110 \\ + 1 \\ \hline 100111 \end{array}$$

Adding 2's Complement of $(25)_{10}$ and $(37)_{10}$ using binary addition

$$\begin{array}{r} 100101 \\ 100111 \\ \hline 1 \\ 001100 \end{array}$$

Carry

Since a carry is generated the result is positive and in true form. Discard the carry to get the final result.

$$\therefore (37)_{10} - (25)_{10} = + (001100)_2 = + (12)_{10}$$

Ex 2 : Find $(25)_{10} - (37)_{10}$ using 2's Complement method.

Sol) Given numbers are $(25)_{10}$ and $(37)_{10}$

$$(25)_{10} = (11001)_2$$

$$(37)_{10} = (100101)_2$$

To equate the number of bits in each equivalent value, append a zero to the left of $(25)_{10}$.

$$\therefore (25)_{10} = (011001)_2$$

2's Complement of $(37)_{10} = ?$

$$1's \text{ Complement of } (37)_{10} = 011010$$

$$2's \text{ Complement of } (37)_{10} = 011011$$

Add 2's Complement of $(37)_{10}$ with $(25)_{10}$ using binary addition

$$\begin{array}{r} 011001 \\ 011011 \\ \hline 110100 \end{array}$$

Since carry is not generated the result is negative and in 2's Complement form.

$$2's \text{ Complement of the result} = \begin{array}{r} 001011 \\ 111 \\ \hline 001100 \end{array}$$

$$\therefore (25)_{10} - (37)_{10} = (-001100)_2 = (-12)_{10}$$

Practice problems :

1) Find the following using 1's Complement method

i) $(69)_{10} - (35)_{10}$

ii) $(43)_{10} - (78)_{10}$

2) Find the following using 2's Complement method.

i) $(55)_{10} - (28)_{10}$

ii) $(39)_{10} - (48)_{10}$

Binary Multiplication:

The multiplication process for binary numbers is similar to the decimal numbers. Actually binary multiplication is simple than decimal multiplication, since it involves only 1's and 0's.

This process involves the following four elementary operations.

$$0 \times 0 = 0$$

$$0 \times 1 = 0$$

$$1 \times 0 = 0$$

$$1 \times 1 = 1$$

Ex:

multiply $(1101)_2$ by $(101)_2$ using binary multiplication.

Sol)

Here we need to find $(1101)_2 \times (101)_2$

Here $(1101)_2$ is multiplicand and $(101)_2$ is multiplier.

$$\begin{array}{r} 1101 \times 101 \\ \hline 1101 \\ 0000 \\ 1101 \\ \hline 1000001 \end{array}$$

$$\therefore (1101)_2 \times (101)_2 = (1000001)_2 = (65)_{10}$$

Ex:

multiply $(101.11)_2$ and $(110.01)_2$ using binary multiplication.

Sol).

$$\begin{array}{r} 101.11 \times 110.01 \\ \hline 10111 \\ 00000 \\ 00000 \\ 10111 \\ 10111 \\ \hline 100011.1111 \end{array}$$

Fractional digits in the multiplication result = Fractional digits in the multiplicand + Fractional digits in the multiplier.

$$= 2 + 2$$

$$= 4$$

$$\therefore (101.11)_2 \times (110.01)_2 = (100011.1111)_2 = (35.9375)_{10}$$

Practice Problems: Find i) $(1110)_2 \times (101)_2$ ii) $(111.001)_2 \times (101.011)_2$ using binary multiplication.

Binary Division:

The division process for binary numbers is similar to the decimal numbers. In binary division, division by 0 has no meaning.

The two elementary operations of binary division are

$$0 \div 1 = 0$$

$$1 \div 1 = 1$$

Ex: Divide $(11011011)_2$ by $(110)_2$

Sol)

$$\begin{array}{r} 110 \overline{) 11011011} \quad (100100 \\ \underline{110} \\ 110 \\ \underline{110} \\ 11 \end{array}$$

$$\text{Quotient} = (100100)_2$$

$$\text{Remainder} = (11)_2$$

Practice Problems: Find i) $(10101101)_2 \div (100)_2$ ii) $(110101011)_2 \div (111)_2$

Signed Binary Numbers:

In practice, we use plus (+) sign to represent a positive number and minus (-) sign to represent a negative number. But in Computers due to some limitations, both positive and negative numbers are represented with only binary digits.

In general any binary number may belong to any one of these two categories given below.

1. Unsigned binary numbers
2. Signed binary numbers.

An unsigned binary number always represent positive number. where as in signed binary number, the number can be either positive

or negative. The most bit in the signed binary number represents the sign of the number. If the sign bit is 0 it represents the number as positive and if the sign bit is 1 it represents the number to be negative.

* The signed binary numbers are represented in a format called signed magnitude form.

* In a signed binary number the left most bit (i.e. MSB) represents the sign of the number and all the remaining bits represent the magnitude of the number. Some of the 8 bit signed binary numbers are given below.

$$\begin{aligned} +6 &= 00000110 \\ -14 &= 10001110 \\ +28 &= 00011100 \\ -64 &= 11000000 \end{aligned}$$

* In unsigned binary number all the bits represent the magnitude.

* If the signed binary number is negative, it can be represented in three ways

- 1) signed magnitude form
- 2) signed 1's Complement form
- 3) signed 2's Complement form.

* If the signed binary number is positive then the signed magnitude form, signed 1's Complement form and signed 2's Complement form all are identical.

Representation of signed binary numbers using 2's Complement and 1's Complement:

▷ If the signed binary number is positive, the sign bit '0' is placed. For such numbers 1's Complement and 2's Complement are equal to signed magnitude form.

2) If the signed binary number is negative, then the magnitude is represented in its complement (or) 2's complement form and then the sign bit 1 is placed in front of MSB.

Ex: Express $+51$ & -51 in signed magnitude format, 1's complement and 2's Complement format.

Sol) Given numbers are $+51$ and -51 .

magnitude of $+51 = (110011)_2$

$$\begin{array}{r} 2 \overline{) 51} \\ \underline{25} \\ 12 \\ \underline{6} \\ 2 \\ \underline{1} \\ 1 \end{array}$$

i) signed magnitude form of $+51$ is obtained by placing '0' in the place of sign bit to 51.

i.e signed magnitude form of $+51 = 0110011$

signed 1's Complement form and signed 2's Complement form are identical to signed magnitude form for positive numbers.

\therefore signed 1's Complement form of $+51 = 0110011$

signed 2's Complement form of $+51 = 0110011$.

ii) signed magnitude form of -51 is obtained by placing '1' in the place of sign bit to 51.

i.e signed magnitude form of $-51 = 1110011$

signed 1's Complement form of $-51 = 1001100$.

signed 2's Complement form of $-51 = \begin{array}{r} 1001100 \\ \underline{1001101} \end{array}$

Practice problems: find the 1's complement, 2's complement and signed magnitude form for the following numbers

i) $+75$, ii) -69 iii) -92

(18)

The following table shows all possible 4-bit signed binary numbers in the signed magnitude form, signed 1's complement and signed 2's complement form.

Decimal	signed 2's complement	Signed 1's complement	Signed magnitude
+7	0111	0111	0111
+6	0110	0110	0110
+5	0101	0101	0101
+4	0100	0100	0100
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
-0	-	1111	1000
-1	1111	1110	1001
-2	1110	1101	1010
-3	1101	1100	1011
-4	1100	1011	1100
-5	1011	1010	1101
-6	1010	1001	1110
-7	1001	1000	1111
-8	1000	-	-

7's Complement and 8's Complement:

In the octal number system the complements are 7's complement and 8's complement.

The 7's Complement of an octal number is obtained by subtracting each octal digit from 7.

The 8's Complement of an octal number is obtained by adding 1 to the least significant digit of 7's Complement.

example: Find the 7's, 8's Complement of $(243)_8$ and $(1024)_8$.

sol To find 7's Complement of $(243)_8$, Subtracting each digit from 7 we get

$$\begin{array}{r} 777 \\ 243 \\ \hline 534 \end{array}$$

$$\therefore 7's \text{ Complement of } (243)_8 = (534)$$

$$8's \text{ Complement of } (243)_8 = 7's \text{ Complement of } (243)_8 + 1$$

$$= \begin{array}{r} 534 \\ 1 \\ \hline 535 \end{array}$$

$$= 535$$

To find 7's Complement of $(1024)_8$, we have to subtract each digit of $(1024)_8$ from 7.

$$\begin{array}{r} 7777 \\ 1024 \\ \hline 6753 \end{array}$$

$$\therefore 7's \text{ Complement of } (1024)_8 = 6753$$

$$8's \text{ Complement of } (1024)_8 = \begin{array}{r} 6753 \\ 1 \\ \hline 6754 \end{array}$$

example Find the 8's Complement of $(365)_8$

sol) To get the 8's Complement firstly we need to get

7's Complement of $(365)_8$.

$$7's \text{ Complement of } (365)_8 = \begin{array}{r} 777 \\ 365 \\ \hline 412 \end{array}$$

$$8's \text{ Complement of } (365)_8 = \begin{array}{r} 412 \\ 1 \\ \hline 413 \end{array} = 413$$

9's and 10's Complements:

The decimal number system has 9's and 10's Complements.

To get the 9's Complement for a decimal number, subtract each digit of the given decimal number from 9.

To get the 10's Complement for a decimal number, add 1 to the least significant digit of the 9's Complement of that number.

15's and 16's Complements:

In hexadecimal number system the Complements are 15's and 16's Complements.

To get the 15's Complement for a given hexadecimal number, subtract each digit of the given hexadecimal number from 15.

To get the 16's Complement for a given hexadecimal number, add 1 to the least significant digit of the 15's Complement of that number.

Ex: Find the i) 9's and 10's complement for $(786)_{10}$
ii) 15's and 16's Complement for $(9BE8)_{16}$

Sol) i) To get the 9's Complement for $(786)_{10}$ subtract each digit of $(786)_{10}$ from 9.

$$\begin{array}{r} 999 \\ 786 \\ \hline 213 \end{array}$$

\therefore 9's Complement = 213
10's Complement = $213 + 1 = 214$.

ii) To get the 15's Complement for $(9BE8)_{16}$ subtract each digit of $(9BE8)_{16}$ from 15.

$$\begin{array}{r} 15 \ 15 \ 15 \ 15 \\ 9 \ B \ E \ 8 \\ \hline 6 \ 4 \ 1 \ 7 \end{array}$$

$$\therefore 15's \text{ complement of } (9BE8)_{16} = 6417$$

$$16's \text{ Complement of } (9BE8)_{16} = \frac{6417}{6418}$$

practice problems: Find

- i) 7's and 8's Complement of $(765)_8$
- ii) 9's and 10's Complement of $(842)_{10}$
- iii) 15's and 16's Complement of $(87ED)_{16}$

subtraction using 2's complement: Minuend - Subtrahend (M-S)

- 1) Equating the number of digits by padding appropriate number of zeros in front of the numbers.
- 2) Find the 2's Complement to subtrahend and add with minuend
- 3) If Carry is generated the result is treated as positive and in true form. Discard the Carry to get the final result.
- 4) If the Carry is not generated the result is treated as negative and the result is present in 2's Complement form. Find the 2's Complement of the result and place a minus (-) sign in front of it to get the final result.

Ex: Find $(3265)_8 - (741)_8$ using 8's Complement subtraction.

$$\text{Minuend} = (3265)_8, \quad \text{Subtrahend} = (741)_8$$

Sol
To equate the number of digits in both Minuend and Subtrahend add a leading zero to $(741)_8$ i.e. $(0741)_8$

Now find the 8's Complement to Subtrahend i.e. 0741 .

For that we need 7's Complement of $(0741)_8$.

$$\therefore 7's \text{ Complement of } (0741)_8 = \frac{7777}{0741}$$

$$8's \text{ Complement of } (0741)_8 = \frac{7036}{7037}$$

Adding 8's Complement of $(0741)_8$ to $(3265)_8$.

$$\begin{array}{r} 3265 \\ 7037 \\ \hline 111 \\ \hline 2324 \end{array}$$

Since carry is generated the result is positive. Discard carry to get the final result.

$$\therefore (3265)_8 - (741)_8 = (2324)_8$$

Example: Find the result of $(9876)_{10} - (345)_{10}$ using 10's Complement subtraction method.

Sol To equate the number of digits in each of the numbers, add a zero to $(345)_{10}$ in leading position.

Now Minuend = $(9876)_{10}$, subtrahend = $(0345)_{10}$.

Finding 10's Complement of $(0345)_{10}$, For that we need 9's complement of $(0345)_{10}$.

$$\begin{array}{r} 9's \text{ Complement of } (0345)_{10} = 9999 \\ 0345 \\ \hline 9654 \end{array}$$

$$10's \text{ Complement} = \begin{array}{r} 9654 \\ \hline 9655 \end{array}$$

Adding 10's Complement of $(0345)_{10}$ to $(9876)_{10}$

$$\begin{array}{r} 9876 \\ 9655 \\ \hline 19531 \\ \hline 9531 \end{array}$$

Since carry is generated the result is positive. Discard carry to get the final result. i.e 9531.

$$\text{i.e } (9876)_{10} - (345)_{10} = (9531)_{10}$$

Example: Find $(327)_8 - (765)_8$ using 8's Complement subtraction

Sol) Minuend = $(327)_8$ Subtrahend = $(765)_8$.

The number of digits are equal in both minuend and subtrahend.

So we need of adding leading zeros to any one of them.

Finding 8's Complement of $(765)_8$, for that firstly we need 7's Complement of $(765)_8$.

$$7's \text{ Complement of } (765)_8 = \begin{array}{r} 777 \\ 765 \\ \hline 012 \end{array}$$

$$8's \text{ Complement of } (765)_8 = \begin{array}{r} 012 \\ 1 \\ \hline 013 \end{array}$$

Adding 8's Complement of $(765)_8$ to $(327)_8$

$$\begin{array}{r} 327 \\ 013 \\ \hline 342 \end{array}$$

Since Carry is not generated the result is negative and in 8's Complement form.

Finding 8's Complement of the result and placing a minus sign before that gives the final result.

$$\therefore 7's \text{ Complement of } (342)_8 = \begin{array}{r} 777 \\ 342 \\ \hline 435 \end{array}$$

$$8's \text{ Complement of } (342)_8 = \begin{array}{r} 435 \\ 1 \\ \hline 436 \end{array}$$

$$\therefore \text{final result} = (-436)_8.$$

practice problem: Find $(497)_{10} - (2563)_{10}$ using 10's Complement method.

(21)

Example: Find $(7D9E)_{16} - (9CB)_{16}$ using 16's Complement Subtraction.

Sol:

$$\text{Minuend} = (7D9E)_{16}, \text{ Subtrahend} = (9CB)_{16}$$

To equate the number of digits in minuend and subtrahend add a zero to subtrahend in its leading position.

$$\therefore \text{Subtrahend} = (09CB)_{16}$$

Find 16's Complement of subtrahend $(09CB)_{16}$.

$$15's \text{ Complement of } (09CB)_{16} = \begin{array}{r} 15 \ 15 \ 15 \ 15 \\ 0 \ 9 \ C \ B \\ \hline F \ 6 \ 3 \ 4 \end{array}$$

$$16's \text{ Complement of } (09CB)_{16} = \begin{array}{r} F634 \\ \hline 1 \\ \hline F635 \end{array}$$

Adding 16's Complement of $(09CB)_{16}$ to $(7D9E)_{16}$

$$\begin{array}{r} 7D9E \\ \textcircled{1} F635 \\ \hline 73D3 \end{array}$$

Since Carry is generated the result is positive. Discard the carry to get the final result.

Practice problem: Find $(3DB)_{16} - (8EF7)_{16}$ using 16's complement subtraction.

Subtraction using $n-1$'s Complement: Minuend (M) - Subtrahend (S)

- 1) Equating the number of digits in minuend and subtrahend by padding appropriate number of leading zeros.
- 2) Find the $n-1$'s Complement to subtrahend and add with minuend.
- 3) If carry is generated the result is treated as positive and in true form. Add the carry to the least significant ^{digit} position of the result to get the final result.
- 4) If carry is not generated the result is treated as negative.

and in $n-1$'s Complement form. Find the $n-1$'s Complement of the result and place minus (-) sign to get the final result.

Example Find $(763)_8 - (245)_8$ using 7's Complement method.

Sol) Minuend = $(763)_8$ Subtrahend = $(245)_8$.

$$7's \text{ Complement of Subtrahend} = \begin{array}{r} 777 \\ 245 \\ \hline 532 \end{array}$$

Adding 7's Complement of Subtrahend with minuend.

$$\begin{array}{r} 763 \\ 0532 \\ \hline 515 \end{array}$$

Since Carry is generated the result is positive. Add carry to the least significant digit of the result to get the final result.

$$\begin{array}{r} 515 \\ \hline 516 \end{array}$$

$$\therefore (763)_8 - (245)_8 = (516)_8$$

Example: Find $(354)_8 - (672)_8$ using 7's Complement method.

Sol) Minuend = $(354)_8$ Subtrahend = $(672)_8$.

$$7's \text{ Complement of Subtrahend} = \begin{array}{r} 777 \\ 672 \\ \hline 105 \end{array}$$

Adding 7's Complement of Subtrahend with minuend

$$\begin{array}{r} 354 \\ 105 \\ \hline 461 \end{array}$$

Since no carry is generated, the result is negative and in 7's Complement form. Find 7's Complement of the result and place minus sign to get final result.

$$\begin{array}{r} 777 \\ 461 \\ \hline 316 \end{array}$$

$$\therefore (354)_8 - (672)_8 = -(316)_8$$

Practice problems: Find i) $(476)_8 - (73)_8$ ii) $(64)_8 - (75)_8$ using 7's Complement method.

Example: Find $(978)_{10} - (98)_{10}$ using 9's Complement method.

Sol) Minuend = $(978)_{10}$, Subtrahend = $(98)_{10}$

To equate the number of zeros in both the given numbers add a leading zero to the Subtrahend.

$$\text{Subtrahend} = (098)_{10}$$

$$\begin{array}{r} 999 \\ 098 \\ \hline 901 \end{array}$$

9's Complement of Subtrahend =

Adding $(978)_{10}$ with 9's Complement of Subtrahend.

$$\begin{array}{r} 978 \\ 0901 \\ \hline 879 \end{array}$$

Since Carry is generated the result is positive and in true form. Add Carry to the least significant digit position of the result to get the final result.

$$\begin{array}{r} 879 \\ 1 \\ \hline 880 \end{array}$$

$$\therefore (978)_{10} - (98)_{10} = (880)_{10}$$

Example: Find $(98D)_{16} - (7BE)_{16}$ using 15's Complement method.

Sol) Given minuend = $(98D)_{16}$, Subtrahend = $(7BE)_{16}$.

$$\begin{array}{r} 15 \quad 15 \quad 15 \\ 7 \quad B \quad E \\ \hline 8 \quad 4 \quad 1 \end{array}$$

15's Complement of Subtrahend =

Adding $(98D)_{16}$ with 15's Complement of $(7BE)_{16}$

$$\begin{array}{r} 98D \\ + 841 \\ \hline 1CE \end{array}$$

Since the carry is generated the result is positive. Add the carry to the least significant digit of the result to get the final result.

$$\begin{array}{r} 1CE \\ + 1 \\ \hline 1CF \end{array}$$

$$\therefore (98D)_{16} - (7BE)_{16} = (1CF)_{16}$$

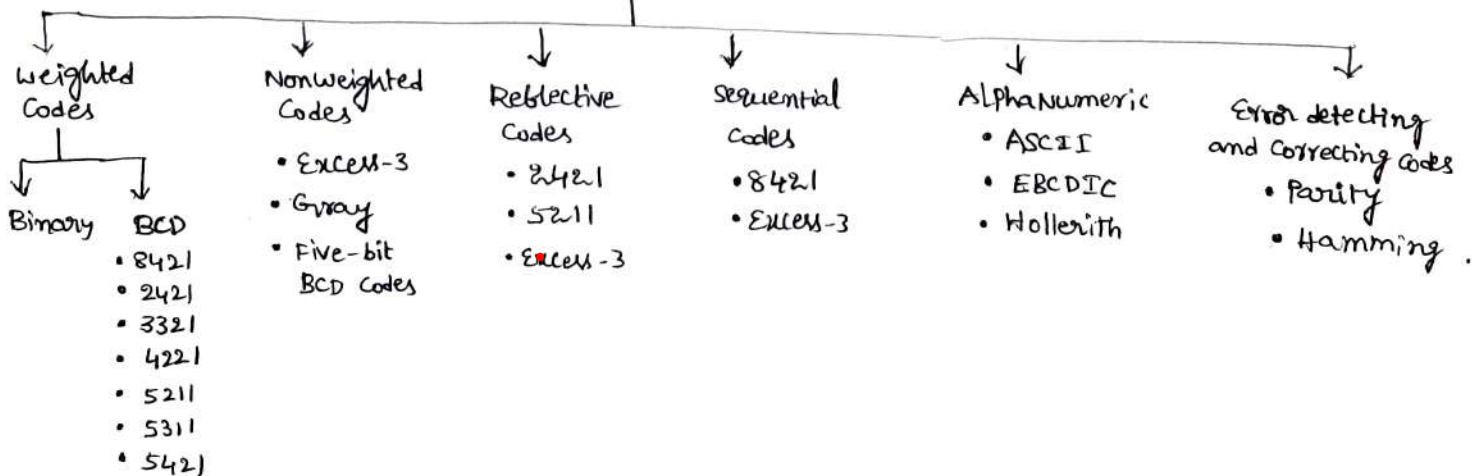
practice problems: Find i) $(179)_{10} - (96)_{10}$ and $(69)_{10} - (187)_{10}$ using 9's Complement method. ii) $(1BD)_{16} - (FC)_{16}$ and $(9D)_{16} - (179)_{16}$ using 15's Complement method.

Binary Codes:

The digital data is represented, stored and transmitted as groups of binary digits. The group of binary digits is known as binary code, that represents numbers, letters of the alphabets, special characters and special functions (or) control functions.

The binary codes are broadly classified as 1) weighted codes 2) Non weighted codes 3) Reflective codes 4) sequential codes 5) Alphabetic codes 6) Error detecting and correcting codes.

Binary Codes



Weighted Codes:

In weighted codes, each digit position of the number represents a specific weight. In weighted binary code each bit has a specific weight and each decimal digit is represented by a group of four bits.

BCD (Binary Coded Decimal) Codes:

BCD is a numeric code in which each digit of a decimal number is represented by a separate group of 4 bits. The most commonly used BCD code is 8421 BCD, in which each decimal digit is represented by a 4 bit binary number. It is called 8421 BCD because the weights associated with 4-bits are 8, 4, 2 and 1 from left to right respectively.

Decimal digit	8421 BCD
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

In multidigit decimal number, each decimal digit is individually coded with 8421 BCD code.

$$\text{Ex: } (58)_{10} = (0101 \ 1000)_{8421}$$

Therefore total 8-bits are required to encode $(58)_{10}$ in 8421 BCD. When we represent the same decimal number in binary: $(111010)_2$, we require only 6 digits. This means that, representation of a decimal number in 8421 BCD is less efficient than normal binary number system. The advantage of a BCD code is that it is easy to convert any decimal number into BCD.

BCD addition:

Procedure for BCD addition is given below.

- 1) Add two BCD numbers using ordinary binary addition.
- 2) If 4-bit sum is equal to (or) less than 9, no correction is needed. The sum is in proper BCD form.
- 3) If the 4-bit sum is greater than 9 (or) if a carry is generated from the 4-bit sum, the sum is invalid. To correct the invalid sum, add 6 i.e. $(0110)_2$ to the 4-bit sum. If carry results from this addition, add it to the next higher order BCD digit.

Ex: Perform each of the following decimal additions in 8421 BCD

a) $24 + 18$ b) $48 + 59$

Sol) a) Given decimal numbers $(24)_{10}$ and $(18)_{10}$

$$\text{BCD eq' of } (24)_{10} = 0010 \ 0100$$

$$\text{BCD eq' of } (18)_{10} = \begin{array}{r} 0001 \ 1000 \\ \hline 0011 \ 1100 \end{array}$$

The lower nibble of the result is 1100 which is greater than 9. So add 0110 to the lower nibble to make 1100 a valid BCD.

$$\begin{array}{r} 0011 \ 1100 \\ 0001 \ 0110 \\ \hline 0100 \ 0010 \end{array} = (42)_{10}$$

$$\therefore \text{The result of } (24)_{10} + (18)_{10} = (42)_{10}$$

b) Sol) Given decimal numbers $(48)_{10}$, $(59)_{10}$

$$\text{BCD equivalent of } (48)_{10} = 0100 \ 1000$$

$$\text{BCD equivalent of } (59)_{10} = \begin{array}{r} 0101 \ 1001 \\ \hline 1010 \ 0001 \end{array}$$

Since a carry is generated from lower nibble to higher nibble add 0110 to the lower nibble of the result. The higher nibble $(1010)_2$ is 10 which is greater than 9, hence add $(0110)_2$ to the higher

- nibble of the result.

$$\begin{array}{r}
 1010 \ 0001 \\
 0110 \ 0110 \\
 \hline
 0001 \ 0000 \ 0111 \\
 \hline
 1 \quad 0 \quad 7
 \end{array}$$

$$\therefore (48)_{10} + (59)_{10} = (107)_{10}$$

Ex: perform the following using 8421 BCD addition.

a) $(23)_{10} + (78)_{10}$ b) $(77)_{10} + (26)_{10}$

BCD subtraction using 9's complement method: (procedure)

- 1) Find the 9's complement of subtrahend number and add it to the minuend using BCD addition.
- 2) If carry is not generated, the result is negative and it is in 9's complement form. Find the 9's complement of the result to get the final result.
- 3) If carry is generated, the result is positive and add carry to the LSB to get the final result.

Example: perform $(46)_{10} - (37)_{10}$ using BCD in 9's complement method.

sol) Minuend = $(46)_{10}$, Subtrahend = $(37)_{10}$

$$9's \text{ complement of subtrahend} = \begin{array}{r} 99 \\ 37 \\ \hline 62 \end{array}$$

Adding minuend with 9's complement of subtrahend using BCD-addition

$$\begin{array}{r}
 46 = 0100 \ 0110 \\
 62 = 0110 \ 0010 \\
 \hline
 1010 \ 1000
 \end{array}$$

Since 1010 is > 9 , add 0110.

$$\begin{array}{r}
 \text{Carry} \rightarrow \textcircled{1} \quad 0110 \\
 \quad \quad \quad 11 \\
 \hline
 0000 \ 1000 \\
 \quad \quad \quad 1
 \end{array}$$

Since carry is generated result is positive. Add this carry to LSB to get final result.

$$0000 \ 1001 = 09$$

$$\therefore (46)_{10} - (37)_{10} = (09)_{10}$$

Example: perform $(37)_{10} - (52)_{10}$ using 9's Complement in BCD.

sol) Given minuend = $(37)_{10}$, Subtrahend = $(52)_{10}$.

$$9's \text{ Complement of Subtrahend} = \begin{array}{r} 99 \\ 52 \\ \hline 47 \end{array}$$

Adding 9's Complement of subtrahend with minuend using BCD - addition.

$$(37)_{10} = 0011 \ 0111$$

$$(47)_{10} = \begin{array}{r} 0100 \ 0111 \\ \hline 0111 \ 1110 \end{array}$$

$$\begin{array}{r} 1111 \ 0110 \\ \hline 1000 \ 0100 \end{array}$$

1110 is 79 so
add 0110

$$= (84)_{10}$$

Since no carry is generated, the result is negative and in

9's Complement form.

$$9's \text{ Complement of the result} = \begin{array}{r} 99 \\ 84 \\ \hline 15 \end{array}$$

$$\therefore (37)_{10} - (52)_{10} = (-15)_{10}$$

Practice problems: Find i) $(29)_{10} - (12)_{10}$ ii) $(67)_{10} - (98)_{10}$ using 9's

Complement in BCD.

BCD Subtraction using 10's Complement : Procedure

- 1) Find the 10's Complement of the subtrahend number and add it to the minuend using BCD addition.
- 2) If carry is generated, the result is positive and in true form discard the carry to get the final result.
- 3) If the carry is not generated, the result is negative and in 10's Complement form. Find the 10's Complement of the result to get the final result.

Example: Find $(49)_{10} - (23)_{10}$ using 10's complement in BCD.

Sol) Given minuend = $(49)_{10}$, subtrahend = $(23)_{10}$

10's Complement of subtrahend = ?

$$9's \text{ Complement of subtrahend} = \begin{array}{r} 99 \\ 23 \\ \hline 76 \end{array}$$

$$10's \text{ Complement of subtrahend} = \begin{array}{r} 76 \\ 1 \\ \hline 77 \end{array}$$

Adding minuend with 10's Complement of subtrahend using BCD addition.

$$(49)_{10} = 0100 \ 1001$$

$$(77)_{10} = \begin{array}{r} 0111 \ 0111 \\ \hline 1111 \ 1111 \\ 1100 \ 0000 \end{array}$$

Since 1100 is 79 so
add 0110 to higher
nibble

$$\begin{array}{r} 0110 \ 0110 \\ \hline \textcircled{1} \ 0010 \ 0110 \\ \downarrow \end{array}$$

Carry

Since Carry is generated
from lower nibble to higher
add 0110 to lower nibble

Since Carry is generated the result is positive and in true form. Discard Carry to get the final result.

$$\therefore (49)_{10} - (23)_{10} = (0010 \ 0110)_{BCD} = (26)_{10}$$

Example: Find $(23)_{10} - (59)_{10}$ using 10's complement in BCD.

Sol) Given minuend = $(23)_{10}$, subtrahend = $(59)_{10}$

$$9's \text{ Complement of subtrahend} = \begin{array}{r} 99 \\ 59 \\ \hline 40 \end{array}$$

10's complement = 41

Adding $(23)_{10}$ with 10's Complement of $(59)_{10}$ using BCD.

$$(23)_{10} = 0010 \ 0011$$

$$(41)_{10} = \begin{array}{r} 0100 \ 0001 \\ \hline 0110 \ 0100 \end{array}$$

$$= (64)_{10}$$

Since no carry is generated the result is negative and in 10's Complement form. So 10's Complement of the result gives the final result.

$$10's \text{ Complement of } (64)_{10} = 9's \text{ Complement of } (64)_{10} + 1$$

$$9's \text{ Complement of } (64)_{10} = \begin{array}{r} 99 \\ 64 \\ \hline 35 \end{array}$$

$$10's \text{ Complement of } (64)_{10} = \begin{array}{r} 35 \\ 1 \\ \hline 36 \end{array}$$

$$\therefore (23)_{10} - (59)_{10} = (-36)_{10} \quad \underline{\quad}$$

Practice problems: Find i) $(57)_{10} - (29)_{10}$ ii) $(37)_{10} - (92)_{10}$ using 10's complement in BCD.

Excess-3 Code (XS-3 Code):

Excess-3 code is a modified form of a BCD number. The excess-3 Code can be derived from the BCD code by adding 3 to each coded number. It is a non-weighted code, self complementing and reflected Code.

Following table shows excess-3 codes of each decimal digit. It is a sequential code because we can get any codeword simply by adding binary '1' to its previous code word as shown in table below.

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

In excess-3 code, we get 9's Complement of a number just by complementing each bit. Due to this excess-3 Code is called as self complementing Code (or) reflected Code.

Example: Find the excess-3 code and its 9's complement for the following decimal numbers i) $(592)_{10}$ ii) $(403)_{10}$

Sol) i) Given decimal number = $(592)_{10}$

Excess-3 code for $(592)_{10} = 1000\ 1100\ 0101$

9's complement of $(592)_{10}$ is =
$$\begin{array}{r} 999 \\ 592 \\ \hline 407 \end{array}$$

Excess-3 for $(407)_{10} = 0111\ 0011\ 1010$

Complement to each other

ii) Given decimal number = $(403)_{10}$

Excess-3 code for $(403)_{10} = 0111\ 0011\ 0110$

9's complement of $(403)_{10} = \begin{array}{r} 999 \\ 403 \\ \hline 596 \end{array}$

Excess-3 for $(596)_{10} = 1000\ 1100\ 1001$

NOTE: From this example it is clear that the 9's complement of a number which is in excess-3 code, is simply obtained by complementing each bit in that excess-3 code.

Excess-3 addition:

To perform excess-3 addition we need to perform the steps given below.

- 1) Add two excess-3 numbers using binary addition.
- 2) If carry = 1 add 3 i.e. $(0011)_2$ to the 4-bit sum
- 3) If carry = 0 subtract 3 from the 4-bit sum.

Ex: perform i) $(37)_{10} + (28)_{10}$ ii) $(247.6)_{10} + (357.4)_{10}$ using excess-3 addition.

Sol) i) Given $(37)_{10}$ in excess-3 form it is = $0110\ 1010$

$(28)_{10}$ in excess-3 form = $0101\ 1011$

(+)
$$\begin{array}{r} 0110\ 1010 \\ 0101\ 1011 \\ \hline 1100\ 0101 \end{array}$$

From lower nibble to higher nibble a carry is generated so add $(0011)_2$ to the lower nibble bits. At higher nibble no carry is

generated. so subtract $(0011)_2$ from higher nibble.

$$\begin{array}{r}
 1100 \ 0101 \\
 (+) \quad \quad 0011 \\
 \hline
 1100 \ 1000 \\
 (-) \ 0011 \\
 \hline
 1001 \ 1000 \\
 \hline
 \underline{\quad 6 \quad} \quad \underline{\quad 5 \quad}
 \end{array}$$

$$\therefore (37)_{10} + (28)_{10} = (65)_{10}$$

ii) Sol)

$$(247.6)_{10} \text{ in Excess-3 form} = 0101 \ 0111 \ 1010 \cdot 1001$$

$$(359.4)_{10} \text{ in excess-3 form} = 0110 \ 1000 \ 1100 \cdot 0111$$

Adding $(0011)_2$ to the
4 bit sums (0000) (0111) (0000)
Subtracting (0011) from 1100 .

$$\begin{array}{r}
 0110 \ 1000 \ 1100 \cdot 0111 \\
 \hline
 1111 \ 1111 \quad \quad 1111 \\
 1100 \ 0000 \ 0111 \cdot 0000 \\
 (+) 0011 \ 0011 \ 0011 \\
 \hline
 1100 \ 0011 \ 1010 \cdot 0011 \\
 (-) 0011 \\
 \hline
 1001 \ 0011 \ 1010 \cdot 0011 \\
 \hline
 \underline{\quad 6 \quad} \quad \underline{\quad 0 \quad} \quad \underline{\quad 7 \quad} \cdot \underline{\quad 0 \quad} = (607)_{10}
 \end{array}$$

$$\therefore (247.6)_{10} + (359.4)_{10} = (607)_{10}$$

Practice Problem: Find $(97)_{10} + (64)_{10}$ using Excess-3 addition.

Excess-3 Subtraction using 9's Complement: (or) Excess-3 subtraction

To perform Excess-3 subtraction we need to follow the steps given below.

- 1) Find the 9's Complement of the subtrahend. To do this simply complement each bit in the excess-3 code of subtrahend.
- 2) Add the minuend with the 9's Complement of the subtrahend - using Excess-3 addition.
- 3) If carry = 1 result is positive and in true form. Add carry to the LSB of the result to get the final result in excess-3 form.
- 4) If carry = 0 result is 9's Complement form. Complement

each bit in the result to get the final result in Excess-3 form.

Example perform the following in Excess-3 code using 9's complement.

i) $(267)_{10} - (175)_{10}$ ii) $(87.6)_{10} - (87.8)_{10}$.

i) sol)

$$(175)_{10} \text{ in Excess-3} = 0100\ 1010\ 1000$$

$$9's \text{ complement of } (175)_{10} \text{ in Excess-3} = 1011\ 0101\ 0111 \quad \left. \vphantom{\begin{matrix} 1011 \\ 0101 \\ 0111 \end{matrix}} \right\} \text{ Excess-3 addition}$$

$$(267)_{10} \text{ in Excess-3} = 0101\ 1001\ 1010$$

$$\begin{array}{r} \text{Carry} \leftarrow \text{①} \\ \begin{array}{r} 0101\ 1001\ 1010 \\ 11\ 11\ 11 \\ \hline 0000\ 1111\ 0001 \\ 0011\ 0011 \\ \hline 0011\ 1111\ 0100 \end{array} \end{array}$$

adding 0011 to 0000 and 0001

subtracting 0011 from 1111

$$\begin{array}{r} 0011 \\ \hline 0011\ 1100\ 0100 \\ 1 \end{array}$$

Since carry is generated, the result is positive. Add Carry to LSB to get the final result

$$\begin{array}{ccc} 0011 & 1100 & 0101 \\ \hline 0 & 9 & 2 \end{array}$$

$$\therefore (267)_{10} - (175)_{10} = (92)_{10}$$

ii) sol

$$(87.8)_{10} \text{ in Excess-3 form} = 1011\ 1010 \cdot 1011$$

$$9's \text{ complement of } (87.8)_{10} \text{ in Excess-3} = 0100\ 0101 \cdot 0100$$

$$(57.6)_{10} \text{ in Excess-3 form} = 1000\ 1010 \cdot 1001$$

subtracting 0011 from 1100, 1111 and 1101

$$\begin{array}{r} 1000\ 1010 \cdot 1001 \\ 1100\ 1111 \cdot 1101 \\ \hline 0011\ 0011 \cdot 0011 \\ 11\ 1 \\ \hline 1001\ 1100 \cdot 1010 \end{array}$$

Since no carry is generated the result is negative and in 9's complement form. So finding the 9's complement of the result gives final result

$$9's \text{ complement of the result} = (0110\ 0011 \cdot 0101)_{\text{XS-3}} = (30.2)_{10}$$

$$\therefore (57.6)_{10} - (87.8)_{10} = (30.2)_{10}$$

Practice Problems: Perform i) $(27)_{10} - (65)_{10}$ ii) $(365)_{10} - (248)_{10}$ in Excess-3 code using 9's complement.

Excess-3 Subtraction using 10's complement:

- 1) Take 10's complement of the subtrahend and add it to the minuend using Excess-3 addition.
- 2) If carry = 1 The result is positive, ignore the carry to get the final result.
- 3) If carry = 0 The result is negative. Find the 10's complement of the result to get the final result.

Example: Find i) $(65)_{10} - (32)_{10}$ ii) $(39)_{10} - (57)_{10}$ using 10's complement Excess-3 subtraction.

i) sol) $(32)_{10}$ in Excess-3 code = 0110 0101

9's complement of $(32)_{10}$ in Excess-3 code = 1001 1010

\Rightarrow 10's complement of $(32)_{10}$ in Excess-3 = 1001 1011 } Excess-3 addition

$(65)_{10}$ in Excess-3 code

$$\begin{array}{r}
 1001 \ 1000 \\
 \underline{11} \\
 0011 \ 0011 \\
 \underline{11} \\
 0011 \ 0011 \\
 \underline{11} \\
 0110 \ 0110
 \end{array}$$

Add 0011 to 0011 and 0011

since the carry is generated the result is positive. Ignore the carry to get the final result.

$$\therefore (65)_{10} - (32)_{10} = (0110 \ 0110)_{\text{Excess-3}} = (33)_{10}$$

ii) sol)

$(57)_{10}$ in Excess-3 form = 1000 1010

9's complement of $(57)_{10}$ in Excess-3 form = 0111 0101

10's complement of $(57)_{10}$ in Excess-3 form = 0111 0110 } Excess-3 addition

$(39)_{10}$ in Excess-3 form

$$\begin{array}{r}
 0110 \ 1100 \\
 \underline{11} \\
 1110 \ 0010 \\
 \underline{11} \\
 0011 \\
 \underline{11} \\
 1110 \ 0101
 \end{array}$$

Add 0011 to 0010

subtract 0011 from 1110

$$\begin{array}{r} 1110 \ 0101 \\ - 0011 \\ \hline 1011 \ 0101 \end{array}$$

(28)

Since the carry is not generated the result is negative.
Find the 10's complement of the result to get the final result.

9's Complement of the result = 0100 1010

10's Complement of the result = 0100 1011

$$\therefore (39)_{10} - (57)_{10} = - (0100 1011)_{\text{Excess-3}} = -(18)_{10}$$

Practice problem: perform i) $(48)_{10} - (19)_{10}$ ii) $(52)_{10} - (81)_{10}$ in Excess-3 Code using 10's complement.

Non weighted Codes: The non weighted codes are not assigned with any weight to each digit position.

Excess-3 Code and Gray Code are the non weighted codes.

Reflective Code:

A reflective code is a binary code in which the n least significant bits for code words 2^n through $2^{n+1}-1$ are the mirror images for the code words for 0 through 2^n-1 .

Reflective code example is gray code.

Sequential Code:

Sequential code is one in which each succeeding code word is one binary number greater than its preceding code word.

Ex: The Excess-3 code and 8421 code are the examples.

Gray code:

Gray code is a unit distance code. Because in this code any two code words differ in one bit position only.

The gray code is a reflective code also. Because the three least significant bits of $(8)_{10}$ through $(15)_{10}$ are the mirror images of those for $(0)_{10}$ through $(7)_{10}$, in the case of four bit gray code.

For three bit gray code the two least significant bits for $(4)_{10}$ through $(7)_{10}$ are the mirror images of those for $(0)_{10}$ through $(3)_{10}$.

The gray code is used in the applications in which the normal sequence of binary numbers may produce an error (or) ambiguity during the transition from one number to the next. For example if binary numbers are used, a change from 0111 to 1000 may produce 1001 if the value of the rightmost bit takes longer time to change than the other three bits. The gray code eliminates this problem, because only one bit changes its value during the transition between any two successive numbers.

The following table shows the gray code and binary code for decimal numbers 0 through 15.

Decimal Number	Binary Equivalent	Gray code
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Binary to Gray code Conversion:

The binary to gray code conversion can be achieved using the following steps.

- 1> The MSB of the gray code is the same as the MSB of the given binary number. So write down MSB as it is.
- 2> To obtain the next gray digit, perform the exclusive OR operation between the previous bit and current binary bit and write down the result.
- 3> Repeat the step 2 until all binary bits have been exclusive-ORed with their previous ones.

Example: Convert 10111011 in binary into its equivalent gray code.

sol) Given Binary code 10111011

	MSB								
binary	1	0	1	1	1	0	1	1	
		\oplus		\oplus		\oplus		\oplus	
	1	0	1	0	1	1	1	0	
	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow	\downarrow
Gray code:	1	1	1	0	0	1	1	0	

$$\therefore (10111011)_2 = (11100110)_{\text{gray}}$$

Practice problem Convert i) $(11010110)_2$ ii) $(10011011)_2$ into their equivalent Gray codes.

Gray code to binary Conversion:

The gray code to binary conversion can be achieved using the following steps.

- 1> The MSB of the binary number is same as the MSB of the given gray code number. So write down the MSB as it is.
- 2> To obtain the next binary digit, perform the exclusive-OR operation between the bit just written down and the next gray code bit. Write down the result.
- 3> Repeat step 2 until all the gray code bits have been exclusive-ORed with binary digits.

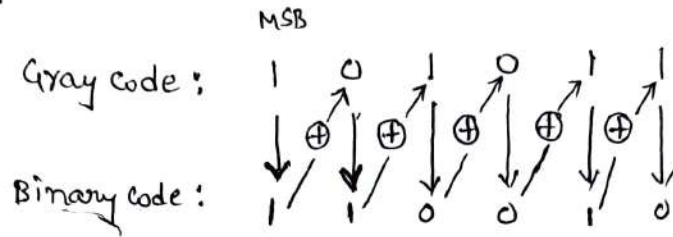
A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Fig: Exclusive OR operation.

Example: Convert gray code 101011 into its binary equivalent.

Sol)

Give gray code is 101011.



$$\therefore (101011)_{\text{Gray code}} = (110010)_{\text{Binary}}$$

Practice problem: Convert i) $(11100101)_{\text{Gray}}$ ii) $(100110110)_{\text{Gray}}$ into binary.

Generate a fourbit gray code sequence using mirror image property (i.e. reflective property)

Sol

2bit Gray code

```

00
01
---
11
10

```

3bit Gray code

```

000
001
011
010
---
110
111
101
100

```

4-bit graycode

```

0000
0001
0011
0010
0110
0111
0101
0100
---
1100
1101
1111
1110
1010
1011
1001
1000

```

Alphanumeric Codes:

(30)

The Codes which consists of both numbers and alphabetic characters are called as alphanumeric codes. Most of these codes however, also represent symbols and various instructions necessary for conveying information.

The most commonly used alphanumeric codes are

- 1) ASCII (American Standard Code for Information Interchange)
- 2) EBCDIC (Extended Binary Coded Decimal Interchange Code)

ASCII Code:

The Standard binary code for the alphanumeric characters is the ASCII code, which uses 7 bits to code 128 characters. The 7 bits of the code are designated by b_1 through b_7 with b_7 as MSB. The ASCII code contains 94 graphic characters that can be printed and 34 non printable characters that are used for various control functions.

The graphic characters consist of the 26 uppercase letters (A through Z), the 26 lowercase letters (a through z), the 10 numerals (0 through 9) and 32 special printable characters such as %, *, #, :, ;, ", \$ etc.

0-9 (Numerals)	48	57
	0110000	- 0111001
A-Z (uppercase letters)	65	90
	1000001	- 1011010
a-z (lowercase letters)	97	122
	1100001	- 1111010

The control characters are used for routing data and arranging the printed text into a prescribed format. There are three types of

- Control Characters: They are Format effectors, information separators and communication control characters.

Format effectors: They are used to control the layout of printing.

Ex: Backspace, Horizontal Tab etc.

Information separators: They are used to separate the data into divisions such as paragraphs and pages. They include the characters such as record separator (RS) and File separator (FS)

Communication control characters: They are used to frame a text message transmitted through a communication channel.

Ex: Start of Text (STX), End of text (ETX).

EBCDIC (Extended Binary Coded Decimal Interchange Code):

EBCDIC code is an 8-bit alphanumeric code. It is an 8-bit code that can code 256 characters. This code includes all the symbols and control characters that are present in ASCII. In addition to this, it includes many other symbols also.

Lower case Letters	a	i	j	r	s	z
	129	137	145	153	162	169
	1000001 - 10001001		1001001 - 10011001		10100010 - 10101001	

Upper case Letters	A	I	J	R	S	Z
	193	201	209	217	226	233
	11000001 - 11001001		11010001 - 11011001		11100010 - 11101001	

Numerals	0	9
	240	249
	11110000 - 11111001	

Ex: Encode the word "BINARY" in ASCII Code.

Sol.

B	I	N	A	R	Y
66	73	78	65	82	89
1000010 1001001 1001110 1000001 1010010 1011001					

Practice problem: Convert the following into ASCII codes

- i) Capital City ii) NEW DELHI

Error detecting and correcting codes:

When the digital information in the binary form is transmitted from one system to another system, an error may occur. This means a signal corresponding to '0' may change to '1' or vice versa due to the presence of noise. To maintain the data integrity between the transmitter and receiver an extra bit (or) more than one bit are added to the data. These extra bits allow the detection and sometimes correction of the error in the data. The data along with the extra bits forms the code.

The codes which allow only error detection are called error-detecting codes and the codes that allow error detection and correction are called as error detecting and correcting codes.

Example for error detecting code is parity bit method, and for error detecting and correcting code the example is -
- Hamming code.

i) Parity bit method for error detection:

* A Parity bit is used for the purpose of detecting errors during the transmission of binary information.

* A Parity bit is an extra bit included with a binary message to make the number of 1's either odd (or) even. The message, including the parity bit is transmitted and then checked at the receiving end for errors. An error is detected if the checked Parity does not correspond with the one transmitted.

* The circuit that generates the parity bit in the transmitter is called a parity generator.

* The circuit that checks the parity in the receiver is called Parity checker.

* In even parity the added parity bit will make the total number

of 1's as even, whereas in odd parity the added parity bit will make the total number of 1's as odd.

The following table shows a 3-bit message with even parity and odd parity.

Message with even parity		message with odd parity	
Message	Parity bit	message	parity bit
000	0	000	1
001	1	001	0
010	1	010	0
011	0	011	1
100	1	100	0
101	0	101	1
110	0	110	1
111	1	111	0

Ex: Write the ASCII code for decimal digit 9 with an even parity. Place parity bit in the most significant position.

Sol) Decimal equivalent for the ASCII code for 9 is $(57)_{10}$, and its ASCII code is 0111001.

To get the even parity for the ASCII code 0111001, the parity bit to be added is '0'.

After adding Parity bit '0' in MSB position the code is 00111001.

Ex: Write the ASCII code for the alphabet A with an odd parity. Place the parity bit in the MSB position.

Sol) The decimal equivalent for the ASCII code of 'A' is 65.

The ASCII code for 'A' is = 1000001

To get odd parity for the ASCII code 1000001, the parity bit to be added is '1'. After adding the parity bit in MSB position the code becomes 11000001

After receiving the message with parity bit at the receiver, the receiver checks for the Concerned Parity. If there is an error in the Parity the receiver will request the transmitter to re-send the message.

ii) Hamming Code for error detection and correction:

Hamming code not only provides the detection of a bit error but also identifies which bit is in error so that it can be corrected. Hence the hamming Code is called as error detecting and error correcting code. The code uses a number of parity bits located at certain positions in the code group.

The number of parity bits to be included depends on the number of information bits. If the number of information bits is designated as x , then the number of parity bits ' p ' is determined by the following relationship.

$$2^p \geq x + p + 1 \quad \text{----- ①}$$

For example if we have a 4-bit message i.e. $x=4$ then p is found by trial and error using the above equation. Let $p=2$, then

$$2^p = 2^2 = 4 \quad \text{and} \quad x + p + 1 = 4 + 2 + 1 = 7$$

Since 2^p must be equal to (or) greater than $x + p + 1$, the relationship in equation ① is not satisfied. Hence we have to try with next value of p . Let $p=3$.

$$\text{Then } 2^p = 2^3 = 8 \quad \text{and} \quad x + p + 1 = 4 + 3 + 1 = 8$$

This value of p satisfies the relationship given in equation ① therefore we can say that three parity bits are required to provide single error correction for four information bits.

Locations of the parity bits in the hamming code:

The parity bits are located in the positions of ascending powers of 2 i.e. $2^0, 2^1, 2^2, 2^3, \dots$ (i.e. 1, 2, 4, 8, \dots). Therefore for 7-bit code, locations for parity bits and information bits are shown below.

7 6 5 4 3 2 1
 D_7 D_6 D_5 P_4 D_3 P_2 P_1

where symbol P_n designates a particular Parity bit and D_n designates a particular information bit and n is the location number.

Let us see how to determine the value of each Parity bit.
 To do this we must write the binary number for each decimal location number as shown in table below.

Bit Designation	D_7	D_6	D_5	P_4	D_3	P_2	P_1
Bit location	7	6	5	4	3	2	1
Binary equivalent of the bit location no.	111	110	101	100	011	010	001
Information bits							
Parity bits							

Assignment of P_1 : Looking at the table, we can see that Parity bit P_1 has a '1' in its rightmost digit. This Parity bit checks all the bit locations including itself that have 1's in the same location. Therefore Parity bit P_1 checks bit locations 1, 3, 5, 7 and assigns P_1 according to even or odd Parity. For even parity hamming code, it assigns P_1 such that bit locations 1, 3, 5 and 7 will have even Parity.

Assignment of P_2 : Looking at the table, we can see that the Parity bit P_2 has a '1' in its middle bit position. This Parity bit checks all the bit locations including itself that have 1's in the middle bit. Therefore, Parity bit P_2 checks the bit locations 2, 3, 6, 7 and assigns P_2 according to even or odd Parity.

Assignment of P_4 : Looking at the table, we can see that the Parity bit P_4 has a '1' in its leftmost bit position. This Parity bit checks all the bit locations including itself that have 1's in the leftmost bit. Therefore Parity bit P_4 checks the bit locations 4, 5, 6 and 7 and assigns P_4 according to even or odd Parity.

Example:

Encode the binary word 1011 into 7-bit even parity hamming code.

Sol) step 1) Given message = 1011, number of information bits = $x = 4$

Let the number of parity bits required be 'P', then it should

$$\text{satisfy } 2^P \geq x + P + 1.$$

$P = 3$ satisfies the above condition.

Since the given message bits are 4, after including 3 parity bits the hamming code will have $4 + 3 = 7$ bits.

step 2)

Bit Designation	D_7	D_6	D_5	P_4	D_3	P_2	P_1
Bit Location	7	6	5	4	3	2	1
Binary Equivalent of the bit location	111	110	101	100	011	010	001
Information bits	1	0	1		1		
Parity bits				0		0	1

fig: Bit location table.

step 3) Determine the parity bits.

For P_1 : Bit locations 3, 5 and 7 have three 1's and therefore to have an even parity P_1 must be '1'.

For P_2 : Bit locations 3, 6 and 7 have two 1's and therefore to have an even parity P_2 must be '0'.

For P_4 : Bit locations 5, 6 and 7 have two 1's and therefore to have an even parity P_4 must be '0'.

step 4) Enter the parity bits into the table to form a 7-bit hamming code 1010101.

practice problem: Determine the hamming code for a binary message

1101 using even parity.

Detecting and Correcting an error using hamming code:

Here we will see how to use a hamming code to locate and correct an error. To do this each parity bit along with its group of bits must be checked for proper parity. The correct result of individual parity check is marked as '0' where as wrong result of parity check is marked by '1'. After all Parity checks, a binary word is formed taking the bit of P, as LSB. The binary word formed like this, gives the bit location where error has occurred. If the binary word has all 0's then there is no error in the hamming code.

Example: Assume that the even parity hamming code in the previous example is (1010101) is transmitted and 1000101 is received. The receiver does not know what was transmitted. Determine bit location where error has occurred using the received code.

Sol Step 1: Construct bit location table

Bit designation	D_7	P_6	D_5	P_4	D_3	P_2	P_1
Bit Location	7	6	5	4	3	2	1
Binary equivalent for bit location	111	110	101	100	011	010	001
Received Code	1	0	0	0	1	0	1

Step 2: Check for parity bits.

For P_1 : P_1 checks the locations 1, 3, 5, 7

There are three 1's in the group of 1, 3, 5, 7.

\therefore Parity checks for even parity is wrong. $\Rightarrow 1$ (LSB)

For P_2 : P_2 checks the locations 2, 3, 6, 7

There are two 1's in the group of 2, 3, 6, 7.

\therefore Parity checks for even parity is correct. $\Rightarrow 0$

For P_4 : P_4 checks the locations 4, 5, 6, 7

There is only one 1 in the group of 4, 5, 6, 7.

\therefore Parity checks for even parity is wrong $\Rightarrow 1$.

\therefore The resultant word is 101. This says that the bit in the bit location '5' is in error. In the received code it is '0'. Hence it should be corrected as '1'.

\therefore The corrected code is 1010101.

Practice problem: The hamming code 101101101 is received. Correct it if there is any error. There are four parity bits and odd parity is used.

single error correction and double error detection:

The hamming code facilitates the detection and correction of only a single error. With a slight modification, it is possible to construct a hamming code that detects double error and corrects single error. One more parity bit is added in the hamming code to ensure that the hamming code has even number of 1's. The added parity bit is not used in determining the values of other parity bits. The resulting hamming code enables single error correction and double error detection.

After receiving the code word, if the over all parity is even then it is even parity represented as $P=0$ or else $P=1$.

In the received code word the parity check bits are evaluated. If the word formed by the check bits is 000 we consider that case as $C=0$, else (if it is a non zero) it is considered as $C=1$.

Based on C and P values there are 4 cases.

- i) If $C=0$ and $P=0$ no error has occurred.
- ii) If $C \neq 0$ and $P=1$ a single error has occurred that can be corrected.
- iii) If $C \neq 0$ and $P=0$ double error occurred that is detected, but that cannot be corrected.
- iv) If $C \neq 0$ and $P=1$ an error occurred in additional parity bit position.

Example: Given the 8-bit data word 01011011, generate the 13-bit composite word for the hamming code that corrects single errors and detects double errors.

Sol > Step 1: Given data word = 01011011

Number of information bits = 8 = x

Let the number of parity bits = P .

Then $2^P \geq x + P + 1$

$2^P \geq 8 + P + 1$

Let $P = 3$ $2^P = 8$, $8 + P + 1 = 8 + 3 + 1 = 12$, $2^P \geq 8 + P + 1$ is not satisfied.

Let $P = 4$, $2^P = 16$, $8 + P + 1 = 8 + 4 + 1 = 13$, $2^P \geq 8 + P + 1$. Condition is satisfied.

$\therefore P = 4$.

Step 2: Construct bit location table.

Bit designation	P_{13}	D_{12}	D_{11}	D_{10}	D_9	P_8	D_7	D_6	D_5	P_4	D_3	P_2	P_1
Bit location	13	12	11	10	9	8	7	6	5	4	3	2	1
Binary equivalent for bit location	1101	1100	1011	1010	1001	1000	0111	0110	0101	0100	0011	0010	0001
Information bits		0	1	0	1		1	0	1		1		
Parity bits						0				0		1	1

Step 3: determine the parity bits.

For P_1 : The bit locations 3, 5, 7, 9, 11 have five 1's and therefore to have an even parity P_1 must be 1.

For P_2 : The bit locations 3, 4, 7, 10, 11 have Three 1's and therefore to have an even parity P_2 must be 1.

For P_4 : The bit locations 5, 6, 7, 12 have two 1's and therefore to have an even parity P_4 must be '0'.

For P_8 : The bit locations 9, 10, 11, 12 have two 1's and therefore to have an even parity P_8 must be '0'.

For P_{13} : since the bits 1 to 12 contain odd number of 1's.

Therefore P_{13} should be 1 for even parity.

Thus the hamming code that corrects single error and detects double errors for the data word 01011011 is = 1010101010111

Practice problem: Given a 7-bit data word 1100111, generate the composite word for the hamming code that corrects single errors and detects double errors.

Self complementing codes:

A Code is said to be self complementing code, if the code for 9 is the complement for 0, the code for 8 is the complement for 1, the code for 7 is the complement for 2, the code for 6 is the complement for 3 and the code for 5 is the code for 4.

Examples for self complementing codes are 2421, 5211 and excess-3.

	2	4	2	1	Code
Code for 9	1	1	1	1	} Complement
Code for '0'	0	0	0	0	

	2	4	2	1	Code
Code for 5	0	1	0	1	} Complement
Code for 4	1	0	1	0	

fig: Self Complementing codes example.

Binary storage and registers:

In digital Computers, the binary information is stored using the binary cells. A binary cell is capable of storing one bit of information. The cell can have two possible states. They are logic 1 or logic '0'. When the cell is in logic 1 state, the information stored in it is 1. When the cell is in logic 0 state the information stored in it is 0.

Registers:

A register is a group of binary cells. A register with n binary cells can store n -bit information. We know that each bit in the register can have either 0 or 1 value. Therefore a 16-bit register can have 2^{16} possible numbers. The information stored in the register may be

interpreted differently. The registers can be used to store excess-3 Code, binary code, BCD code, gray code or any other code. Likewise the number stored in the register has different interpretations according to the desired code.

Register Transfer:

We know that the binary information can be stored in the registers. The registers are the part of the digital system. The processing unit in the digital system gets the information to be processed from the registers. The processed information is again stored in the registers. Some times it is necessary to transfer the information from one register to another register. Such operation is known as register transfer operation.

Usually the information is stored in the memory. At the time of processing it is brought into the registers so that the processing unit gets access to it. The processed information available in the registers is stored into the memory.

The following figure shows how the information is transferred between the processing unit, registers and memory unit.

It is important to know that, the processing unit is incapable of processing information stored in the memory registers. Therefore it is necessary to transfer the information to be processed into the registers of the processing unit. Usually, the processing unit has a limited number of registers and hence many times the information to be processed is brought into registers of processing unit, before processing and the processed result is stored back into the memory registers after processing.

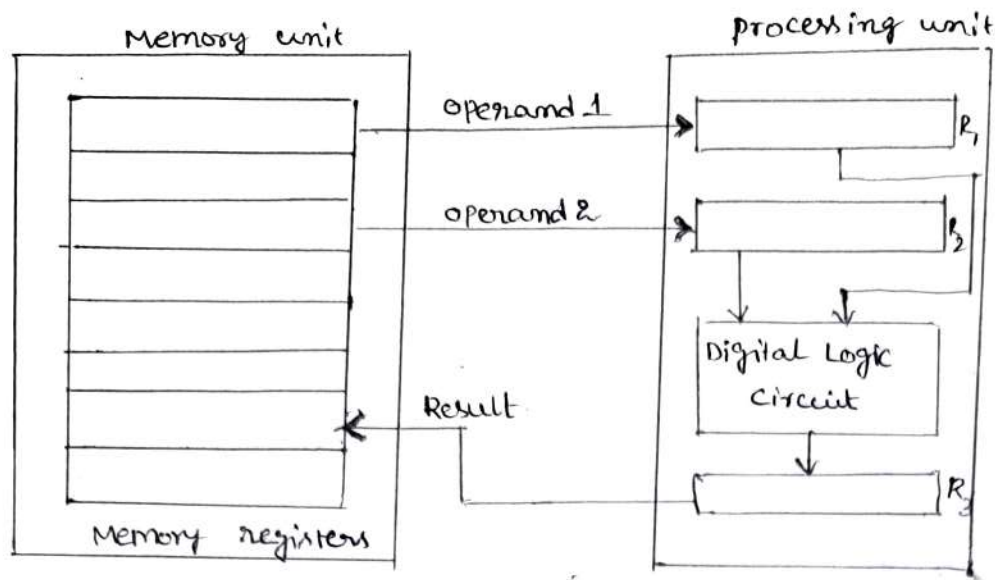


Fig: Binary information processing blocks.

Binary Logic:

Binary Logic consists of binary variables and a set of logical operations. The variables are designated by the letters of the alphabet such as A, B, C, x, y, z etc with each variable having two and only two distinct possible values. They are 0 and 1.

There are three basic logical operations: AND, OR, and NOT. Each operation produces a binary result denoted by Z.

AND: This operation is represented by a dot or by the absence of an operator. For example $x \cdot y = z$ (or) $xy = z$ and is read as "x AND y is equal to z". This means that $z=1$ if and only if $x=1$ and $y=1$ otherwise $z=0$.

OR: This operation is represented by a plus sign. For example, $x + y = z$ is read as "x OR y is equal to z", meaning that $z=1$ if $x=1$ or if $y=1$ or if both $x=1$ and $y=1$. If both $x=0$ and $y=0$ then $z=0$.

NOT: This operation is represented by a prime (or) sometimes by an overbar. For example $x' = z$ (or) $\bar{x} = z$ is read as "not x is equal to z", meaning that if $x=1$, $z=0$ or else, if $x=0$ then $z=1$. The NOT operation is also referred as the complement operation.

Binary logic resembles binary arithmetic, and the operations AND, OR have the similarities to multiplication and addition respectively. The symbols used for AND and OR the same as those for multiplication and addition. However binary logic should not be confused with binary arithmetic.

One should realise that an arithmetic variable designates a number that may consist of many digits. But a logic variable is always either 1 or 0. For example in binary arithmetic, we have $1+1=10$, where as in binary logic we have $1+1=1$.

For each combination of the values of x and y , there is a value of z specified by the definition of the logical operation. The definitions of the logical operations may be listed in a compact form called truth tables. A truth table is a table of all possible combinations of the variables, that shows the relation between the values that the variables may take and the result of the operation.

The truth tables for the operations AND and OR with the variables x and y are obtained by listing all possible values of x and y .

For each combination of x and y the result of the operation is also included in the table.

The following tables show the truth tables for AND, OR and NOT operations.

AND

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

NOT

x	x'
0	1
1	0

Boolean Algebra and Logic Gates:

37

Boolean Algebra:

In 1854 George Boole introduced an algebraic system, which is now called as Boolean Algebra.

Basic Definitions:

- * The Boolean Algebra is defined with a set of elements, a set of operators and a number of rules, laws, theorems and postulates.
- * The postulates of a mathematical system are the basic assumptions from which it is possible to deduce the laws, theorems, rules and properties of the system.
- * The Boolean Algebra is formulated by a defined set of elements together with two binary operators '+' and '·'.
- * Set: A set is a group of objects having a common property. If 'S' is a set and x and y are the objects of S. Then $x \in S$, $y \in S$ denotes x, y are the elements of S. Then if $z \notin S$, denotes z is not the element of set 'S'.

Closure: A particular set is closed with respect to a binary operator if, every pair of elements of set obtains a unique element of the same set after being operated by that operator.

Laws of Boolean Algebra:

Associative Law:

Law 1: $A + (B + C) = (A + B) + C$; This law states that the OR'ing of several variables gives the same result regardless of grouping of the variables.

Law 2: $(A \cdot B) \cdot C = A \cdot (B \cdot C)$; This law states that the AND'ing of several variables gives the same result regardless of the grouping of the variables.

Commutative Law:

Law 1: $A+B = B+A$: This states the order in which the variables are ORed makes no difference in the output.

A	B	A+B	B+A
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

From the truth table it is clear that $A+B = B+A$

Law 2: $A \cdot B = B \cdot A$ This states the order in which the variables are ANDed makes no difference in the output.

A	B	A · B	B · A
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

From the truth table it is clear that $A \cdot B = B \cdot A$

Distributive Law: Law 1: $A \cdot (B+C) = A \cdot B + A \cdot C$

Law 2: $A + B \cdot C = (A+B) \cdot (A+C)$

Axiomatic Definition of Boolean Algebra:

The postulates of a mathematical system forms the basic assumption from which it is possible to deduce the theorems, laws and properties of the system. The postulates are also called as axioms.

The postulates of the boolean algebra are also called as Huntington postulates as they were proposed by E.V. Huntington, in the year 1904. The postulates of boolean algebra are discussed below

1) closure: closure with respect to '+' operator.

When two binary elements are operated by an operator '+' the result is a unique binary element.

Closure(b): Closure with respect to the operator \cdot (dot).

When two binary elements are operated by the operator \cdot (dot), the result is a unique binary element.

2a) An identity element with respect to $+$ is designated by '0':

$$A + 0 = 0 + A = A$$

2b) An identity element with respect to \cdot is designated by '1':

$$A \cdot 1 = 1 \cdot A = A$$

3a) commutative with respect to $+$: $A + B = B + A$

3b) Commutative with respect to \cdot : $A \cdot B = B \cdot A$

4a) Distributive property of \cdot over $+$: $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

4b) Distributive property of $+$ over \cdot : $A + (B \cdot C) = (A + B) (A + C)$

5) For every binary element, there exists complement element.

For example if A is an element, we have A' which is the complement of A i.e. if $A = 0$, $A' = 1$ and if $A = 1$, $A' = 0$. Then $A + A' = 1$ and $A \cdot A' = 0$.

We can summarize these Postulates of boolean algebra as shown in the table below.

Postulate	a	b
Postulate 1	When two binary elements are operated by the operator $+$ the result is a unique binary element	When two binary elements are operated by the operator \cdot (dot) the result is a unique binary element
Postulate 2 (Identity)	$A + 0 = A$	$A \cdot 1 = A$
Postulate 3 (Commutative)	$A + B = B + A$	$A \cdot B = B \cdot A$
Postulate 4 (Distributive)	$A \cdot (B + C) = A \cdot B + A \cdot C$	$A + (B \cdot C) = (A + B) (A + C)$
Postulate 5	$A + A' = 1$	$A \cdot A' = 0$

BASIC THEOREMS:

We can define the following theorems using fundamental postulates of Boolean Algebra.

Theorem 1(a): $A + A = A$

Proof:

$$\begin{aligned} A + A &= (A + A) \cdot 1 && [\because \text{Postulate 2(b)}] \\ &= (A + A) (A + \bar{A}) && [\because \text{Postulate 5(a)}] \\ &= A + A\bar{A} && [\because \text{Postulate 4(b)}] \\ &= A + 0 && [\because \text{Postulate 5(b)}] \\ &= A && [\because \text{Postulate 2(a)}] \end{aligned}$$

$$\therefore A + A = A$$

Theorem 1(b): $A \cdot A = A$

Proof:

$$\begin{aligned} A \cdot A &= (A \cdot A) + 0 && [\because \text{Postulate 2(a)}] \\ &= AA + A\bar{A} && [\because \text{Postulate 5(b)}] \\ &= A(A + \bar{A}) && [\because \text{Postulate 4(a)}] \\ &= A \cdot 1 && [\because \text{Postulate 5(a)}] \\ &= A && [\because \text{Postulate 2(b)}] \end{aligned}$$
$$\therefore A \cdot A = A$$

Theorem 2(a): $A + 1 = 1$

Proof:

$$\begin{aligned} A + 1 &= 1 \cdot (A + 1) && [\because \text{by Postulate 2(b)}] \\ &= (A + \bar{A}) (A + 1) && [\because \text{Postulate 5(a)}] \\ &= A + \bar{A} \cdot 1 && [\because \text{Postulate 4(b)}] \\ &= A + \bar{A} && [\because \text{Postulate 2(b)}] \\ &= 1 && [\because \text{Postulate 5(a)}] \end{aligned}$$

$$\therefore A + 1 = 1$$

Theorem 2(b): $A \cdot 0 = 0$

Proof:

$$\begin{aligned}
 A \cdot 0 &= A \cdot 0 + 0 && [\text{Postulate 2(a)}] \\
 &= A \cdot 0 + A \cdot A' && [\text{Postulate 5(b)}] \\
 &= A \cdot (0 + A') && [\text{Postulate 4(a)}] \\
 &= A \cdot A' && [\text{Postulate 2(a)}] \\
 &= 0 && [\text{Postulate 5(b)}]
 \end{aligned}$$

$$\therefore A \cdot 0 = 0.$$

Theorem 3:

$$(A')' = A$$

Proof:

$$\text{Let } A = 0, \text{ then } A' = 1, (A')' = 0$$

$$\therefore A = (A')' = 0$$

$$\text{Let } A = 1 \text{ Then } A' = 0, (A')' = 1$$

$$\therefore A = (A')' = 1$$

$$\therefore A = (A')'$$

Theorem 4(a): $A + AB = A$

Proof:

$$\begin{aligned}
 A + AB &= A \cdot 1 + AB && [\text{Postulate 2(b)}] \\
 &= A(1 + B) && [\text{Postulate 4(a)}] \\
 &= A \cdot (1) && [\text{Theorem 2(a)}] \\
 &= A && [\text{Postulate 2(b)}]
 \end{aligned}$$

$$\therefore A + AB = A$$

Theorem 4(b): $A \cdot (A + B) = A$

Proof:

$$\begin{aligned}
 A \cdot (A + B) &= (A + 0) \cdot (A + B) && [\text{Postulate 2(a)}] \\
 &= A + (0 \cdot B) && [\text{Postulate 4(b)}] \\
 &= A + 0 && [\text{Theorem 2(b)}] \\
 &= A && [\text{Postulate 2(a)}]
 \end{aligned}$$

Theorem 5(a): $A + A'B = A + B$

Proof:

$$\begin{aligned}
 A + A'B &= A + AB + A'B && [\because \text{Theorem 4(a)}] \\
 &= A + (A + A')B && [\because \text{Postulate 4(a)}] \\
 &= A + (1 \cdot B) && [\because \text{Postulate 5(a)}] \\
 &= A + B && [\because \text{Postulate 2(b)}]
 \end{aligned}$$

Theorem 5(b): $A \cdot (A' + B) = AB$

Proof:

$$\begin{aligned}
 A \cdot (A' + B) &= A \cdot (A + B) (A' + B) && [\because \text{Theorem 4(b)}] \\
 &= A \cdot [B + AA'] && [\because \text{Postulate 4(b)}] \\
 &= A \cdot [B + 0] && [\because \text{Postulate 5(b)}] \\
 &= A \cdot [B] && [\because \text{Postulate 2(a)}] \\
 &= AB
 \end{aligned}$$

$\therefore A \cdot (A' + B) = AB$

All the above theorems are summarized in the following table.

Theorems	a	b
Theorem 1 (Idempotency)	$A + A = A$	$A \cdot A = A$
Theorem 2 (Null)	$A + 1 = 1$	$A \cdot 0 = 0$
Theorem 3 (Involution)	$(A')' = A$	
Theorem 4 (absorption)	$A + AB = A$	$A \cdot (A + B) = A$
Theorem 5 (adsorption)	$A + A'B = A + B$	$A \cdot (A' + B) = AB$
Theorem 6 Demorgan's Theorem	$(A + B)' = A' \cdot B'$	$(AB)' = A' + B'$

Demorgan's Theorems:

Demorgan suggested two theorems that form an important part of the Boolean Algebra. Demorgan's Theorem states that

$$(AB)' = A' + B' \quad \text{and} \quad (A + B)' = A' \cdot B'$$

Demorgan's theorem: a) $(AB)' = A' + B'$

(40)

Proof:

The Demorgan's Theorem can be proved by means of truth table

A	B	AB	$(AB)'$	A'	B'	$A' + B'$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

fig: Truth table

from the truth table it is clear that $(AB)' = A' + B'$.

Demorgan's Theorem(b): $(A+B)' = A' \cdot B'$

Proof:

The demorgan's theorem can be proved by means of truth table

A	B	A+B	$(A+B)'$	A'	B'	$A' \cdot B'$
0	0	0	1	1	1	1
0	1	1	0	1	0	0
1	0	1	0	0	1	0
1	1	1	0	0	0	0

fig: Truth table

from the truth table it is clear that $(A+B)' = A' \cdot B'$

NOTE:

$$\text{similarly } (A+B+C)' = A' \cdot (B+C)'$$

$$= A' \cdot B' \cdot C'$$

$$(A \cdot B \cdot C)' = A' + (B \cdot C)'$$

$$= A' + B' + C'$$

We can also prove that $(A+B+C+D+E+\dots+Z)' = A' \cdot B' \cdot C' \cdot D' \cdot E' \cdot \dots \cdot Z'$

$$(A \cdot B \cdot C \cdot D \cdot E \cdot \dots \cdot Z)' = A' + B' + C' + D' + E' + \dots + Z'$$

Duality principle: This principle is a very important principle of Boolean algebra. Duality principle says that starting with a Boolean expression, we can derive another Boolean expression by following the steps given below.

- 1) Change each OR with AND and AND with OR.
- 2) Replace 0's by 1's and 1's by 0's that are appearing in the given expression.

Consensus Theorem:

While simplifying the Boolean expression of the form $AB + A'C + BC$ the term BC is redundant and can be eliminated to form $AB + A'C$.

$$\text{i.e. } AB + A'C + BC = AB + A'C.$$

Proof:

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

$$\therefore AB + A'C + BC = AB + A'C.$$

Dual of the Consensus theorem: The dual of the consensus theorem

says that $(A + B)(A' + C)(B + C) = (A + B)(A' + C).$

Example:

Solve $A'B' + AC + BC' + B'C + AB$ using Consensus theorem.

Sol:

$$\begin{aligned} A'B' + AC + BC' + B'C + AB &= \underbrace{A'B' + AC + B'C + BC'}_{\downarrow} + AB \\ &= A'B' + AC + BC' + AB \\ &= A'B' + \underbrace{CA + C'B + AB}_{\downarrow} \\ &= A'B' + CA + C'B \end{aligned}$$

$$\therefore A'B' + AC + BC' + B'C + AB = A'B' + AC + BC'$$

(41)

Example Solve $(A+B)(A'+C)(B+C)(A'+D)(B+D)$ using the dual of Consensus theorem.

$$\begin{aligned} \text{Sol} \rangle \text{ Given } (A+B)(A'+C)(B+C)(A'+D)(B+D) &= (A+B)(A'+C)(A'+D)(B+D) \\ &= (A+B)(A'+D)(B+D)(A'+C) \\ \therefore (A+B)(A'+C)(B+C)(A'+D)(B+D) &= (A+B)(A'+D)(A'+C). \end{aligned}$$

Boolean functions:

Boolean Algebra is an algebra that deals with binary variables and logic operations. A boolean function described by an expression consists of binary variables, the constants 0 and 1 and the logic operation symbols.

For the given values of binary variables, the function can be equal to either 1 (or) 0. For example consider the Boolean function

$$F_1 = x + y'z.$$

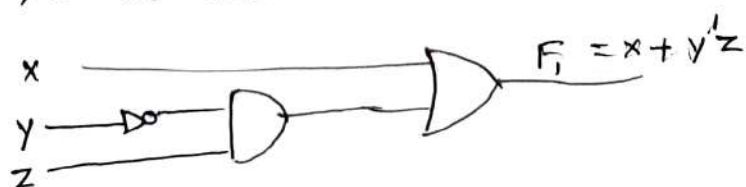
The function F_1 is equal to 1, if x is equal to 1 (or) if both y' and z are equal to 1. otherwise F_1 is equal to 0. If y' is 1, y should be equal to 0.

Therefore $F_1 = 1$ if $x=1$ (or) if $y=0$ and $z=1$.

The boolean function can be represented in a truth table. The number of rows in a truth table is 2^n , where n is the number of variables in the Boolean function. The following table shows the truth table for $F_1 = x + y'z$

X	Y	Z	F_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	1

A boolean function can be transformed from an algebraic expression into a circuit diagram composed of logic gates. The logic diagram for $F_1 = x + y'z$ is shown below



Algebraic Manipulation:

When a boolean expression is implemented with logic gates, each term requires a gate and each variable within the term designates an input to that gate. A variable within a term, either in Complemented form (0) or in uncomplemented form is said to be a literal.

By reducing the number of terms, number of literals in a boolean expression, it is possible to obtain a simple circuit.

The techniques used for the simplification of boolean expression is discussed below.

- 1) Multiply all variables necessary to remove parenthesis.
- 2) Look for identical terms. Only one of those terms is retained and all the other terms are dropped.

Ex: $AB + AB + AB + AB = AB$
 $AC' + AC' + AC' = AC'$

- 3) Look for a variable and its negation in the same term. This term can be dropped.

Ex: $ABCC' = AB(0) = 0$; $ABB = A(0) = 0$.

- 4) Look for pairs of terms that are identical except one variable is appearing extra in one of the two terms. Then the larger term can be dropped.

$$ABC' + ABC'D' = ABC'(1 + D') = ABC'(1) = ABC'$$

- 5) Look for pairs of terms which have the same variables such that a variable in one of the term is complemented while in the other term it is not, then such terms are combined into a single term by dropping that variable.

Ex: $ABC'D + AB'C'D = AC'D(B + B') = AC'D(1)$
 $\therefore ABC'D + AB'C'D = AC'D$

Example: Reduce the expression $f = A [B + C' (AB + AC')']$

Sol) Given $f = A [B + C' (AB + AC')']$ Applying De Morgan's Theorem

$$\begin{aligned}
 &= A [B + C' (AB)' \cdot (AC')'] \\
 &= A [B + C' (A' + B') (A' + C)] \\
 &= A [B + C' (A'A' + A'C + A'B' + B'C)] \\
 &= A [B + C' (0 + A'C + A'B' + B'C)] \\
 &= A [B + C' A'C + C' A'B' + C' B'C] \quad (\because C' \cdot C = 0) \\
 &= A [B + 0 + A'B'C' + 0] \\
 &= A (B + A'B'C') \\
 &= AB + AA'B'C' \quad (\because AA' = 0) \\
 \therefore f &= AB
 \end{aligned}$$

Example: $f = A + B [AC + (B + C')D]$

$$\begin{aligned}
 &= A + B (AC + BD + C'D) \\
 &= A + ABC + BBD + BC'D \quad (\because B \cdot B = B) \\
 &= A + ABC + BD + BC'D \\
 &= A(1 + BC) + BD(1 + C') \quad \left[\begin{array}{l} \because 1 + BC = 1 \\ 1 + C' = 1 \end{array} \right] \\
 &= A(1) + BD(1) \\
 &= A + BD
 \end{aligned}$$

Practice problems: 1) $f(A, B, C) = (A + B)(A + C') + A'B' + A'C'$ Simplify $f(A, B, C)$.

Ans: $f(A, B, C) = A + B' + C'$
 2) Simplify $f = (B + BC)(B + B'C)(B + D)$ (Ans: $f = B$)

3) $f(A, B, C) = AB'C + B + BD' + ABD' + A'C$, Simplify $f(A, B, C)$
 (Ans: $f(A, B, C) = B + C$)

Complement of a boolean expression:

The Complement of a boolean function can be derived by using Demorgan's Theorems. The Demorgan's theorems can be extended to 3(or) more Variables

3 variable Demorgan's theorem:

$$\begin{array}{l|l} (A+B+C)' = (A+x)' \quad (\text{let } x=B+C) & (ABC)' = (Ax)' \quad (\text{let } x=BC) \\ = A' \cdot x' & = A' + x' \\ = A' \cdot (B+C)' & = A' + (BC)' \\ \therefore (A+B+C)' = A' \cdot B' \cdot C' & \therefore (ABC)' = A' + B' + C' \end{array}$$

similarly $(A+B+C+D+E+F)' = A'B'C'D'E'F'$
 $(ABCDEF)' = A' + B' + C' + D' + E' + F'$

Example: Find the complement of $F_1 = x'yz' + x'y'z$

$$F_2 = x(y'z' + yz)$$

Sol)

Given $F_1 = x'yz' + x'y'z$ and $F_2 = x(y'z' + yz)$

$$\begin{aligned} F_1' &= (x'yz' + x'y'z)' = (x'yz')' \cdot (x'y'z)' \\ &= ((x')' + y' + (z')') \cdot ((x')' + (y')' + z') \\ &= (x + y' + z)(x + y + z') \quad [\because (x')' = x] \end{aligned}$$

$$\begin{aligned} F_2' &= [x(y'z' + yz)]' = x' + (y'z' + yz)' \\ &= x' + [(y'z')' \cdot (yz)'] \\ &= x' + [(y')' + (z')'] \cdot (y' + z') \\ &= x' + (y + z)(y' + z') \end{aligned}$$

NOTE: The complement of a boolean function can also be obtained by taking the dual of a boolean function and complementing each literal

Ex: Find the complement of the function $F = AB + A(B+C) + B'(B+D)$ (43)

Sol) Given function $F = AB + A(B+C) + B'(B+D)$

$$\text{Dual of } F = (A+B) \cdot (A+BC) \cdot (B'+BD)$$

$$\text{Complementing each literal in the dual of } F = (A'+B') (A'+B'C') (B'+B'D')$$

$$\therefore \text{Complement of } F = F' = (A'+B') (A'+B'C') (B'+B'D')$$

Practice problems: Complement i) $F = B'C'D + (B+C+D)' + B'C'D'E$
ii) $F = A + B'C (A+B+C')$

Canonical and standard forms:

Canonical form:

In a boolean function a binary variable may appear as in its normal form (or) in its complement form. Now consider two binary variables x and y combined with AND operation. Since each variable may appear in either form, there are 4 possible combinations: $x'y'$, $x'y$, xy' and xy . Each of these 4 AND terms is called as a minterm (or) a standard product term. Similarly for n variables there are 2^n possible minterms. Each minterm is formed by performing AND operation of n variables, with each variable being primed if the corresponding bit of the binary variable is '0' and unprimed if it is '1'. Each minterm is designated as m_j , where 'j' denotes the decimal equivalent of the binary number.

In the similar fashion when two binary variables are (say x and y) combined with OR operation, there are 4 possible combinations. They are $x'+y'$, $x'+y$, $x+y'$ and $x+y$. Each of these 4 OR terms are called as a max term (or) a standard sum term. Similarly for n variables there are 2^n possible max terms. Each

max term is formed by performing OR operation among n variables, with each variable is being primed if the corresponding bit of the binary variable is '1' and unprimed if it is '0'. Each maxterm is designated as M_j , where j represents the decimal equivalent of the binary number.

The following table shows the minterms and maxterms for 3 variables of binary.

x	y	z	Min term		Max term	
			Term	Designation	Term	Designation
0	0	0	$x'y'z'$	m_0	$x+y+z$	M_0
0	0	1	$x'y'z$	m_1	$x+y+z'$	M_1
0	1	0	$x'yz'$	m_2	$x+y'+z$	M_2
0	1	1	$x'yz$	m_3	$x+y'+z'$	M_3
1	0	0	$xy'z'$	m_4	$x'+y+z$	M_4
1	0	1	$xy'z$	m_5	$x'+y+z'$	M_5
1	1	0	xyz'	m_6	$x'+y'+z$	M_6
1	1	1	xyz	m_7	$x'+y'+z'$	M_7

* A boolean function can be expressed algebraically from a given truth table by forming a minterm for each combination of the variables that produces 1 in the function and then taking OR of all those terms.

* A boolean function can be expressed algebraically from a given truth table by forming a maxterm for each combination of the variables that produces 0 in the function and then taking the AND of all those terms.

For example consider the functions F_1 and F_2 as shown in below table.

x	y	z	F ₁	F ₂
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$F_1 = m_1 + m_4 + m_7$$

$$= x'y'z + xy'z' + xyz$$

$$= \Sigma m(1, 4, 7)$$

$$F_2 = x'yz + xy'z + xyz' + xyz$$

$$= m_3 + m_5 + m_6 + m_7$$

$$= \Sigma m(3, 5, 6, 7)$$

The above two functions are represented in terms of minterms. Now the following functions from the above table shows the functions in terms of maxterms.

$$F_1 = (x+y+z)(x+y+z)(x+y'+z')(x'+y+z')(x'+y'+z)$$

$$= M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_6$$

$$= \Pi M(0, 2, 3, 5, 6)$$

$$F_2 = (x+y+z)(x+y+z')(x+y'+z)(x'+y+z)$$

$$= M_0 \cdot M_1 \cdot M_2 \cdot M_4$$

$$= \Pi M(0, 1, 2, 4)$$

* The complement of a boolean function can be obtained from a truth table by forming the minterms for each combination that produces a '0' in the function and then ORing those terms.

Similarly the complement of a boolean function can be obtained from a truth table by forming the maxterms for each combination that produces a '1' in the function and then ANDing those terms.

- * Any boolean function can be expressed as a sum of minterms (or) as the product of maxterms. Boolean functions expressed as a sum of minterms is known as canonical sop form (or) minterm canonical form (or) sum of minterms form.
- * Boolean functions expressed as a product of maxterms is known as Canonical Pos form (or) maxterm canonical form (or) Product of maxterms form.
- * usually each minterm (or) maxterm contains, by definition all the variables either in complemented form (or) in uncomplemented form.

Standard forms: Another way to represent the boolean functions is in standard form. In this configuration each term of a boolean function may contain one, (or) two (or) any number of literals.

- * There are two types of standard forms. They are
 - 1) Sum of products form (sop form)
 - 2) Product of sum form (pos form).

* The sum of products form is a form in which a boolean function contains AND terms called product terms, of one (or) more literals each. The sum denotes the ORing of these terms. Example for sop form function is $F_1 = y' + xy + x'yz'$. This expression has three product terms y' , xy , $x'yz'$ of 1 literal, 2 literals and 3 literals respectively.

* The product of sum form is a form in which a boolean function contains OR terms called sum terms, of one or more literals each. The product denotes the ANDing of these terms. The example for pos form function is $F_2 = x(y' + z)(x' + y + z' + w)$

In this example we have three sum terms x , $y' + z$, $x' + y + z' + w$ of 1 literal, 2 literals and 4 literals respectively.

Conversion of SOP form to canonical SOP form:

To convert a boolean function of SOP form to Canonical SOP form the following steps are to be followed.

- 1) Find the missing literal of each product term if any.
- 2) AND each product term that has missing literal / literals with term/terms formed by ORing the literal and it's complement.
- 3) Expand the terms by applying distributive law and reorder the literals in the product terms.
- 4) Reduce the expression by omitting the repeated product terms if any.
ex: $ABC + ABC + ABC = ABC$

Example: Convert the function $f(A, B, C) = AB + BC + AC$ into Canonical - SOP form and Canonical POS form.

Sol) Given $f(A, B, C) = AB + BC + AC$

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

$$\left(\begin{array}{l} \because x(y+z) \\ = xy + xz \end{array} \right)$$

$$= ABC + ABC' + BCA + BCA' + ACB + ACB'$$

$$= ABC + ABC' + ABC + A'BC + ABC + AB'C$$

$$f(A, B, C) = ABC + ABC' + A'BC + AB'C$$

$$= m_7 + m_6 + m_3 + m_5$$

\therefore Canonical SOP form = $\sum m(3, 5, 6, 7) = ABC + ABC' + A'BC + AB'C$

Canonical POS form = $\prod M(0, 1, 2, 4) = (A+B+C)(A+B+C')(A+B'+C)(A'+B+C)$

NOTE: The canonical POS form function from a Canonical SOP function can be written simply by writing the missing decimal numbers from the Canonical SOP form function, and vice versa.

Canonical Pos form from the given pos form function:

The conversion of pos to Canonical pos can be accomplished by using the following steps.

- 1) Find the missing literal in each sum term if any.
- 2) OR each sum term that has missing literal / literals with the term / terms formed by ANDing the literal and it's complement.
- 3) Expand the terms by applying distributive law and reorder the literals in the sum terms.
- 4) Reduce the expression by omitting repeated sum terms if any.
for example $(A+B+C)(A+B+C)(A+B+C) = A+B+C$.

Example: Convert the pos form function $F(A,B,C) = (A+B)(B+C)(A+C)$ to Canonical pos form and Canonical sop form.

sol) Given $F(A,B,C) = (A+B)(B+C)(A+C)$

$$\begin{aligned} &= (A+B+Cc') (B+C+AA') (A+C+BB') \\ &\quad [\because A+Bc = (A+B)(A+C)] \\ &= (A+B+C) (A+B+c') (B+C+A) (B+C+A') (A+C+B) \\ &\quad (A+C+B') \\ &= (A+B+C) (A+B+c') (A+B+C) (A'+B+C) (A+B+C) \\ &\quad (A+B'+C) \\ &= (A+B+C) (A+B+c') (A'+B+C) (A+B'+C) \\ &= M_0 M_1 M_4 M_2 \\ &= \Pi M (0, 1, 2, 4) \end{aligned}$$

$$\therefore \text{Canonical pos form of } F(A,B,C) = \Pi M (0, 1, 2, 4) = (A+B+C) (A+B+c') (A'+B+C) (A+B'+C)$$

$$\text{Canonical sop form of } F(A,B,C) = \Sigma m (3, 5, 6, 7) = A'BC + AB'C + ABC' + ABC$$

Practice problems:

- i) Convert $F_1 = A + Bc' + ABD' + ABCD$ to Canonical SOP form and Canonical POS form
- ii) Convert $F_2 = A(A' + B)(A' + B + c')$ to Canonical POS form and Canonical SOP form
- iii) Convert $F_3 = B' + Bc' + A'B'c'$ to Canonical SOP form.

Other Logic operations :

There are 2^{2^n} possible functions for n binary variables. For two variables (i.e. $n=2$), the number of possible functions are $2^{2(2)} = 16$. Therefore we can have 16 different functions formed with two binary variables say x and y . Let the functions be represented as F_0, F_1, \dots, F_{15} . The truth table for these 16 functions of two variables is given below.

x	y	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

The 16 functions listed in the truth table can be expressed algebraically by means of Boolean expressions.

Each function can be expressed in terms of the Boolean operators, but some of the functions can not be expressed in terms of operators, there is no specific reason for that.

The 16 functions listed can be subdivided into three categories.

1) Two functions that produce a constant 0 (or) 1.

2) Four functions with unary operations: Complement and transfer.

3) Ten functions with binary operations that define eight different operations: AND, OR, NAND, NOR, Exclusive-OR, Equivalence, inhibition

and implication. The 16 boolean functions, their names and operators are listed below.

Boolean Functions	Operator Symbol	Name	Comments
$F_0 = 0$	-	Null	Binary Constant 0
$F_1 = xy$	$x \cdot y$	AND	x and y .
$F_2 = xy'$	x/y	Inhibition	x but not y
$F_3 = x$	-	Transfer	x
$F_4 = x'y$	y/x	Inhibition	y but not x
$F_5 = y$	-	Transfer	y
$F_6 = x'y + xy'$	$x \oplus y$	Exclusive-OR	x or y but not both.
$F_7 = x+y$	$x+y$	OR	x or y .
$F_8 = (x+y)'$	$x \downarrow y$	NOR	NOT OR
$F_9 = x'y' + xy$	$x \odot y$	Equivalence	x equals y .
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x+y'$	$x \subset y$	Implication	If y then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x'+y$	$x \supset y$	Implication	If x then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not AND
$F_{15} = 1$	-	Identity	Binary Constant 1.

NOTE: The boolean functions shown in the above table are the simplified forms of the canonical sop form of the function.

For example from the truth table $F_5 = x'y + xy$

$$= y(x' + x)$$

$$\therefore F_5 = y$$

$$\text{similarly } F_{11} = x'y' + xy' + xy = y'(x' + x) + xy = y' + xy = y' + x$$

$$\therefore F_{11} = x + y'$$

Similarly, all the remaining functions also can be derived.

Digital Logic gates:

(47)

AND Gate: An AND gate has two (or) more inputs but only one output. The output will be at logic 1 state only when each of its inputs is at logic 1 state. The output is at logic 0 state even if one of its inputs is at logic 0 state.

The logic symbol and the truth table of a two input AND gate is as shown in figure below.

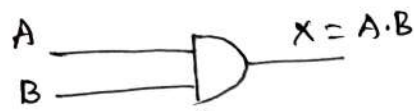


fig: Logic symbol of a two input AND gate

Truth table

Inputs		output
A	B	$X = A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate: An OR gate has two or more inputs but only one output. The output assumes logic 1 state, even if one of its inputs is at logic 1 state. The output is logic 0 only when each of the inputs is at logic 0 state.

The logic symbol and the truth table of a two input OR gate is as shown in figure below.

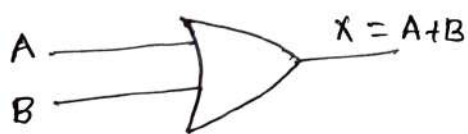


fig: Logic symbol for a two-input OR gate

Truth table

Inputs		output
A	B	$X = A + B$
0	0	0
0	1	1
1	0	1
1	1	1

NOT gate: A NOT gate is also called as an inverter, it has only one input and one output. The output of NOT gate is always the complement of its input. i.e the output of the NOT gate is at

logic 1 state when its input is in logic 0 state and it is at logic 0 state when its input is in logic 1 state.

The logic symbol and truth table of the inverter is as shown in below figure.

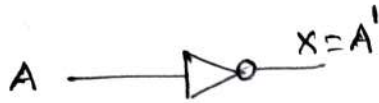


Fig: Logic symbol of a NOT gate.

Truth table

Input A	output $X = A'$
0	1
1	0

NOTE: The AND gate, OR gate and the NOT gate are called as the basic gates.

Universal Gates:

Though the logic circuits of any complexity can be realized by using only the three basic gates (AND, OR and NOT), there are two universal gates (NAND and NOR), each of which can also realize logic circuits single-handedly. Therefore the NAND and NOR gates are called universal building blocks.

That means the NAND gate (or) NOR gate can perform all the three basic logic functions i.e. AND, OR and NOT.

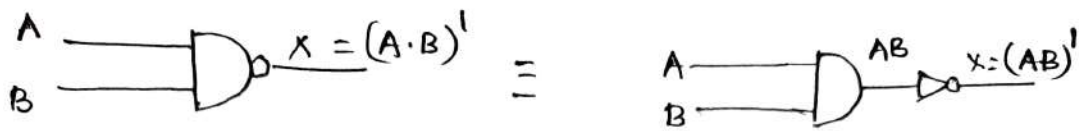
NAND Gate:

NAND means NOT + AND. i.e., the AND output is inverted.

The expression for the output of a NAND gate is written as

$X = (A \cdot B \cdot C \dots)'$. The output of the NAND gate is at logic 0 state when each of the input is at logic 1 state. For any other combination of inputs, the output is at logic 1 state.

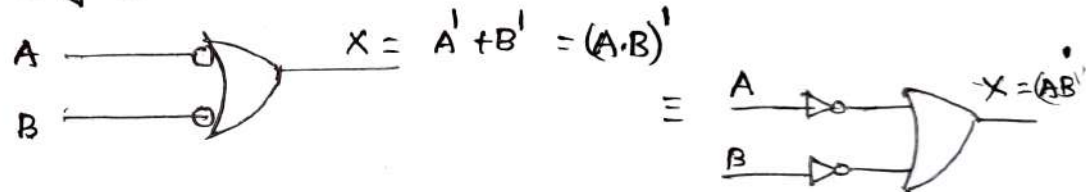
The logic symbol and the truth table of a two input NAND gate is as shown in below figure.



Truth Table

Inputs		output
A	B	X
0	0	1
0	1	1
1	0	1
1	1	0

**** NOTE :** The NAND Gate can be represented alternatively by using bubbled OR gate. The OR gate with inverted inputs is called as a bubbled OR gate.



Realisation of basic logic gates using NAND gate:

NOT Gate

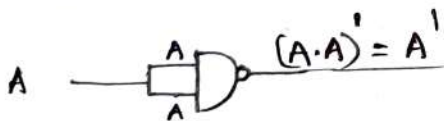


Fig: NAND gate as an inverter

AND gate

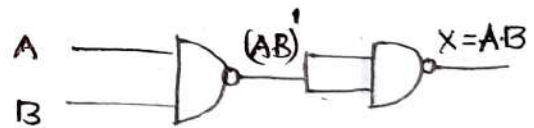


Fig: AND gate using NAND gates

OR Gate

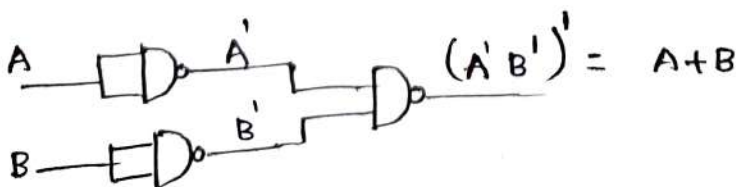


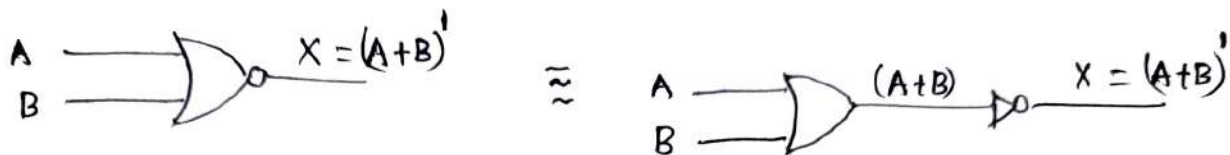
Fig: OR Gate using NAND gates.

NOR Gate:

NOR means NOT+OR i.e. the OR output is complemented. The expression for the output of NOR gate is $X = (A+B+C+\dots)'$

The output of NOR gate is at logic 1 state when each one of its inputs is at logic '0' level. For any other combination of inputs, the output is at logic '0' state.

The logic symbol and truth table for a two input NOR gate is as shown in below figures.

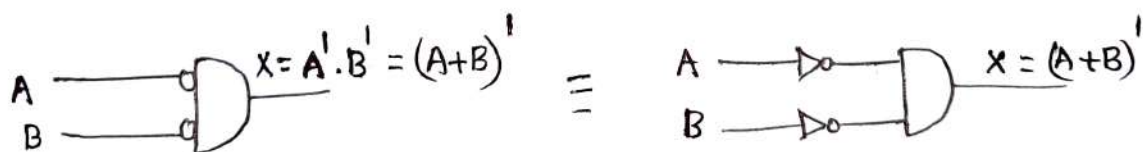


Truth Table

A	B	$X = (A+B)'$
0	0	1
0	1	0
1	0	0
1	1	0

**** NOTE:**

Alternatively the NOR Gate can be represented by using bubbled AND gate. In a bubbled AND gate each input is in complemented form.



Realization of basic logic gates using NOR gate:

NOT Gate:

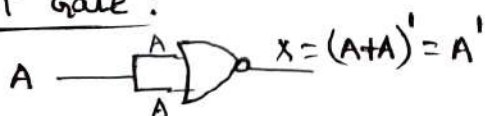


fig: NOR Gate as an inverter

OR Gate

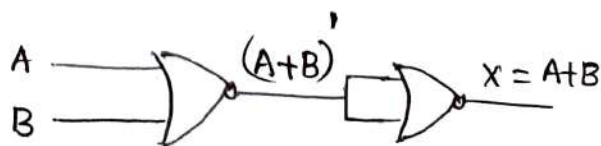


fig: OR Gate using NOR gates

AND Gate:

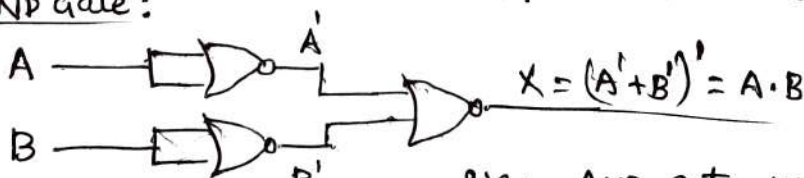


fig: AND gate using NOR gates

Exclusive OR Gate (or) Ex-OR Gate (or) XOR gate: This gate has

only two inputs. XOR gate output is at logic 1 state when one and only one of its two inputs is at logic '1' state. Otherwise the output is at logic '0' state. Since an XOR gate produces the output 1 only when the inputs are not equal, it is also called as an anti coincidence gate (or) inequality detector.

The operator symbol for XOR is \oplus . The symbol for an XOR gate and its truth table is as shown in below figure.

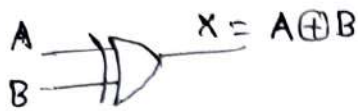


Fig: XOR gate symbol

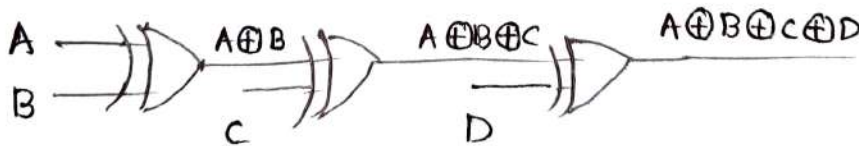
$$A \oplus B = A\bar{B} + \bar{A}B$$

Truth Table

A	B	$X = A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

NOTE: The XOR gate has only two inputs. To perform XOR operation among more than two variables firstly perform XOR operation between two variables using an XOR gate whose output is given as an input to another XOR gate as one of the input and the third variable as the second variable and so on. The same thing is applicable for XNOR.

$$A \oplus B \oplus C \oplus D$$

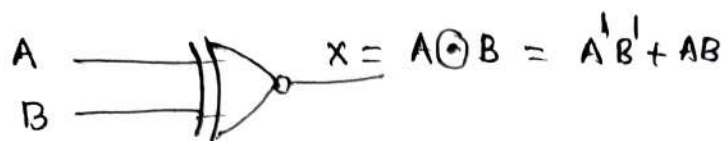


Exclusive NOR Gate (or) EX-NOR gate (or) XNOR gate:

An XNOR gate has two inputs and only one output. The output of an XNOR gate is at logic 1 state only when both the inputs are at either logic '0' state (or) at logic 1 state. Otherwise the output is at logic '0' state. Since an XNOR gate produces the output 1 only when the two inputs are equal, it is also called

as a coincidence gate (or) an equality detector.

The operator symbol for XNOR operation is \odot . The logic symbol and the truth table for an XNOR gate is as shown in below.



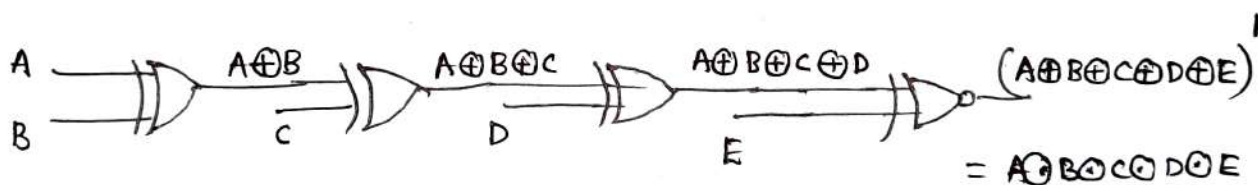
$$A \odot B = (A \oplus B)'$$

Truth table

A	B	$X = A \odot B$
0	0	1
0	1	0
1	0	0
1	1	1

NOTE: When there are multiple inputs to perform XNOR, we should perform the XOR operation for the first two variables using an XOR gate whose output is taken as one of the input for another XOR gate with third variable as second input. This is continued till the last but one variable. But for last variable we use XNOR gate.

$$A \odot B \odot C \odot D \odot E$$



INTEGRATED CIRCUITS (IC): An IC is also called as a chip.

An integrated circuit (IC) is a small silicon semiconductor crystal containing the electronic components for the digital gates. The various gates are interconnected inside the chip to form the required circuit.

The chip is mounted on a ceramic (or) plastic container and connections are welded to external pins to form the integrated circuit.

The traditional method of a combinational logic circuit design involves simplification of a logic function and realizing that function using logic gates. When the circuit is more complex, this traditional method becomes time taking and less reliable. Hence we prefer these integrated circuits.

Levels of Integration:

Digital IC's are often categorized according to their circuit complexity as measured by the number of logic gates in a single package. The IC's have the following levels of integration based on the number of logic gates.

- i) Small scale Integration (SSI): The IC's of small scale integration contain usually fewer than 10 gates and it is limited by the number of pins available in the IC. The inputs and outputs of the gates are connected directly to the pins in the package.
- ii) Medium scale Integration (MSI): The IC's of medium scale integration contain approximately 10 to 100 gates in a single package. They usually perform specific elementary digital operations such as decoders, adders or multiplexers.
- iii) Large scale Integration (LSI): The IC's of large scale integration have the gates between 100 to a few thousands in a single package. They include digital systems such as processors, memory chips, and programmable logic devices.
- iv) Very Large scale Integration (VLSI): The IC's of VLSI contain thousands of gates within a single package. They include advanced microprocessors, large memories, and larger programmable logic devices. Because of their small size and low cost, VLSI devices have brought a revolution in the computer system design technology.

Digital Logic Families:

Digital IC's are not only classified based on their complexity or logical operation, but also the circuit technology to which they belong to. The circuit technology is referred to as a digital logic family. Each logic family has its own basic electronic circuit upon which

more complex digital circuits are developed. The electronic circuit used in the construction of the basic circuit is usually used as the name of the technology. The most popular logic families of digital ICs are discussed below.

TTL (Transistor-Transistor Logic) : uses bipolar transistors as their major circuit element.

ECL (Emitter coupled Logic) : It uses bipolar transistors as their major circuit element.

MOS (Metal oxide Semiconductor) : uses unipolar MOSFETs as their major components.

CMOS (Complementary Metal oxide Semiconductor) : uses unipolar MOSFETs as their major components.

Because of the use of different principle components, their electrical behaviour are different. The most important parameters that are evaluated for each logic family are discussed below.

Fan out : It specifies the number of standard loads that the output of a typical gate can drive without impairing its normal operation. A standard load is usually defined as the amount of current needed by the input of another similar gate of the same family.

Power Dissipation : It is the power consumed by the gate that is supplied by the supply voltage.

Propagation Delay : It is the time interval between the application of an input pulse and the occurrence of the resulting output pulse from a gate.

Noise margin : It is the maximum external noise voltage being added to an input signal that does not cause an undesirable change in the circuit output.

Fan-in : It is the number of inputs that a logic gate has. The Fan-in for inverter is 1, 2-input NAND gate Fan-in is 2 etc.

①

UNIT-II

GATE LEVEL MINIMISATION

* We know that the boolean functions can be realized using logic gates. The total number of logic gates and literals can be reduced, if the boolean function is simplified. The simplification of a boolean function is required for reducing the complexity and cost of the designing of its logic circuit.

* During the process of simplification using boolean algebra one must know the boolean laws, rules, properties and theorems thoroughly. And also it is required to predict the successive steps to get the simplest expression.

* The map method gives us the systematic procedure to simplify the given boolean expression. It is also called as karnaugh map method (or) k-map method.

* The map method was first proposed by Veitch and modified by karnaugh. And hence map method is also called as Veitch diagram (or) karnaugh-map.

The map method (or) karnaugh-map (or) K-map method:

→ The K-map is a diagram made of square boxes. Each square box is called as a cell that represents either a minterm or a max term.

→ The simplified function produced using K-map is present in any one of the standard forms i.e either in product of sums (POS) form or in sum of products form.

→ The simplified function should have less number of terms and each term should have minimum number of literals.

→ A K-map contains 2^n cells for its n -variable boolean expression.

For example a 4-variable boolean expression contains $2^4 = 16$ cells.

2-Variable K-map:

In a 2-variable K-map there are $2^2 = 4$ cells. Each cell represents a minterm (or) a max term.

A two variable K-map with minterm representation and maxterm representation are as shown in below.

A \ B	B'	B
A'	0 $A'B'$	1 $A'B$
A	2 AB'	3 AB

$A=0 \Rightarrow A'$
 $A=1 \Rightarrow A$

fig: 2-variable K-map with minterm representation

A \ B	B	B'
A	0 $A+B$	1 $A+B'$
A'	2 $A'+B$	3 $A'+B'$

$A=0 \Rightarrow A$
 $A=1 \Rightarrow A'$

fig: 2-variable K-map with max-term representation

3-Variable K-map:

A 3-variable K-map consists of $2^3 = 8$ cells. Each cell represents a minterm or a max term.

Here the minterms (or) maxterms are arranged in graycode - sequence but not in the ordinary binary sequence.

The advantage of Graycode over normal binary sequence is that only one bit position is having a change between its present value to previous value (or) between its present value to its next value.

The three variable K-map with minterm and maxterm representations are given in below figures.

A \ BC	B'C'	B'C	BC	BC'
A'	0 $A'B'C'$	1 $A'B'C$	3 $A'BC$	2 $A'BC'$
A	4 ABC'	5 ABC	7 ABC	6 ABC'

fig: 3-variable K-map with minterm representation

A \ BC	B'C'	B'C	BC	BC'
A	0 $A+B+C$	1 $A+B+C'$	3 $A+B+C$	2 $A+B+C'$
A'	4 $A'+B+C$	5 $A'+B+C'$	7 $A'+B+C$	6 $A'+B+C'$

fig: 3-variable K-map with maxterm representation

(2)

4-variable K-map:

A 4-variable K-map contains $2^4 = 16$ cells. Each cell contains either minterm or maxterm.

A 4-variable K-map containing minterm representation and maxterm representation are as shown in below.

AB \ CD	C'D	CD	CD	C'D
	00	01	11	10
A'B' 00	A'B'C'D	A'B'C'D	A'B'CD	A'B'CD
A'B 01	A'BC'D	A'BC'D	A'BCD	A'BCD
AB 11	ABC'D	ABC'D	ABCD	ABCD
AB' 10	AB'C'D	AB'C'D	AB'CD	AB'CD

fig: 4-variable K-map with minterm representation.

AB \ CD	C+D	C+D'	C'+D	C'+D'
	00	01	11	10
A+B 00	A+B+C+D	A+B+C+D	A+B+C'+D	A+B+C'+D
A+B' 01	A+B'+C+D	A+B'+C+D	A+B'+C'+D	A+B'+C'+D
A+B 11	A+B+C+D	A+B+C+D	A+B+C'+D	A+B+C'+D
A+B' 10	A+B'+C+D	A+B'+C+D	A+B'+C'+D	A+B'+C'+D

fig: 4-variable K-map with maxterm representation.

The following terms are defined with respect to K-map simplification.

Pair: A group of two adjacent minterms (or) maxterms is called as a pair. A pair eliminates one variable from the resultant term.

Quad: A group of four adjacent minterms (or) maxterms is called as a quad. A quad eliminates two variables from its resultant term.

Octet: A group of eight adjacent minterms (or) maxterms is called as an octet. An octet eliminates three variables from its resultant term.

* NOTE

In a K-map any two cells are said to be adjacent, if their binary equivalent values are having only a one bit position change.

Ex: Though cell number '0' and cell number '2' are not looking like adjacent, they are adjacent. Because the binary equivalent for '0' and '2' are having a change, only in one bit position.
 $0 = 0000$, $2 = 0010$. only 2nd LSB bit is getting changed.

Minimal SOP form: (or) simplified SOP form:

To get the simplified expression in SOP form we have to follow the steps below.

- 1) Plot the K-map and place 1's in the cells corresponding to the given minterms in the given boolean expression.
- 2) Check for the 1's and encircle those 1's which are not adjacent to any other 1's. These are called as isolated 1's.
- 3) Check for the 1's which are adjacent to only one '1' and encircle them as a pair.
- 4) Check for the 1's that are having 4-adjacent 1's (quad) and 8 adjacent 1's (octet), even if some of the 1's among them are already encircled. While doing this make sure that there are less number of groups.
- 5) Form the simplified boolean function by summing all the product terms of all groups.

Problems:

using K-map.

Simplify the following two variable boolean functions in SOP.

1) $f(x, y) = \sum m(0, 1, 3)$

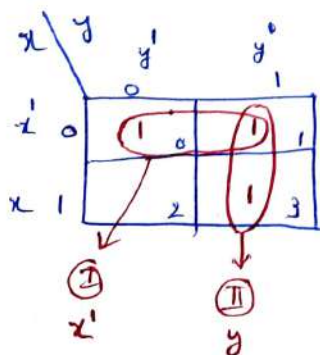
2) $f(A, B) = \sum m(1, 2, 3)$

3) $f(A, B) = \sum m(1, 3)$

4) $f(x, y) = \sum m(0, 1, 2, 3)$

1. Sol) Given boolean function

$$f(x, y) = \sum m(0, 1, 3)$$

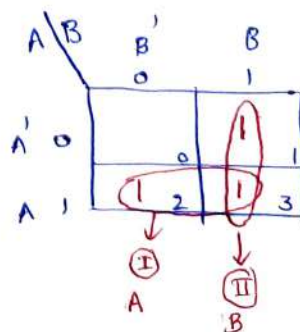


The simplified SOP form

function is obtained by summing each term of each group $f(x, y) = x' + y$

2) Sol) Given boolean function

$$f(A, B) = \sum m(1, 2, 3)$$



$$\therefore f(A, B) = A + B$$

(3)

3) Sol

Given boolean function

$$f(A, B) = \sum m(1, 3)$$

A \ B	B'	B
A'	0	1
A	2	3

(I) B

$$\therefore f(A, B) = B$$

4) Given boolean function

$$f(x, y) = \sum m(0, 1, 2, 3)$$

x \ y	y'	y
x'	0	1
x	2	3

(I) 1

In a K-map, if every cell covers a minterm for the given boolean function, the function will be equal to unity. i.e. $f(x, y) = 1$

Q) Simplify the following boolean functions in Sop form using K-map

$$1) F(x, y, z) = \sum m(2, 3, 4, 5)$$

$$2) F(x, y, z) = \sum m(0, 2, 4, 5)$$

$$3) F(A, B, C) = \sum m(0, 1, 2, 3, 4, 5, 6, 7)$$

$$4) F(x, y, z) = \sum m(1, 3, 5, 7)$$

1) Sol) Given boolean function

$$F(x, y, z) = \sum m(2, 3, 4, 5)$$

x \ yz	y'z'	y'z	yz	yz'
x'	0	1	2	3
x	4	5	6	7

(I) $x'y$
(II) xy'

$$\therefore F(x, y, z) = x'y + xy'$$

2) Sol) Given boolean function

$$F(x, y, z) = \sum m(0, 2, 4, 5)$$

x \ yz	y'z'	y'z	yz	yz'
x'	0	1	2	3
x	4	5	6	7

(I) $x'z'$
(II) xy'

$$\therefore F(x, y, z) = \sum m(0, 2, 4, 5) = x'z' + xy'$$

3) Given

$$F(A, B, C) = \sum m(0, 1, 2, 3, 4, 5, 6, 7)$$

A \ BC	B'C'	B'C	BC	BC'
A'	0	1	2	3
A	4	5	6	7

(I) 1

$$F(A, B, C) = 1$$

$$4) \text{ Given } F(x, y, z) = \sum m(1, 3, 5, 7)$$

Sol

x \ yz	y'z'	y'z	yz	yz'
x'	0	1	2	3
x	4	5	6	7

(I) xy
(II) z

$$\therefore f(x, y, z) = xy + z$$

NOTE:

Irrespective of number of variables, if a K-map contains all 1's the simplified function value is unity.

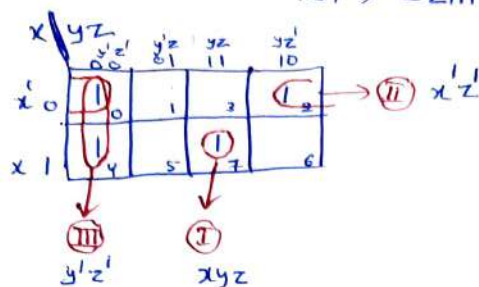
Simplify the following boolean functions into SOP form using K-map

1) $F(x, y, z) = \sum m(0, 2, 4, 7)$ 2) $F(x, y, z) = \sum m(0, 2, 3, 6, 7)$

3) $Y(A, B, C) = \prod M(0, 2, 4, 7)$ 4) $F(x, y, z) = \prod M(3, 5)$

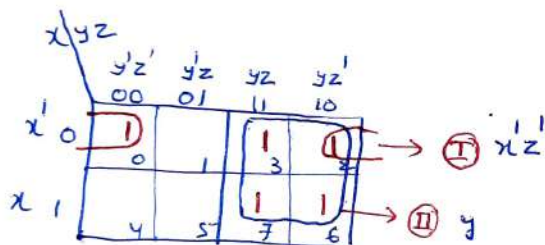
5) $F(A, B, C) = A'B + A'C + B'C + ABC \Leftrightarrow Y(A, B, C) = \sum m(1, 2, 4, 5, 6, 7)$

1) Sol Given boolean function $F(x, y, z) = \sum m(0, 2, 4, 7)$



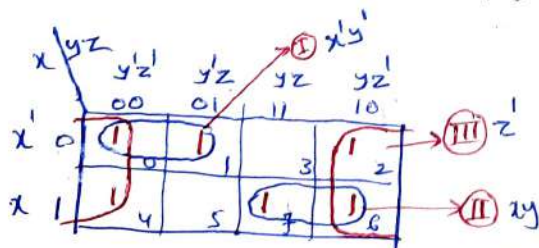
\therefore simplified form of $F(x, y, z)$ in SOP = $x'yz' + x'z' + xyz$

2) Sol $F(x, y, z) = \sum m(0, 2, 3, 6, 7)$



\therefore simplified form of $F(x, y, z)$ in SOP form $F(x, y, z) = y + x'z'$

4) Sol Given $F(x, y, z) = \prod M(3, 5) = \sum m(0, 1, 2, 4, 6, 7)$

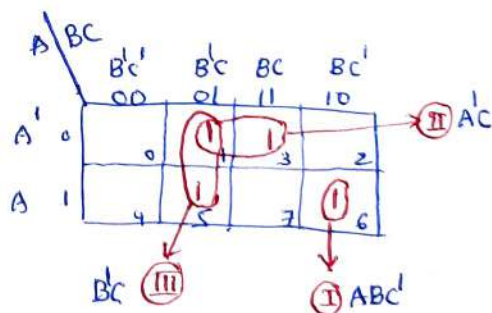


$\therefore F(x, y, z) = x'y' + xy + z'$

3) Sol $Y(A, B, C) = \prod M(0, 2, 4, 7)$

To simplify a boolean function into SOP form, it should be in canonical minterm form. So convert the given $Y(A, B, C)$ which is present in canonical maxterm form into canonical minterm form.

$\therefore Y(A, B, C) = \sum m(1, 3, 5, 6)$



$\therefore Y(A, B, C) = ABC' + A'C + BC'$

$F(A, B, C) = AB(C+C') + A'C(B+B') + B'C(A+A') + ABC$

$\Rightarrow F(A, B, C) = ABC + A'BC' + A'BC + A'B'C + ABC + A'B'C + ABC$

$= A'BC + A'BC' + A'BC + A'B'C + ABC$

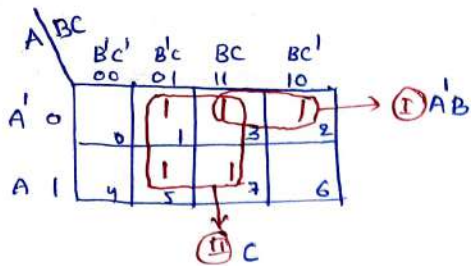
$= m_3 + m_2 + m_1 + m_5 + m_7$

$= \sum m(1, 2, 3, 5, 7)$

5) Sol Given $F(A, B, C) = A'B + A'C + B'C + ABC$

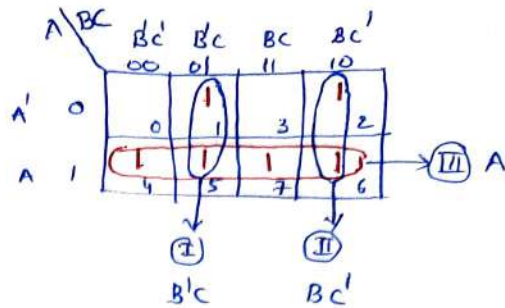
To simplify any function, it should be in canonical form. but the given function is not in canonical form. So convert the given function $F(A, B, C)$ into minterm canonical form.

$$\therefore F(A,B,C) = \sum m(1,2,3,5,7)$$



$$\therefore \text{Simplified SOP form of } F(A,B,C) \text{ is } F(A,B,C) = A'B + C$$

$$\text{Q4) Given } Y(A,B,C) = \sum m(1,2,4,5,6,7)$$



$$\therefore \text{Simplified SOP form of } Y(A,B,C) \text{ is } Y(A,B,C) = B'C + BC' + A$$

- H.W. { simplify the following boolean functions into SOP using k-map.
- 1) $F(A,B,C) = A'B + A'C + ABC$
 - 2) $F(X,Y,Z) = \sum m(0,2,3,4,6,7)$
 - 3) $Y(A,B,C) = \sum m(2,3,4,5)$
 - 4) $F(A,B,C) = \sum m(0,2,4,5,7)$

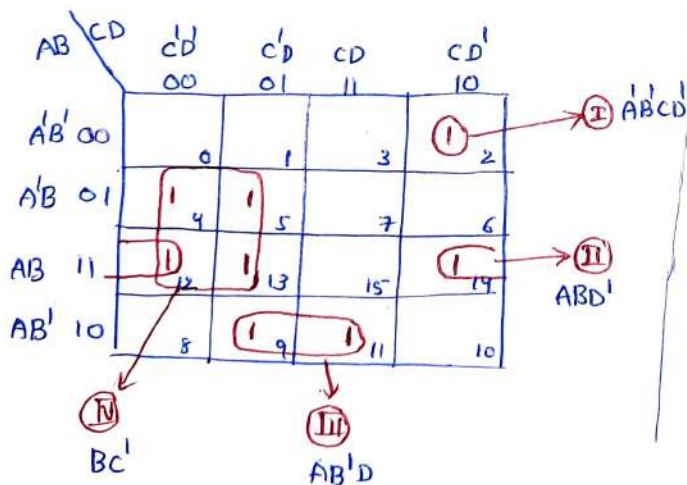
simplify the following boolean functions into SOP using k-map.

$$1) Y(A,B,C,D) = \sum m(2,4,5,9,11,12,13,14) \quad 2) F(A,B,C,D) = \sum m(0,1,3,7,11,13,14,15)$$

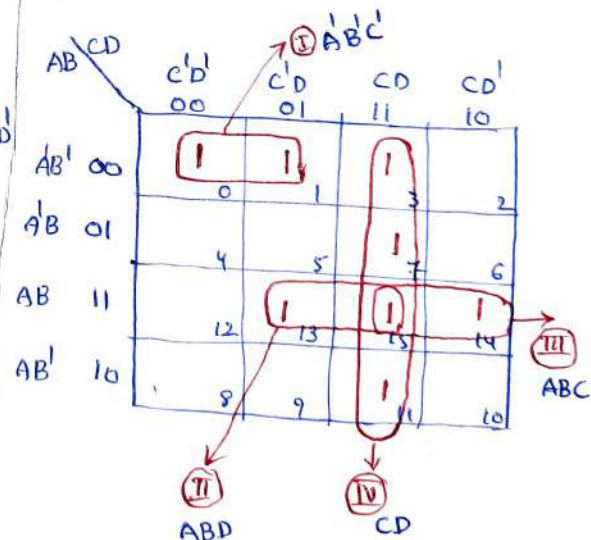
$$3) F(W,X,Y,Z) = \sum m(2,3,6,10,12,13,14,15) \quad 4) F(A,B,C,D) = \sum m(3,4,5,7,9,13,14,15)$$

$$5) F = \sum m(1,2,4,6,7,11,12,14) \quad 6) F(W,X,Y,Z) = \sum m(2,3,7,9,12,13,14,15)$$

$$\text{Sol) Given } Y(A,B,C,D) = \sum m(2,4,5,9,11,12,13,14) \quad \text{Sol) Given } F(A,B,C,D) = \sum m(0,1,3,7,11,13,14,15)$$



$$\therefore Y(A,B,C,D) = A'B'CD + ABD' + AB'D + BC'$$

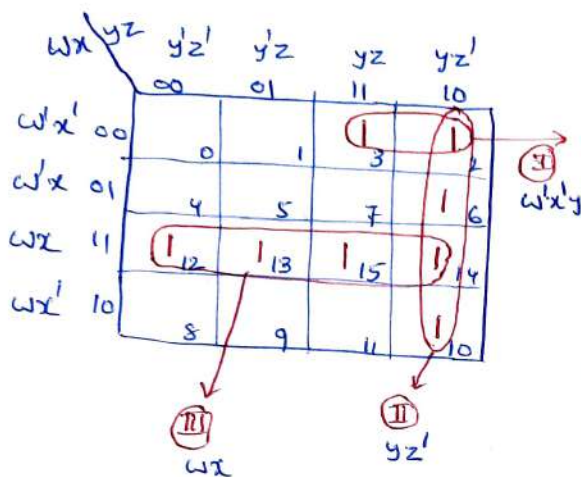


$$\therefore F(A,B,C,D) = A'B'C' + ABD + ABC + CD$$

NOTE: Sometimes the given boolean function may not be having variables. In that case the desired alphabets can be taken as variables but the number of variables is to be decided based on the highest minterm decimal equivalent. For example $Y = \sum m(0,1,4,7,9,11,13)$. In this

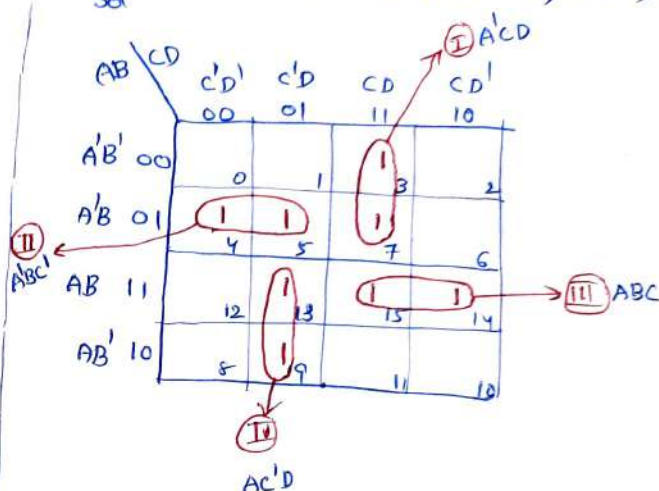
example the highest minterm decimal equivalent is 13, which can be written as 1101 in it's binary. Here 1101 has 4 digits of binary so we can take 4 alphabets as we desire say w, x, y, z (or) A, B, C, D. But, if the question it-self contains variables we have to stick on to the given variables only.

3) Sol) Given $F(w, x, y, z) = \sum m(2, 3, 6, 10, 12, 13, 14, 15)$



$$\therefore F(w, x, y, z) = w'x'y + wx + yz'$$

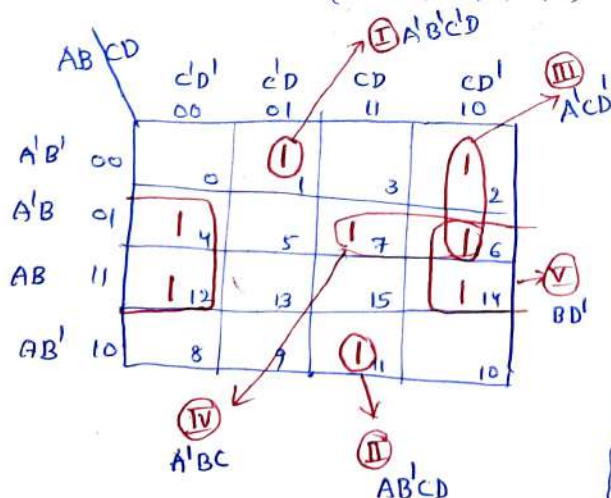
4) Sol) Given $F(A, B, C, D) = \sum m(3, 4, 5, 7, 9, 13, 14, 15)$



$$\therefore F(A, B, C, D) = A'CD + A'BC' + ABC + AC'D$$

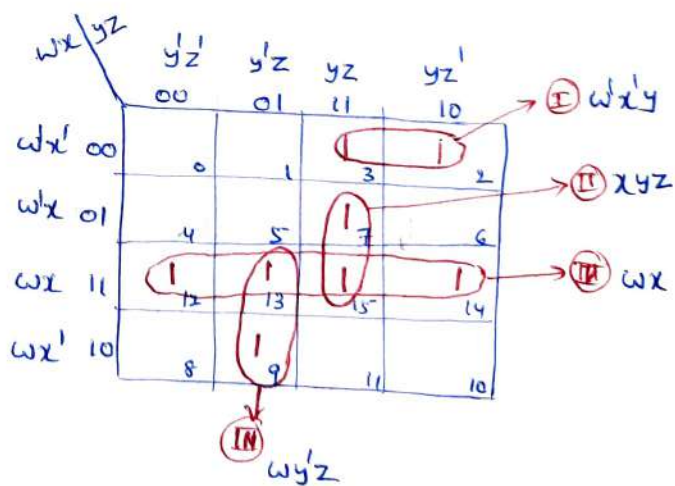
5) Sol) Given function $F = \sum m(1, 2, 4, 6, 7, 11, 12, 14)$
Let the variables be A, B, C, D.

$$\therefore F(A, B, C, D) = \sum m(1, 2, 4, 6, 7, 11, 12, 14)$$



$$\therefore F(A, B, C, D) = A'B'C'D + A'BCD + A'CD + A'BC + BD'$$

6) Sol) Given $F(w, x, y, z) = \sum m(2, 3, 7, 9, 12, 13, 14, 15)$



$$\therefore F(w, x, y, z) = w'x'y + xyz + wx + wy'z$$

H.W) Simplify the following boolean functions into SOP using k-map

1) $Y(A, B, C, D) = A'B'CD' + A'BC'D' + A'BC'D + ABC'D' + ABC'D + AB'C'D$

2) $F(w, x, y, z) = \sum m(0, 2, 3, 7, 8, 9, 10, 11)$

Prime implicants & essential prime implicants:

(5)

A Prime implicant is a product term that is obtained by combining maximum number of possible adjacent minterms as a group in the K-map.

If a minterm is covered only one time in a prime implicant then that prime implicant is said to be an essential prime implicant.

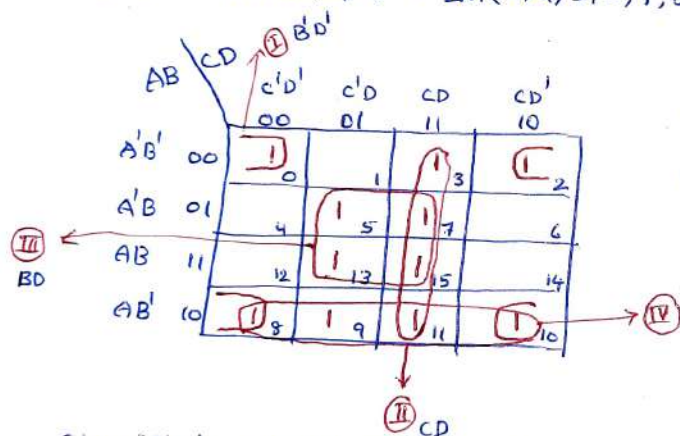
The essential prime implicants are formed by looking at each cell that is marked with '1' and checking the number of prime implicants that cover it. If only one prime implicant covers a particular cell that prime implicant is essential otherwise it is non essential.

Ex: 1) Find the simplified expression for the following boolean function first by finding essential prime implicants using K-map.

$$f(A, B, C, D) = \sum m(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

Sol)

$$\text{Given } f(A, B, C, D) = \sum m(0, 2, 3, 5, 7, 8, 9, 10, 11, 13, 15)$$

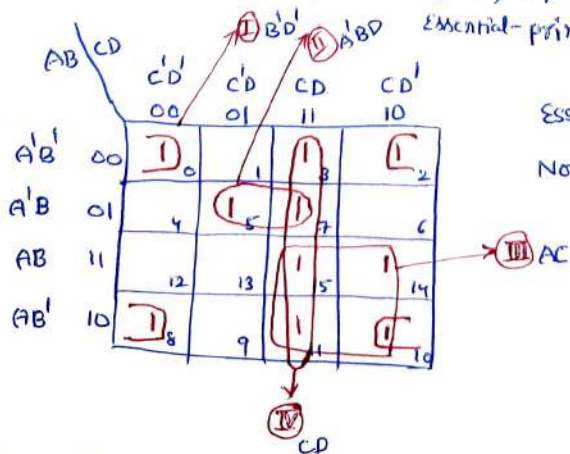


Essential prime implicants - $B'D'$, BD

Non essential prime implicants - AB' , CD

Simplified expression for $f(A, B, C, D) = B'D' + BD + AB' + CD$.

2) $f(A, B, C, D) = \sum m(0, 2, 3, 5, 7, 8, 10, 11, 14, 15)$ find simplified sop by finding essential prime implicants using K-map?



Essential prime implicants = $B'D'$, $A'BD$, AC

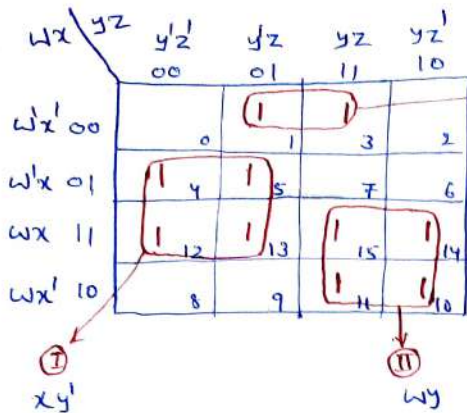
Non essential prime implicants = CD

\therefore simplified expression for $f(A, B, C, D) = B'D' + A'BD + AC + CD$

3)

$F(W, X, Y, Z) = \sum m(1, 3, 4, 5, 10, 11, 12, 13, 14, 15)$ Find Simplified SOP form function by finding essential prime implicants

Sol)



Essential prime implicants = $XY', WY, W'X'Z$

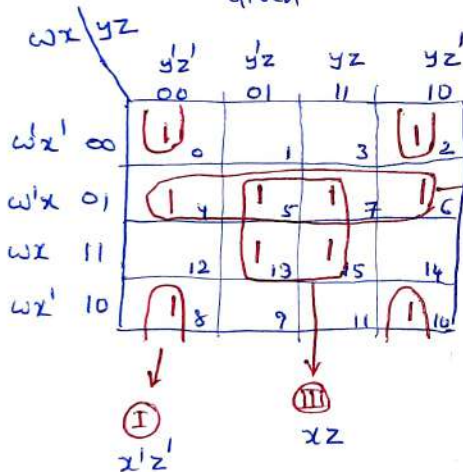
There is no non essential prime implicant in this k-map.

$$\therefore F(W, X, Y, Z) = XY' + WY + W'X'Z$$

4) $F(W, X, Y, Z) = \sum m(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$ find the simplified form of SOP by finding essential prime implicants.

Sol)

Given $F(W, X, Y, Z) = \sum m(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$



Essential prime implicants = $X'Z', XZ$

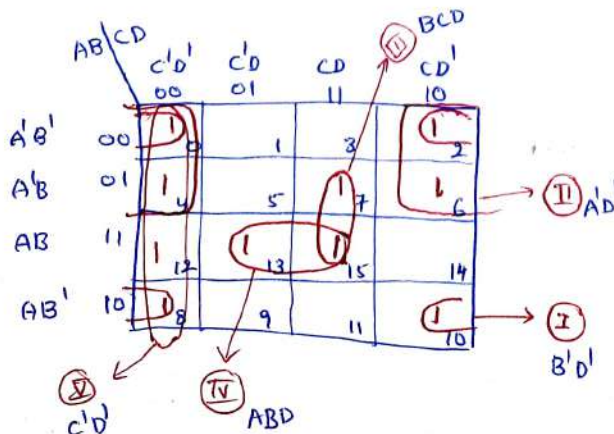
Non Essential prime implicants = $W'Z$

$$\therefore \text{Simplified SOP form function } F(W, X, Y, Z) = X'Z' + XZ + W'X$$

5) $F(A, B, C, D) = \sum m(0, 2, 4, 6, 7, 8, 10, 12, 13, 15)$ find the simplified form of SOP by finding essential prime implicants.

Sol)

Given $F(A, B, C, D) = \sum m(0, 2, 4, 6, 7, 8, 10, 12, 13, 15)$



Essential prime implicants = $B'D', A'D', C'D'$

Non essential prime implicants = BCD, ABD

$$\therefore f(A, B, C, D) = B'D' + A'D' + C'D' + BCD + ABD$$

(6)

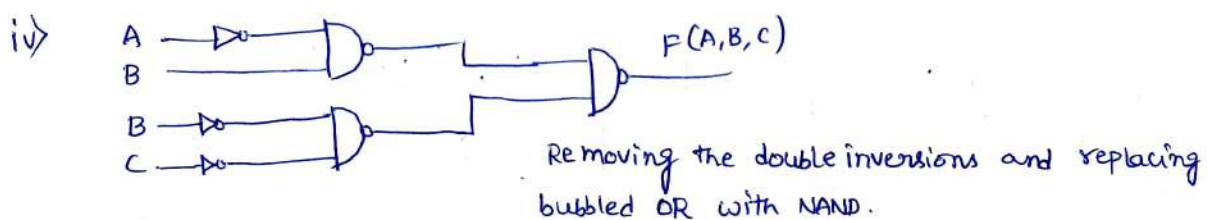
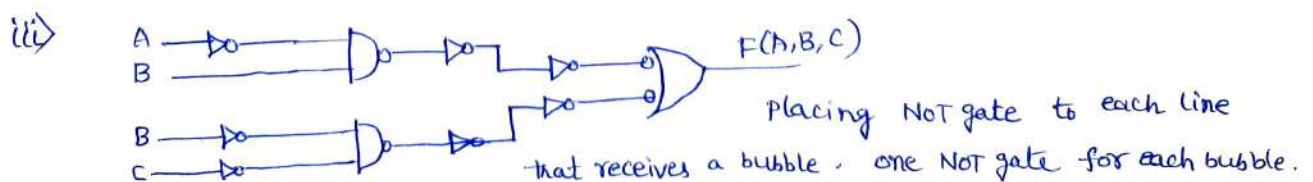
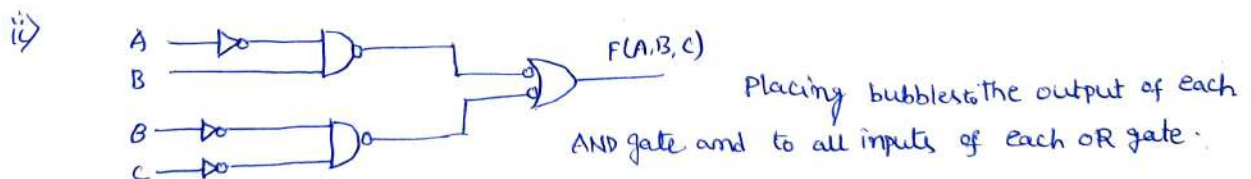
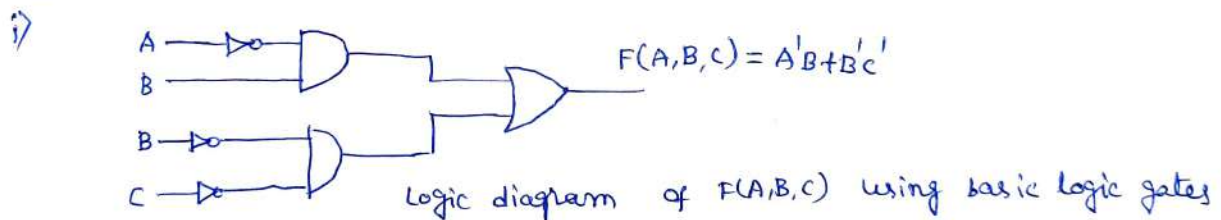
NAND & NOR Implementation (or) NAND & NOR Realization:

- 1> Draw the Logic diagram using basic logic gates (AND, OR, NOT)
- 2> If NAND Logic is being implemented, add bubbles to the output of the AND gates and to the inputs of OR gates.
- 3> If NOR Logic is being implemented, add bubbles to the output of the OR gates and to the inputs of the AND gates.
- 4> Add an inverter (not gate) to each line that receives a bubble in step 2 or step 3.
- 5> Eliminate double inversions and replace bubbled AND by NOR, bubbled OR by NAND
- 6> Replace single inverter with NAND inverter (in NAND realization) / NOR inverter (in NOR realization).

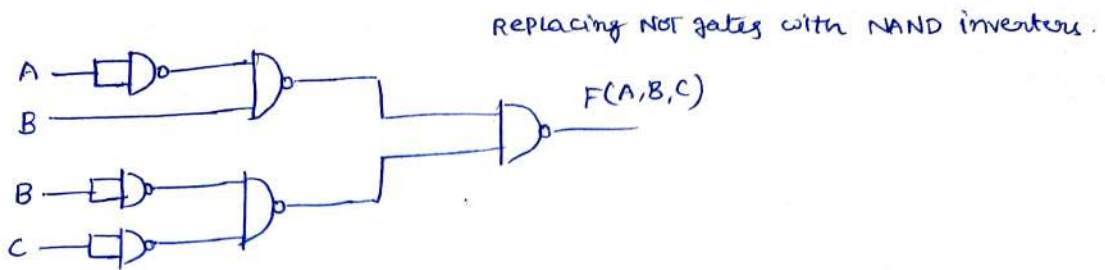
Ex: Implement $F(A,B,C) = A'B + B'C'$ using NAND, NOR logic gates.

sol> Given $F(A,B,C) = A'B + B'C'$

NAND Realization:

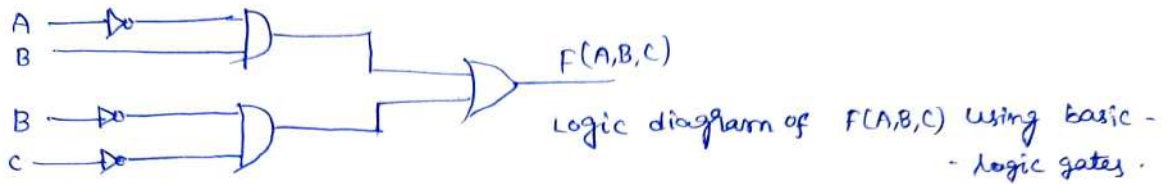


v)

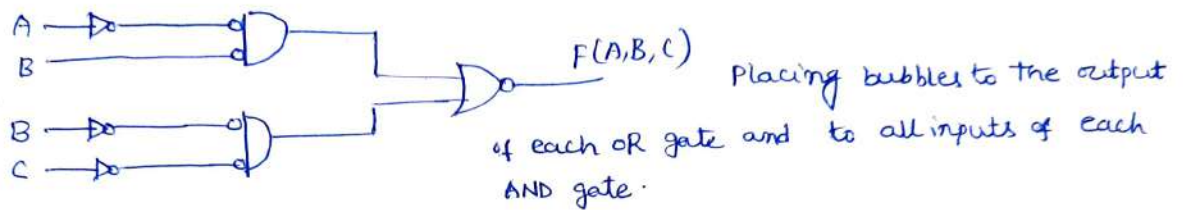


NOR Realization:

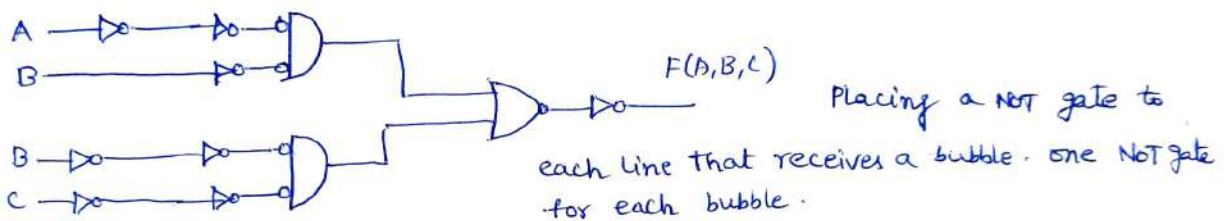
i)



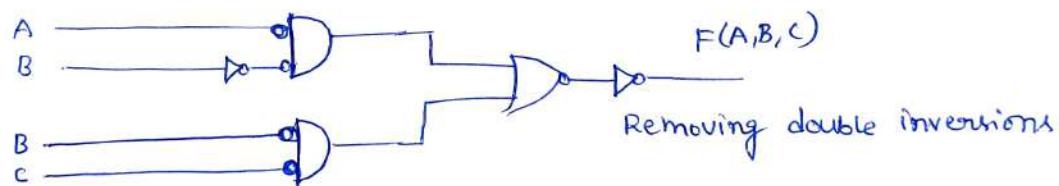
ii)



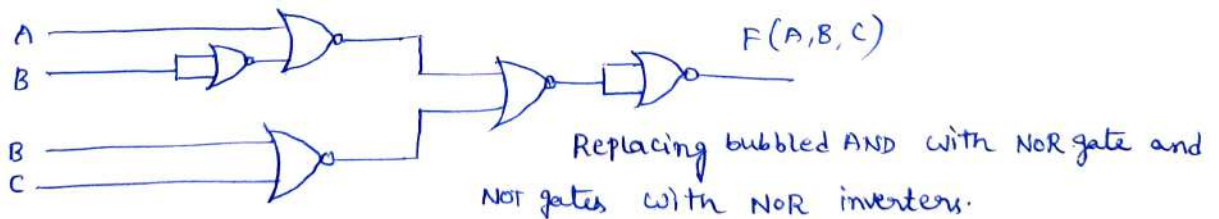
iii)



iv)



v)

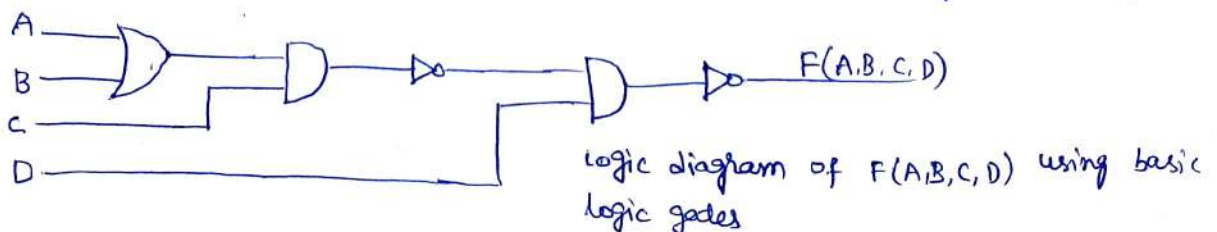


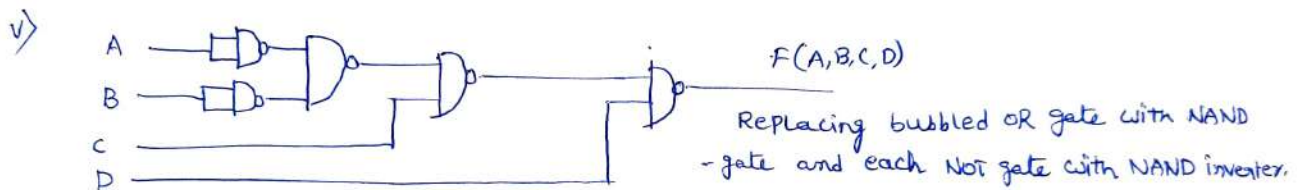
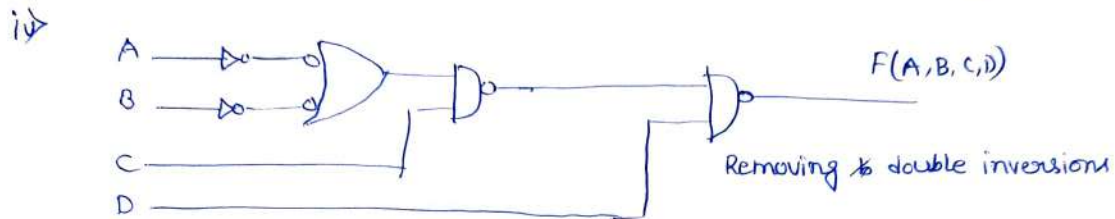
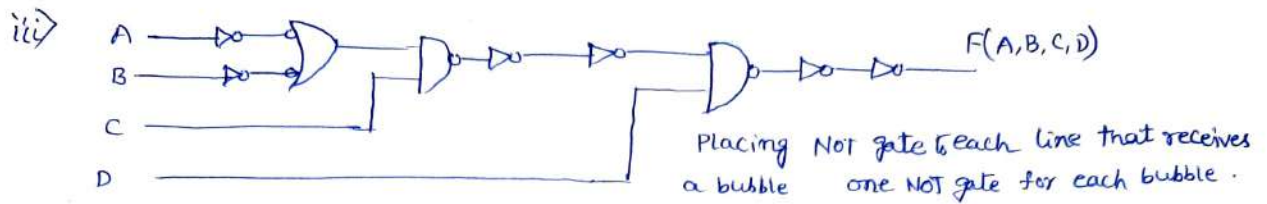
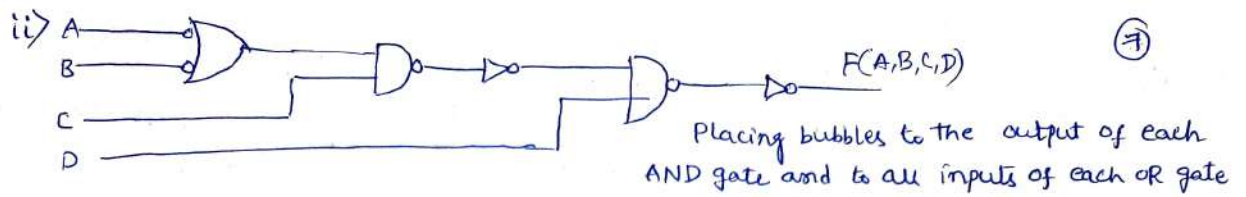
2)

$F = \overline{(A+B)C} \cdot D$ implement using NAND, NOR logic gates.

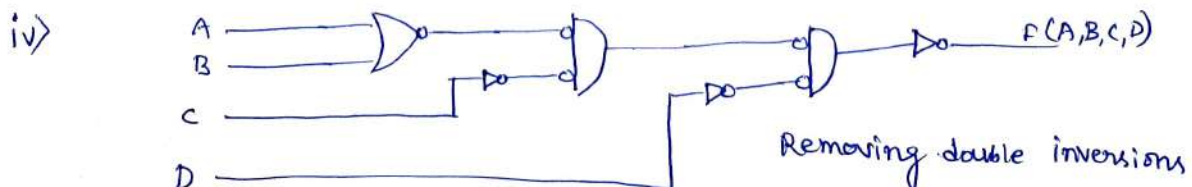
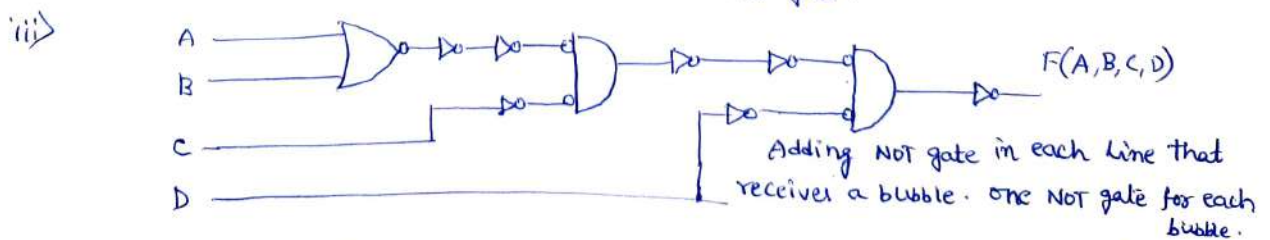
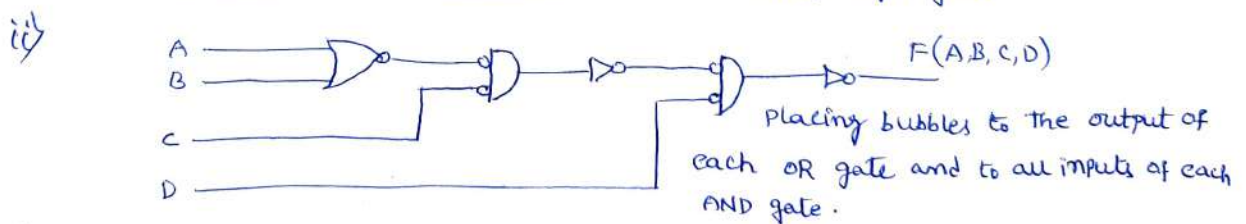
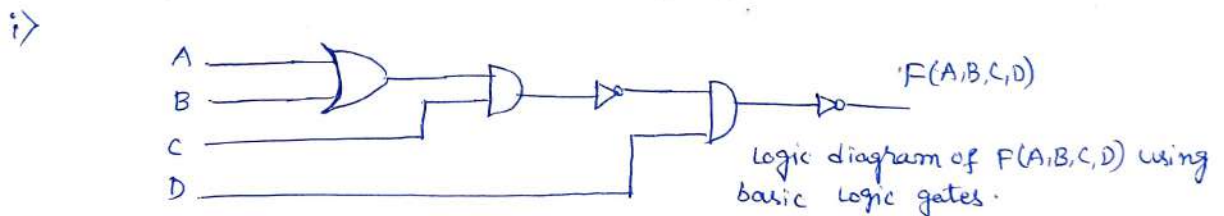
Set) using NAND Logic: Step 1 Draw the logic diagram using basic gates

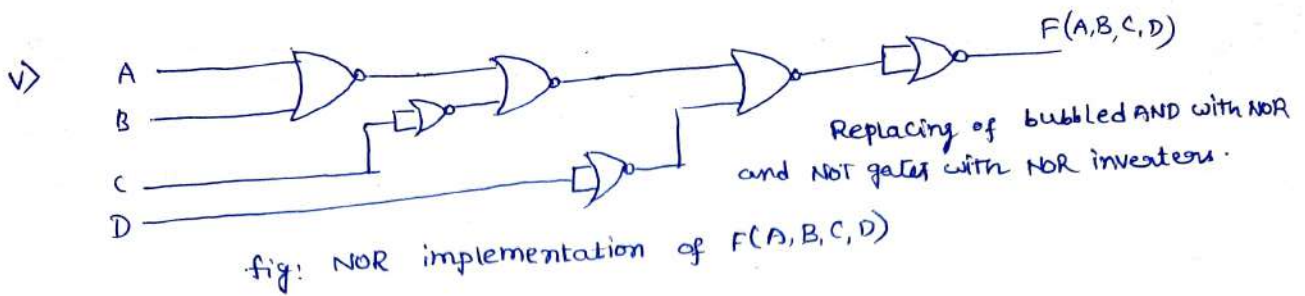
i)





NOR logic implementation for $F = ((A+B) \cdot C \cdot D)$

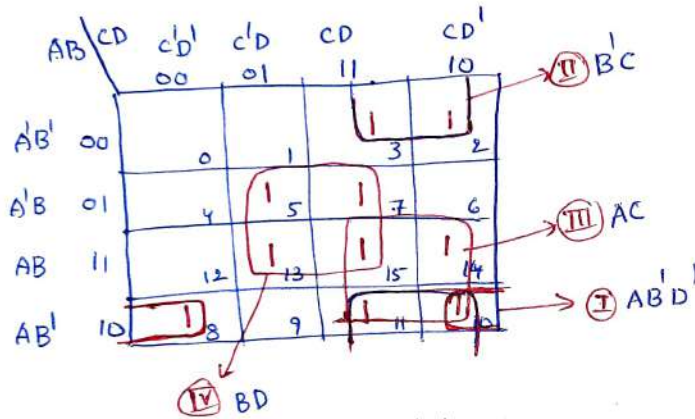




H.w Implement $F(A,B) = A \oplus B$ using NAND, NOR (Hint: $A \oplus B = AB' + A'B$)

Problem Simplify $f(A,B,C,D) = \sum m(2,3,5,7,8,10,11,13,14,15)$ in Sop form using K-map and implement with NAND gates.

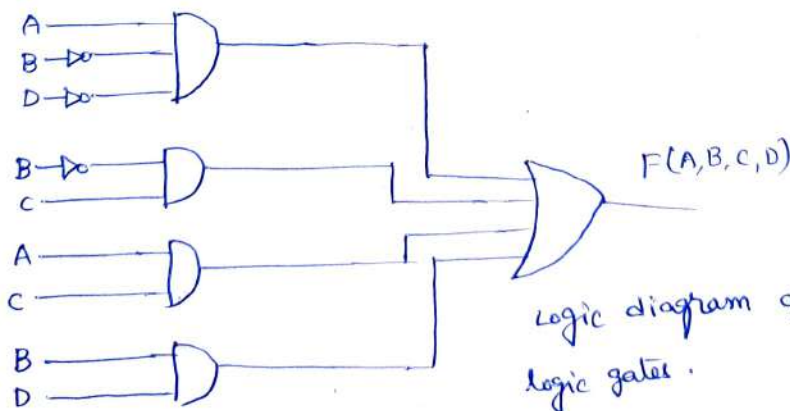
sol> Given $f(A,B,C,D) = \sum m(2,3,5,7,8,10,11,13,14,15)$



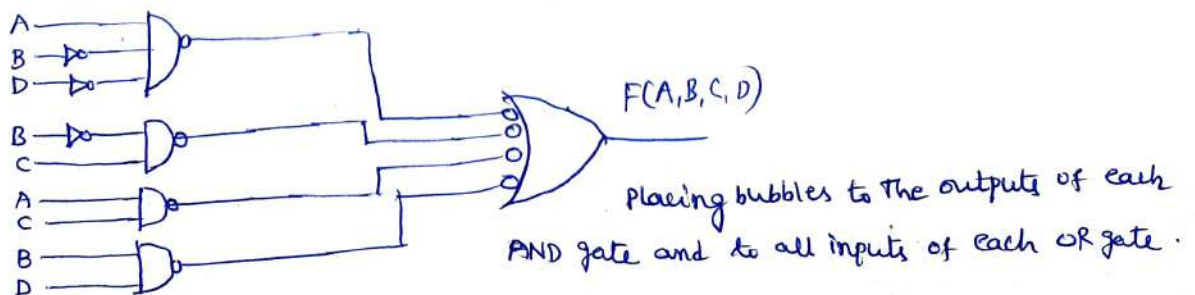
$$\therefore f(A,B,C,D) = AB'D' + B'C + AC + BD$$

Implementation of $F(A,B,C,D)$ using NAND gates.

i)

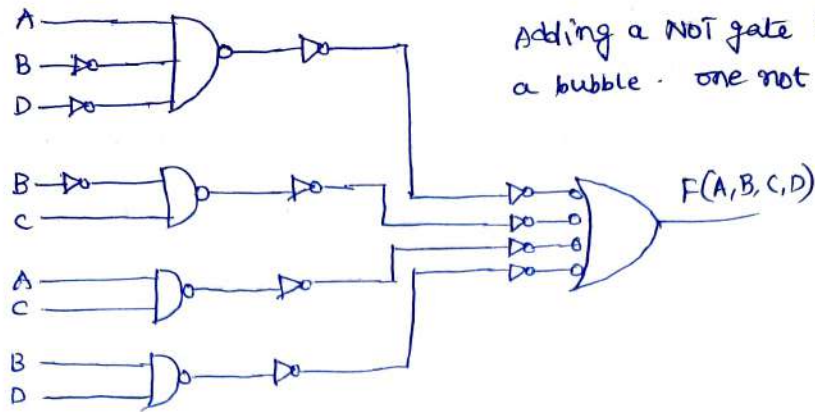


ii)

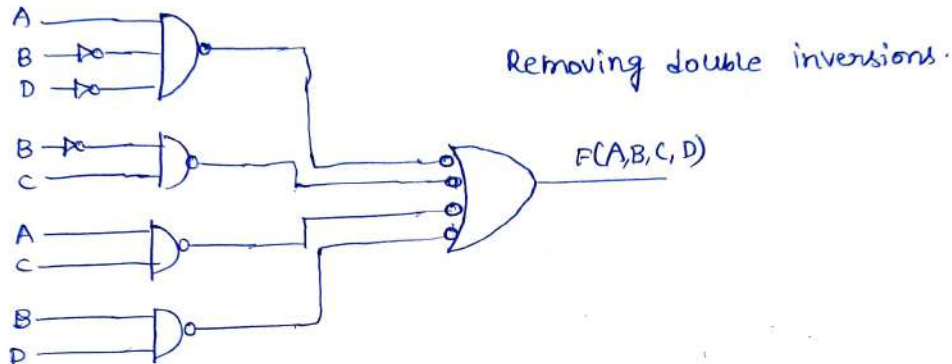


(8)

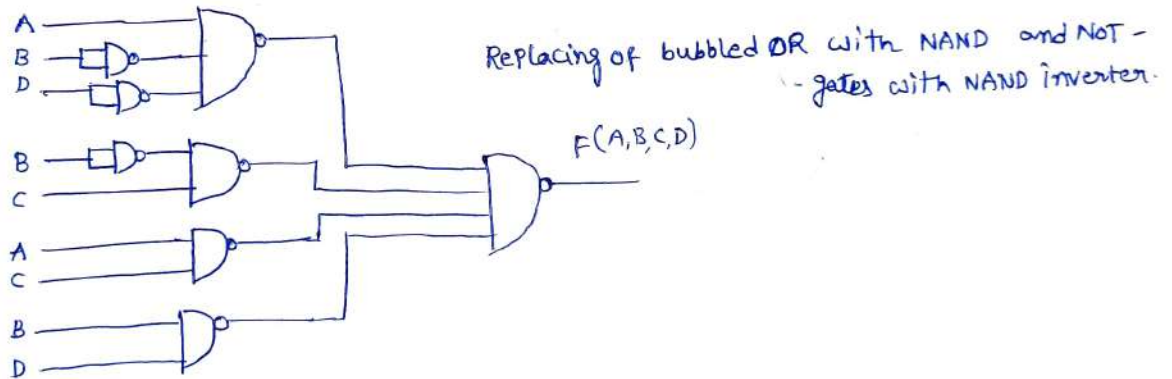
iii)



iv)



v)



(H.W)

Implement NAND logic for the simplified SOP form of the following functions i) $F(A, B, C, D) = \Sigma m(5, 9, 11, 12, 13, 14, 15)$. Use K-map method.

ii) $Y(W, X, Y, Z) = \Sigma m(0, 1, 3, 5, 6, 7, 10, 14, 15)$ using K-map method.

Minimal POS form (or) Simplified POS form:

To get the simplified expression in POS form we have to follow the steps given below.

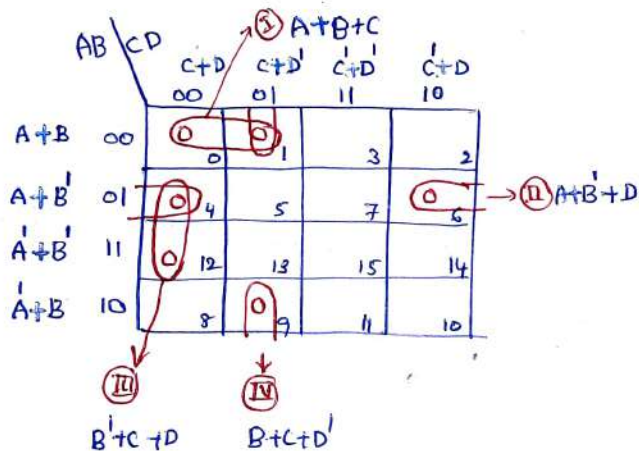
1) plot the K-map and place 0's in the place of maxterms that are given in the given boolean expression.

2) check for the 0's and encircle those 0's that are not adjacent to any other 0's. These are called as isolated 0's.

- 3> Check for the 0's which are adjacent to only one '0' and encircle them as a pair.
- 4> Check for quads (4 adjacent 0's) and octets (8 adjacent 0's) even if some of the 0's among them are already encircled. While doing this make sure that there are less number of groups.
- 5> Form the simplified boolean function by making the product of all sum terms of all groups.

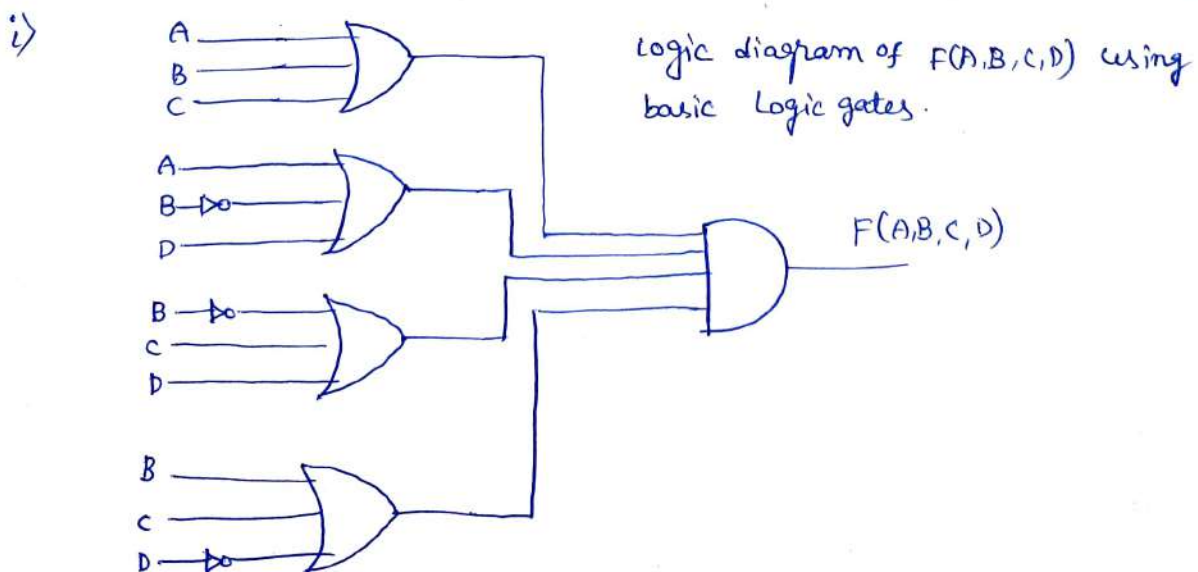
Problem 1) $F(A, B, C, D) = \Pi M(0, 1, 4, 6, 9, 12)$ simplify this function into POS form using K-map and implement using NOR gates.

Sol) Given $F(A, B, C, D) = \Pi M(0, 1, 4, 6, 9, 12)$

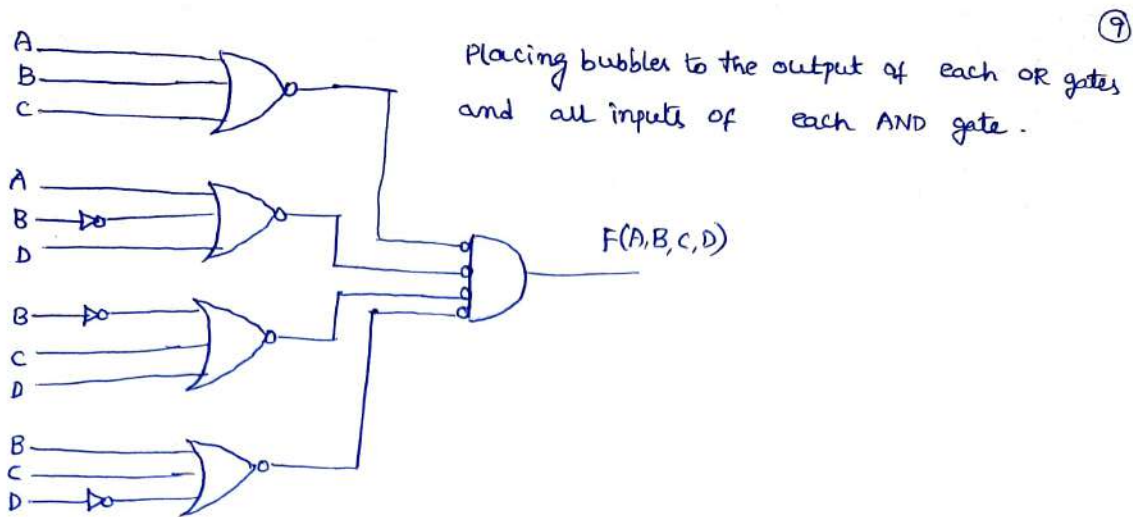


\therefore Simplified expression in POS form = $(A+B+C) \cdot (A+B'+D) \cdot (B'+C+D) \cdot (B+C+D')$

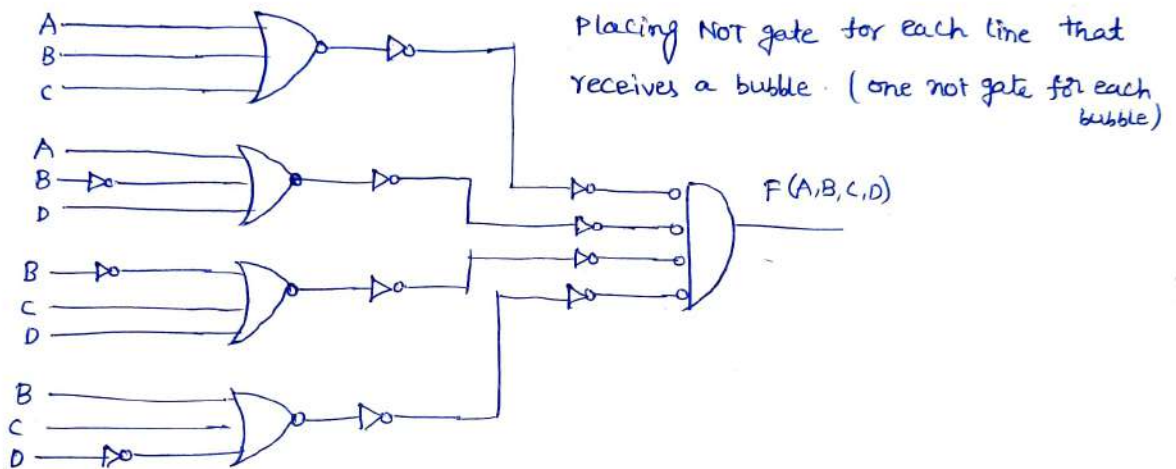
Implementation of the above simplified POS form of $F(A, B, C, D)$ in NOR gates :



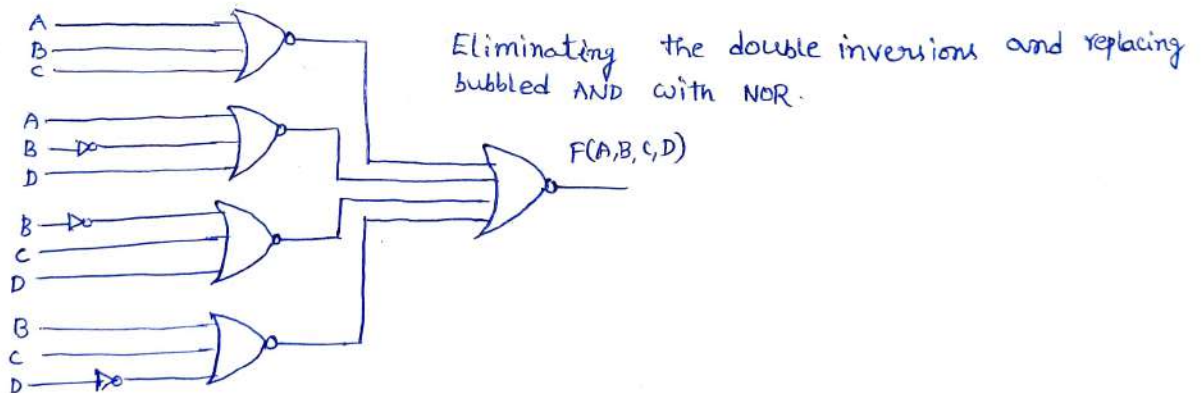
ii)



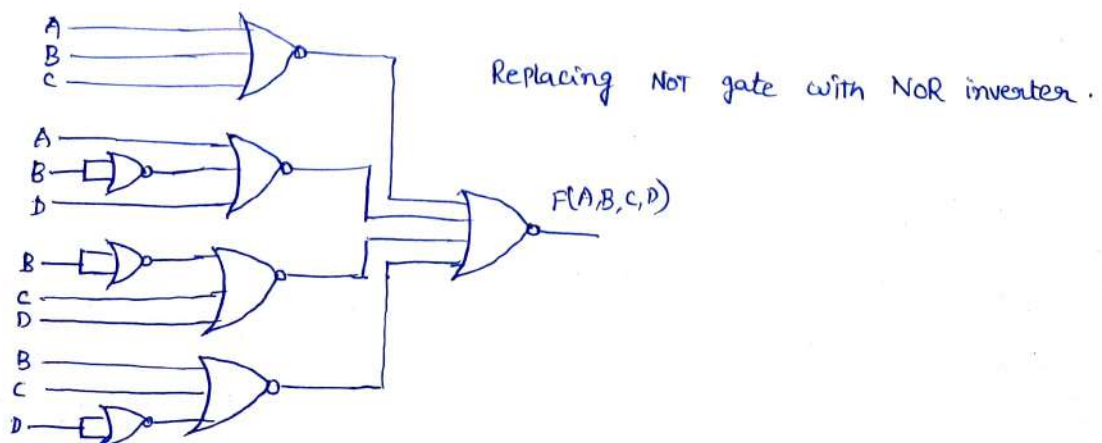
iii)



iv)



v)



2) simplify the following function in pos form using k-map.

$$F(A, B, C, D) = (\bar{A} + B + \bar{D}) (\bar{A} + \bar{B} + \bar{C}) (\bar{A} + \bar{B} + C) (\bar{B} + \bar{C} + \bar{D}) \text{ using k-map.}$$

sol) Given $F(A, B, C, D) = (\bar{A} + B + \bar{D}) (\bar{A} + \bar{B} + \bar{C}) (\bar{A} + \bar{B} + C) (\bar{B} + \bar{C} + \bar{D})$

The above function is not given in minterm canonical form. So we have to convert it into minterm canonical form.

$$F(A, B, C, D) = (\bar{A} + B + \bar{D} + C\bar{C}) (\bar{A} + \bar{B} + \bar{C} + D\bar{D}) (\bar{B} + \bar{C} + \bar{D} + A\bar{A}) (\bar{A} + \bar{B} + C + D\bar{D})$$

$$F(A, B, C, D) = (\bar{A} + B + \bar{D} + C) (\bar{A} + B + \bar{D} + \bar{C}) (\bar{A} + \bar{B} + \bar{C} + D) (\bar{A} + \bar{B} + \bar{C} + \bar{D}) (\bar{B} + \bar{C} + \bar{D} + A) (\bar{A} + \bar{B} + C + D) (\bar{A} + \bar{B} + C + \bar{D}) \cdot (\bar{B} + \bar{C} + \bar{D} + \bar{A})$$

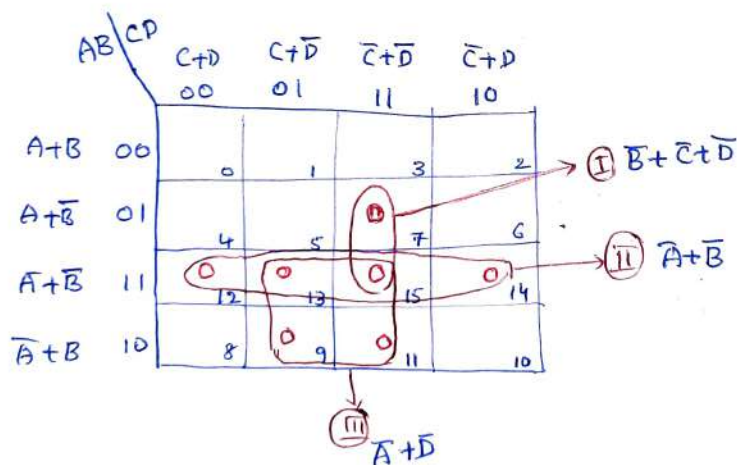
$$\Rightarrow F(A, B, C, D) = (\bar{A} + B + C + \bar{D}) (\bar{A} + B + \bar{C} + \bar{D}) (\bar{A} + \bar{B} + \bar{C} + D) (\bar{A} + \bar{B} + \bar{C} + \bar{D}) (\bar{A} + \bar{B} + C + D) (\bar{A} + \bar{B} + C + \bar{D}) \cdot (\bar{A} + \bar{B} + \bar{C} + \bar{D})$$

Removing the repeatedly occurring terms we get

$$F(A, B, C, D) = (\bar{A} + B + C + \bar{D}) (\bar{A} + B + \bar{C} + \bar{D}) (\bar{A} + \bar{B} + \bar{C} + D) (\bar{A} + \bar{B} + \bar{C} + \bar{D}) (\bar{A} + \bar{B} + C + D) (\bar{A} + \bar{B} + C + \bar{D}) \cdot (\bar{A} + \bar{B} + C + D) (\bar{A} + \bar{B} + C + \bar{D})$$

$$= M_9 \cdot M_{11} \cdot M_{14} M_{15} M_7 M_{12} M_{13}$$

$$= \Pi M(7, 9, 11, 12, 13, 14, 15)$$



$$\therefore F(A, B, C, D) = (\bar{B} + \bar{C} + \bar{D}) \cdot (\bar{A} + \bar{B}) \cdot (\bar{A} + \bar{D})$$

3) simplify the following boolean function into pos using k-map

$$F(A, B, C, D) = \Sigma m(2, 4, 8, 9, 11, 12, 13)$$

sol) Given $F(A, B, C, D) = \Sigma m(2, 4, 8, 9, 11, 12, 13)$

To simplify in pos form it is better to have the function in product of max terms form. Converting $F(A, B, C, D)$ into product of max terms form

we get $F(A, B, C, D) = \prod M(0, 1, 3, 5, 6, 7, 10, 14, 15)$

(10)

AB \ CD	C+D 00	C+D̄ 01	C̄+D̄ 11	C̄+D 10
A+B 00	0	1	3	2
A+B̄ 01	4	5	7	6
Ā+B̄ 11	12	13	15	14
Ā+B 10	8	9	11	10

Groupings and prime implicants:

- Group 1: (0, 1, 3, 2) → $A+B+C$
- Group 2: (0, 1, 4, 5) → $A+B+C$
- Group 3: (3, 7, 15, 11) → $A+\bar{D}$
- Group 4: (5, 7, 13, 15) → $\bar{B}+\bar{C}$
- Group 5: (12, 13, 15, 14) → $\bar{A}+\bar{C}+D$

$$\therefore F(A, B, C, D) = (A+B+C)(\bar{A}+\bar{C}+D)(A+\bar{D})(\bar{B}+\bar{C})$$

4) $F(A, B, C, D) = \prod M(4, 6, 10, 12, 13, 15)$ Convert into simplified POS form using K-map.

Sol) Given $F(A, B, C, D) = \prod M(4, 6, 10, 12, 13, 15)$

AB \ CD	C+D 00	C+D̄ 01	C̄+D̄ 11	C̄+D 10
A+B 00	0	1	3	2
A+B̄ 01	4	5	7	6
Ā+B̄ 11	12	13	15	14
Ā+B 10	8	9	11	10

Groupings and prime implicants:

- Group 1: (4, 5, 12, 13) → $\bar{A}+\bar{B}+D$
- Group 2: (4, 12) → $\bar{B}+C+D$
- Group 3: (13, 15) → $\bar{A}+\bar{B}+\bar{D}$
- Group 4: (10, 11) → $\bar{A}+B+\bar{C}+D$

$$\therefore F(A, B, C, D) = (\bar{A}+\bar{B}+\bar{C}+D)(\bar{A}+\bar{B}+D)(\bar{B}+C+D)(\bar{A}+\bar{B}+\bar{D})$$

5) $F(A, B, C, D) = \prod M(0, 1, 4, 7, 9, 14, 15)$ Simplify in POS using K-map.

Sol) Given $F(A, B, C, D) = \prod M(0, 1, 4, 7, 9, 14, 15)$

AB \ CD	C+D 00	C+D̄ 01	C̄+D̄ 11	C̄+D 10
A+B 00	0	1	3	2
A+B̄ 01	4	5	7	6
Ā+B̄ 11	12	13	15	14
Ā+B 10	8	9	11	10

Groupings and prime implicants:

- Group 1: (0, 1, 4, 5) → $A+C+D$
- Group 2: (4, 5, 12, 13) → $\bar{A}+\bar{B}+\bar{C}$
- Group 3: (9, 11) → $B+C+\bar{D}$
- Group 4: (13, 15) → $\bar{B}+\bar{C}+\bar{D}$

$$\therefore F(A, B, C, D) = (A+C+D)(\bar{A}+\bar{B}+\bar{C})(B+C+\bar{D})(\bar{B}+\bar{C}+\bar{D})$$

H.W

- Q) Simplify the following in pos form using K-map, and implement using NOR gate
- $F(W, X, Y, Z) = \Sigma m(0, 2, 4, 5, 6, 7, 8, 10, 13, 15)$
 - $F(A, B, C, D) = \Pi M(1, 4, 6, 9, 12, 13)$

Don't Care Conditions:

In some logic circuits certain input conditions will never occur. For those input conditions output of the circuit is not defined clearly. It can be either logic '1' or logic '0'. Such input conditions for which the output of the function is not defined is known as don't care conditions (or) incompletely specified functions.

For example in 4-bit BCD code the decimal digits from '0' through '15' are possible but the decimal digits '0' through '9' are considered to be valid BCD and the remaining 6 values (i.e from 10 to 15) are invalid BCD.

Ex. ① Let us consider the following truth table in which the outputs are defined for the inputs from 000 to 101, for the remaining input conditions the output is marked as don't care.

Don't care condition is denoted by any one of the following 'x' (or) 'φ' (or) d

A	B	C	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	x
1	1	1	x

From this truth table $Y(A, B, C)$ can be written as $Y(A, B, C) = \Sigma m(1, 3, 5) + d(6, 7)$

Ex. ② : Let us consider an example of even parity generator for a 4-bit BCD. The output for the last 6 input conditions can't be specified. So the output of the even parity generator for last 6 i/p conditions of BCD are don't care.

A	B	C	D	output of even parity generator P
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	x
1	0	1	1	x
1	1	0	0	x
1	1	0	1	x
1	1	1	0	x
1	1	1	1	x

Problem simplify the function given by the following truth table using K-map in SOP form.

A	B	C	$Y(A,B,C)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	x
1	1	1	x

sol)

From the given truth table $Y(A,B,C)$ can be written as
 $Y(A,B,C) = \sum m(1,3,5) + d(6,7)$

So drawing the k-map for $V(A, B, C)$ we get the following.

A \ BC	BC			
	$\overline{B}\overline{C}$ 00	$\overline{B}C$ 01	$B\overline{C}$ 11	BC 10
\overline{A} 0	0	1	1	2
A 1	4	1	X	X

↓
Ⓘ C

$$\therefore V(A, B, C) = C$$

2) $F(W, X, Y, Z) = \sum m(1, 3, 7, 11, 15) + d(0, 2, 5)$ simplify this function in sop using k-map.

sol) Given $F(W, X, Y, Z) = \sum m(1, 3, 7, 11, 15) + d(0, 2, 5)$

WX \ YZ	YZ			
	$\overline{Y}\overline{Z}$ 00	$\overline{Y}Z$ 01	YZ 11	$Y\overline{Z}$ 10
$\overline{W}\overline{X}$ 00	X	1	1	X
$\overline{W}X$ 01		X	1	
WX 11			1	
$W\overline{X}$ 10			1	

Ⓘ $\overline{W}\overline{X}$
Ⓜ YZ

$$\therefore F(W, X, Y, Z) = \overline{W}\overline{X} + YZ$$

3) Simplify $F(W, X, Y, Z) = \sum m(1, 3, 10) + d(0, 2, 8, 12)$ using k-map.

sol) Given $F(W, X, Y, Z) = \sum m(1, 3, 10) + d(0, 2, 8, 12)$

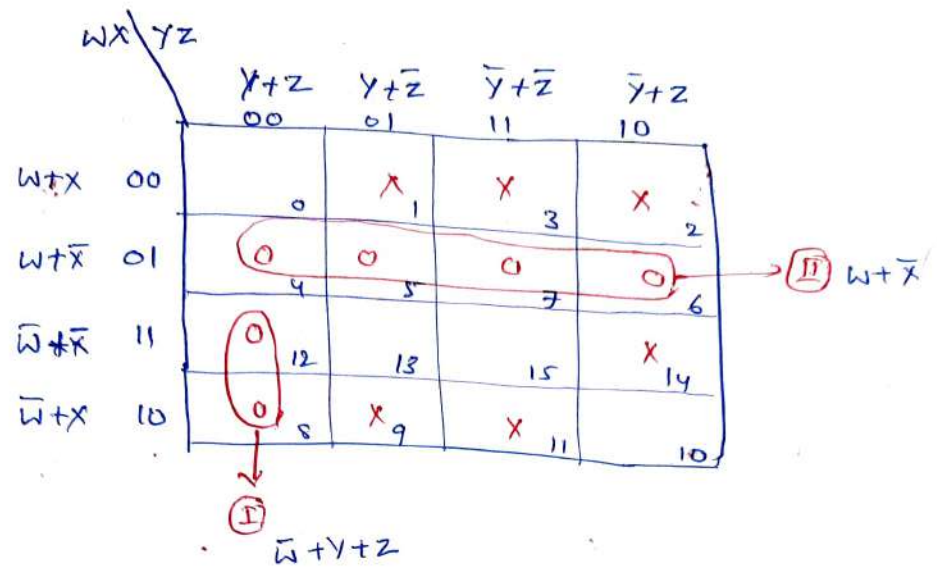
WX \ YZ	YZ			
	$\overline{Y}\overline{Z}$ 00	$\overline{Y}Z$ 01	YZ 11	$Y\overline{Z}$ 10
$\overline{W}\overline{X}$ 00	X	1	1	X
$\overline{W}X$ 01				
WX 11	X			
$W\overline{X}$ 10	X			1

Ⓘ $\overline{W}\overline{X}$
Ⓜ $\overline{X}\overline{Z}$

$$\therefore F(W, X, Y, Z) = \overline{W}\overline{X} + \overline{X}\overline{Z}$$

4) $F(W, X, Y, Z) = \Pi M(4, 5, 6, 7, 8, 12) \cdot d(1, 2, 3, 9, 11, 14)$ simplify using k-map in POS form.

sol) Given $F(W, X, Y, Z) = \Pi M(4, 5, 6, 7, 8, 12) \cdot d(1, 2, 3, 9, 11, 14)$



$\therefore F(W, X, Y, Z) = (\bar{W} + Y + Z)(W + \bar{X})$

Five Variable K-map:

A five variable K-map contains $2^5 = 32$ cells, but adjacent cells are difficult to identify on a single 32-cell k-map. Therefore, two 16-cell k-maps are used generally, to form a 32-cell k-map.

If the variables are A, B, C, D, and E, then two identical 16-cell k-maps containing B, C, D and E are constructed, one of the 16-cell k-map has $A=0$ (i.e. \bar{A} is present) and the other one has $A=1$ (i.e. A is present).

Every cell in one 16-cell k-map is adjacent to the corresponding cell in the other 16-cell k-map, because only one variable (A) changes between the corresponding cells of two 16-cell k-maps.

Thus every row on one 16-cell k-map is adjacent to the corresponding row (the one occupying the same position) on the other 16-cell k-map, as are corresponding columns.

Also the rightmost and leftmost columns within each 16-cell map are adjacent, just as they are in any 16-cell k-map, as are the top

and bottom rows. However the rightmost column of one k-map is not adjacent to the leftmost column of the other k-map. Since they are not corresponding columns. Nor is the top row of one 16-bit k-map adjacent to the bottom row of the other, 16-bit k-map.

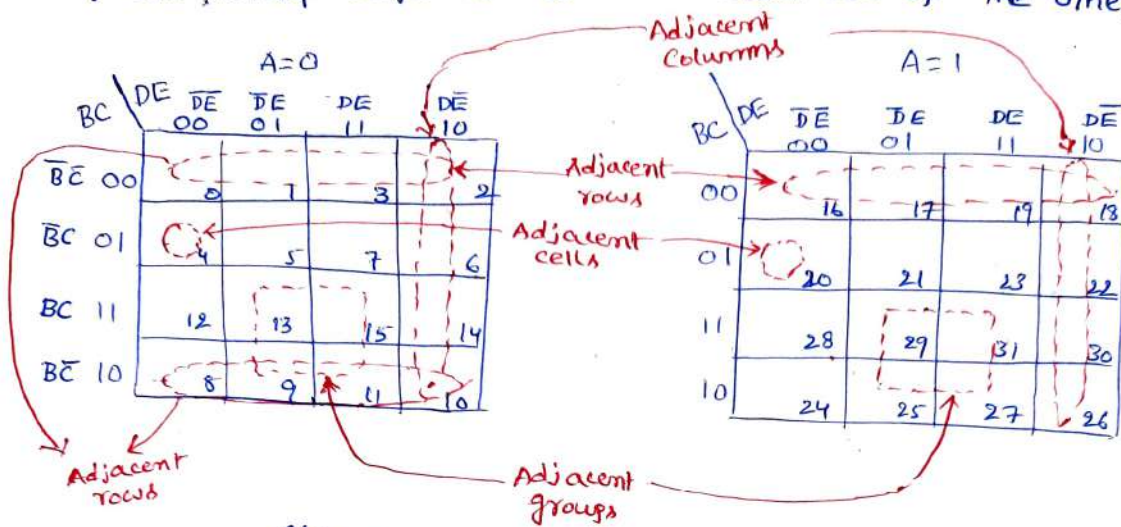


fig: A 5-variable k-map with examples of adjacencies.

Problems

Simplify the following 5-variable functions in SOP using k-map.

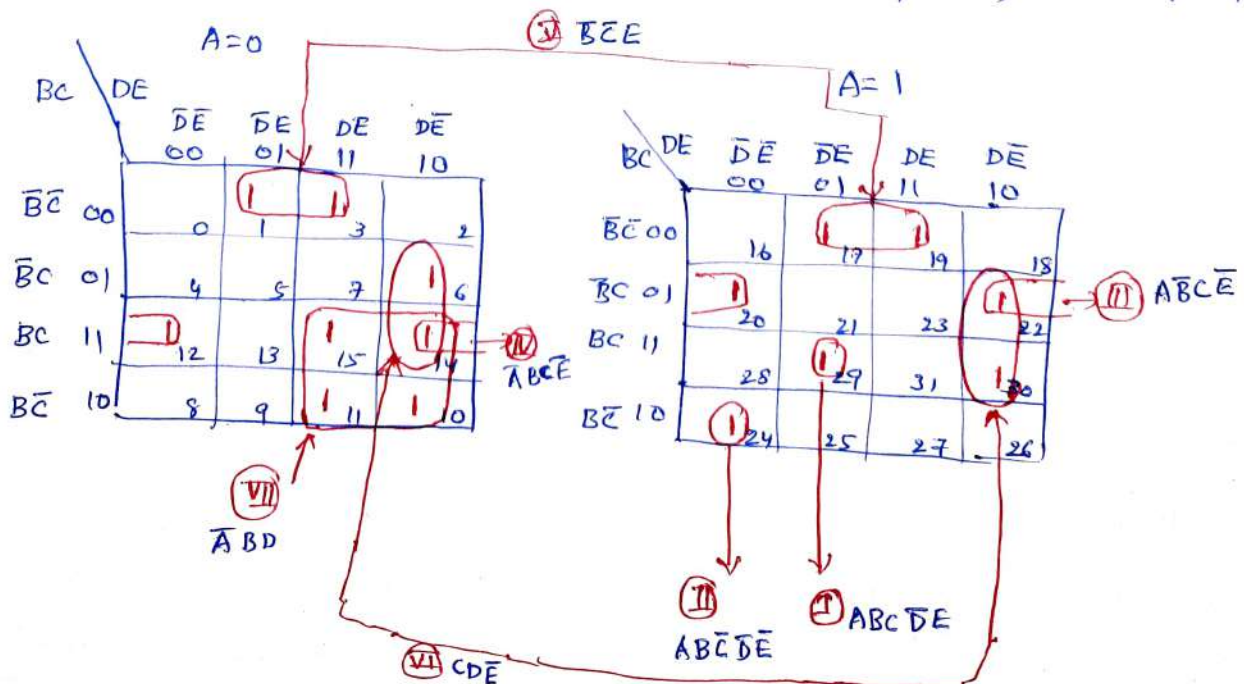
i) $f(A, B, C, D, E) = \sum m(1, 3, 6, 10, 11, 12, 14, 15, 17, 19, 20, 22, 24, 29, 30)$

ii) $f(V, W, X, Y, Z) = \sum m(3, 5, 6, 8, 9, 12, 13, 14, 19, 22, 24, 25, 30)$

iii) $f(A, B, C, D, E) = \sum m(0, 1, 2, 3, 6, 7, 14, 15, 17, 19, 31)$

iv) $f(A, B, C, D, E) = \sum m(3, 6, 7, 8, 10, 12, 14, 17, 19, 20, 21, 24, 25, 27, 31)$

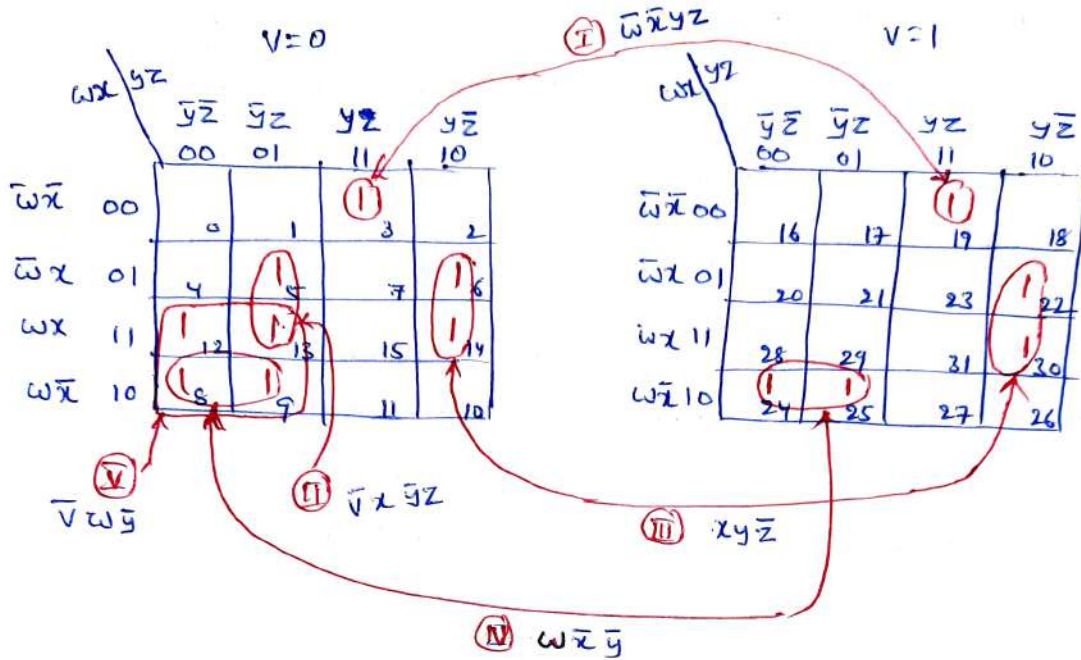
sol) Given $f(A, B, C, D, E) = \sum m(1, 3, 6, 10, 11, 12, 14, 15, 17, 19, 20, 22, 24, 29, 30)$



$$\therefore F(A,B,C,D,E) = ABC\bar{D}E + AB\bar{C}\bar{D}\bar{E} + A\bar{B}C\bar{E} + \bar{A}BC\bar{E} + \bar{B}\bar{C}E + CD\bar{E} + \bar{A}BD.$$

ii)
Sol)

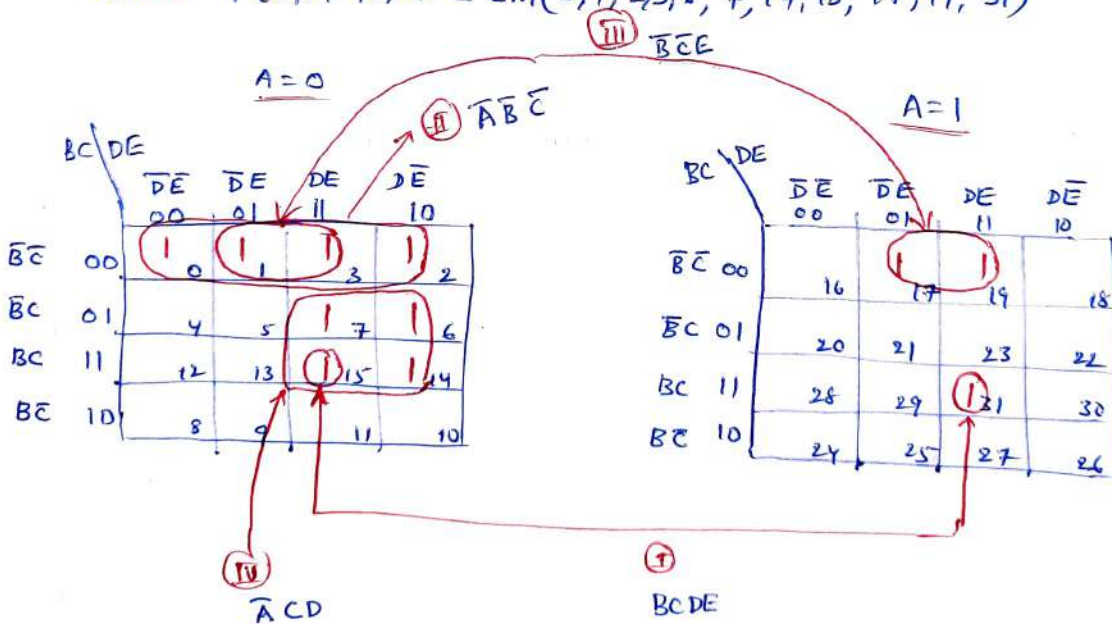
Given $F(V,W,X,Y,Z) = \Sigma m(3,5,6,8,9,12,13,14,19,22,24,25,30)$



$$\therefore F(V,W,X,Y,Z) = \bar{W}\bar{X}YZ + \bar{V}X\bar{Y}Z + XY\bar{Z} + W\bar{X}\bar{Y} + \bar{V}W\bar{Y}$$

iii)
Sol

Given $F(A,B,C,D,E) = \Sigma m(0,1,2,3,6,7,14,15,17,19,31)$



$$F(A,B,C,D,E) = \bar{A}\bar{B}\bar{C} + \bar{A}CD + \bar{B}\bar{C}E + BCDE$$

H.w

Simplify using K-map

$$F(A,B,C,D,E) = \Sigma m(0,4,8,12,18,20,26,28)$$

Limitations of Karnaugh map:

- * The K-map method of simplification is convenient as long as the number of variables does not exceed five or six. As the number of variables increases it is difficult to make judgements about which combinations form the minimum expression.
- * The K-map simplification is manual technique and simplification of a problem using K-map is highly depending on human abilities.

Tabular method (or) Quine Mc-cluskey method:

- * The K-map simplification is a manual technique and simplification process is totally depending on the human abilities.
- * To meet this need, W.V. Quine and E.J. McCluskey developed an exact tabular method to simplify the boolean expression. This tabular method is also known as Quine-McCluskey method.
- * The minterms whose binary equivalent differs only in one place, can be combined to reduce minterms. This is the fundamental - principle of the Quine McCluskey method.

The procedure for quine McCluskey method is as follows.

- 1) List out all the minterms in their binary equivalent form.
- 2) Arrange the minterms into groups according to the number of 1s and separate them by drawing a horizontal line between each group and its next group. This helps to search the binary minterms that differ only in one place.
- 3) Each binary number of a group is compared with every binary number of its next group and place a check mark beside each of the two terms if they are differing in only one place. Copy the term in the second column of the next table with a '-' in the position where ^{they} are differing.
- 4) This process of comparison is repeated for every minterm. Once this process is completed the same process is applied to the

new resultant terms copied into the next table.

5) These cycles are continued until a single pass through a cycle yields no further elimination of literals is possible.

6) Remaining terms that did not receive any check mark while comparing the terms, are called as prime implicants.

7) List all the prime implicants in a table and select the minimum number of prime implicants that cover all the minterms.

Example: Simplify the following boolean function by using tabular method.

$$F(A, B, C, D) = \sum m(0, 1, 3, 7, 8, 9, 11, 15)$$

Sol) Given $F(A, B, C, D) = \sum m(0, 1, 3, 7, 8, 9, 11, 15)$

Minterms	Binary Representation A B C D
m_0	0 0 0 0
m_1	0 0 0 1
m_3	0 0 1 1
m_7	0 1 1 1
m_8	1 0 0 0
m_9	1 0 0 1
m_{11}	1 0 1 1
m_{15}	1 1 1 1

Minterms	Binary Representation A B C D	
m_0	0 0 0 0	✓
m_1	0 0 0 1	✓
m_8	1 0 0 0	✓
m_3	0 0 1 1	✓
m_9	1 0 0 1	✓
m_7	0 1 1 1	✓
m_{11}	1 0 1 1	✓
m_{15}	1 1 1 1	✓

Minterms	Binary representation A B C D
0, 1	0 0 0 - ✓
0, 8	- 0 0 0 ✓
1, 3	0 0 - 1 ✓
1, 9	- 0 0 1 ✓
8, 9	1 0 0 - ✓
3, 7	0 - 1 1 ✓
3, 11	- 0 1 1 ✓
9, 11	1 0 - 1 ✓
7, 15	- 1 1 1 ✓
11, 15	1 - 1 1 ✓

Minterms	Binary representation A B C D
0, 1, 8, 9	- 0 0 -
0, 8, 1, 9	- 0 0 -
1, 3, 9, 11	- 0 - 1
1, 9, 3, 11	- 0 - 1
3, 7, 11, 15	- - 1 1
3, 11, 7, 15	- - 1 1

Minterms	Prime Implicants	Binary representation A B C D
0, 1, 8, 9	$\overline{B}\overline{C}$	- 0 0 -
1, 3, 9, 11	$\overline{B}D$	- 0 - 1
3, 7, 11, 15	CD	- - 1 1

From the list of prime implicants select the minimum number of prime implicants that cover all the minterms using the following procedure.

1) search for single dot columns and select the prime implicant corresponding to that dot by putting a check mark in front of it.

(15)

- 2) search for multi dot columns one by one if the corresponding minterm is already included in the final expression ignore it and goto the next multidot column otherwise include that prime implicant.
- 3) To implement the above two steps draw the prime implicant chart and place dot '(•)' against each prime implicant under the respective minterm columns as shown in below.

Prime Implicants		Minterms							
		m_0	m_1	m_3	m_7	m_8	m_9	m_{11}	m_{15}
$\checkmark \bar{B} \bar{C}$	0, 1, 8, 9	⊙	⊙			⊙	⊙		
$\bar{B} D$	1, 3, 9, 11		•	•			•	•	
$\checkmark CD$	3, 7, 11, 15			⊙	⊙			⊙	⊙

$$\therefore f(A, B, C, D) = \bar{B} \bar{C} + CD$$

② simplify the following boolean function by using Tabular method

$$F(A, B, C, D) = \sum m(0, 2, 3, 6, 7, 8, 10, 12, 13)$$

Sol) Given $F(A, B, C, D) = \sum m(0, 2, 3, 6, 7, 8, 10, 12, 13)$

Minterms	Binary representation
	A B C D
m_0	0 0 0 0
m_2	0 0 1 0
m_3	0 0 1 1
m_6	0 1 1 0
m_7	0 1 1 1
m_8	1 0 0 0
m_{10}	1 0 1 0
m_{12}	1 1 0 0
m_{13}	1 1 0 1

Minterm	Binary representation				
	A	B	C	D	
m_0	0	0	0	0	✓
m_2	0	0	1	0	✓
m_8	1	0	0	0	✓
m_3	0	0	1	1	✓
m_6	0	1	1	0	✓
m_{10}	1	0	1	0	✓
m_{12}	1	1	0	0	✓
m_7	0	1	1	1	✓
m_{13}	1	1	0	1	✓

Minterms	Binary representation				
	A	B	C	D	
m_0, m_2	0	0	-	0	✓
m_0, m_8	-	0	0	0	✓
m_2, m_3	0	0	1	-	✓
m_2, m_6	0	-	1	0	✓
m_2, m_{10}	-	0	1	0	✓
m_8, m_{10}	1	0	-	0	✓
m_8, m_{12}	1	-	0	0	
m_3, m_7	0	-	1	1	✓
m_6, m_7	0	1	1	-	✓
m_{12}, m_{13}	1	1	0	-	

Minterms	Binary representation				
	A	B	C	D	
m_0, m_2, m_8, m_{10}	-	0	-	0	
m_0, m_8, m_2, m_{10}	-	0	-	0	
m_2, m_3, m_6, m_7	0	-	1	-	
m_2, m_6, m_3, m_7	0	-	1	-	

Minterms	Prime implicants	Binary representation A B C D
8, 12	$\overline{A} \overline{C} \overline{D}$	1 - 0 0
12, 13	$A B \overline{C}$	1 1 0 -
0, 2, 8, 10	$\overline{B} \overline{D}$	- 0 - 0
2, 3, 6, 7	$\overline{A} C$	0 - 1 -

From the above prime implicants select minimum number of prime implicants that cover all the minterms using the following procedure.

- 1) Draw the prime implicant chart and place '•' against the prime implicants under the respective minterm columns.
- 2) search for single dot columns and select the prime implicant corresponding to that dot by putting a check mark in front of it.
- 3) search for multi dot columns one by one, if the corresponding minterm is already included in the final expression ignore that prime implicant and go to the next multi dot column, otherwise include the corresponding prime implicant in the final expression.

Prime implicant chart:

Prime implicants		Minterms								
		m_0	m_2	m_3	m_6	m_7	m_8	m_{10}	m_{12}	m_{13}
$\overline{A} \overline{C} \overline{D}$	8, 12						•		•	
✓ $A B \overline{C}$	12, 13								•	•
✓ $\overline{B} \overline{D}$	0, 2, 8, 10	•	•				•	•		
✓ $\overline{A} C$	2, 3, 6, 7		•	•	•	•				

$$\therefore f(A, B, C, D) = \overline{A} \overline{C} \overline{D} + \overline{B} \overline{D} + \overline{A} C$$

③ Simplify the following boolean expression using tabular method.

$$Y(A, B, C, D) = \bar{A} B \bar{C} \bar{D} + \bar{A} B \bar{C} D + A B \bar{C} \bar{D} + A B \bar{C} D + \bar{A} \bar{B} C \bar{D} + A B \bar{C} D$$

Sol) $Y(A, B, C, D) = \bar{A} B \bar{C} \bar{D} + \bar{A} B \bar{C} D + A B \bar{C} \bar{D} + A B \bar{C} D + \bar{A} \bar{B} C \bar{D} + A B \bar{C} D$

$$= m_2 + m_5 + m_{12} + m_9 + m_2 + m_{13}$$

$$\therefore Y(A, B, C, D) = \sum m(2, 4, 5, 9, 12, 13)$$

Minterm	Binary representation A B C D
m_2	0 0 1 0
m_4	0 1 0 0
m_5	0 1 0 1
m_9	1 0 0 1
m_{12}	1 1 0 0
m_{13}	1 1 0 1

Minterm	Binary representation A B C D
m_2	0 0 1 0
m_4	0 1 0 0 ✓
m_5	0 1 0 1 ✓
m_9	1 0 0 1 ✓
m_{12}	1 1 0 0 ✓
m_{13}	1 1 0 1 ✓

Minterms	Binary representation A B C D
m_2, m_5	0 1 0 - ✓
m_4, m_{12}	- 1 0 0 ✓
m_5, m_{13}	- 1 0 1 ✓
m_9, m_{13}	1 - 0 1
m_{12}, m_{13}	1 1 0 - ✓

Minterms	Binary representation A B C D
m_4, m_5, m_{12}, m_{13}	- 1 0 -
m_4, m_{12}, m_5, m_{13}	- 1 0 -

Prime Implicants	Minterms	Binary representation A B C D
$\bar{A}\bar{B}C\bar{D}$	m_2	0 0 1 0
$A\bar{C}D$	m_9, m_{13}	1 - 0 1
$B\bar{C}$	m_4, m_5, m_{12}, m_{13}	- 1 0 -

From the above prime implicants select minimum number of prime implicants that cover all the minterms using the following procedure.

- 1) Draw the prime implicant chart and place '•' against the prime implicants under the respective minterm columns.
- 2) Search for single dot columns and select the prime implicant corresponding to that by putting a check mark in front of it.
- 3) Search for multi dot columns one by one, if the corresponding minterm is already included in the final expression ignore that prime implicant and go to the next multi dot column, otherwise include the corresponding prime implicant in the final expression.

Prime implicant chart

Prime implicants		Minterms						
		m_2	m_4	m_5	m_9	m_{12}	m_{13}	
$\checkmark \bar{A}\bar{B}C\bar{D}$	2	⊙						
$\checkmark A\bar{C}D$	9, 13				⊙		⊙	
$\checkmark B\bar{C}$	4, 5, 12, 13		⊙	⊙		⊙	⊙	

$$\therefore Y(A, B, C, D) = \bar{A}\bar{B}C\bar{D} + A\bar{C}D + B\bar{C}$$

4> Simplify the following boolean function using tabulation method.

$$Y(A,B,C,D) = \sum m(1,2,3,5,9,12,14,15) + \sum d(4,8,11)$$

Sol> Given $Y(A,B,C,D) = \sum m(1,2,3,5,9,12,14,15) + \sum d(4,8,11)$

Minterms	Binary representation A B C D
m_1	0 0 0 1
m_2	0 0 1 0
m_3	0 0 1 1
m_5	0 1 0 1
m_9	1 0 0 1
m_{12}	1 1 0 0
m_{14}	1 1 1 0
m_{15}	1 1 1 1
$d m_4$	0 1 0 0
$d m_8$	1 0 0 0
$d m_{11}$	1 0 1 1

Minterm	Binary representation A B C D	
m_1	0 0 0 1	✓
m_2	0 0 1 0	✓
$d m_4$	0 1 0 0	✓
$d m_8$	1 0 0 0	✓
m_3	0 0 1 1	✓
m_5	0 1 0 1	✓
m_9	1 0 0 1	✓
m_{12}	1 1 0 0	✓
$d m_{11}$	1 0 1 1	✓
m_{14}	1 1 1 0	✓
m_{15}	1 1 1 1	✓

Minterms	Binary representation A B C D
m_1, m_3	0 0 - 1 ✓
m_1, m_5	0 - 0 1
m_1, m_9	- 0 0 1 ✓
m_2, m_3	0 0 1 -
dm_4, m_5	0 1 0 -
dm_4, m_{12}	- 1 0 0
dm_8, m_9	1 0 0 -
dm_8, m_{12}	1 - 0 0
m_3, dm_{11}	- 0 1 1 ✓
m_9, dm_{11}	1 0 - 1 ✓
m_{12}, m_{14}	1 1 - 0
dm_{11}, m_{15}	1 - 1 1
m_{14}, m_{15}	1 1 1 -

minterms	Binary representation A B C D
m_1, m_3, m_9, dm_{11}	- 0 - 1
m_1, m_9, m_3, dm_{11}	- 0 - 1

prime implicants	Minterms	Binary representation A B C D
$\bar{A} \bar{C} D$	m_1, m_5	0 - 0 1
$\bar{A} \bar{B} C$	m_2, m_3	0 0 1 -
$\bar{A} B \bar{C}$	dm_4, m_5	0 1 0 -
$B \bar{C} \bar{D}$	dm_4, m_{12}	- 1 0 0
$A \bar{B} \bar{C}$	dm_8, m_9	1 0 0 -
$A \bar{C} \bar{D}$	dm_8, m_{12}	1 - 0 0
$A B \bar{D}$	m_{12}, m_{14}	1 1 - 0
$A C D$	dm_{11}, m_{15}	1 - 1 1
$A B C$	m_{14}, m_{15}	1 1 1 -
$\bar{B} D$	m_1, m_3, m_9, dm_{11}	- 0 - 1

- From the above prime implicants select minimum numbers of Prime -
 - implicants that cover all the minterms using the following procedure
- 1) Draw the prime implicant chart and place '•' against the prime-implicants under the respective minterm columns.
 - 2) Search for single dot columns and select the prime implicant -
 corresponding to that by putting a check mark in front of it.
 - 3) Search for multidot columns one by one, if the corresponding min-
 -term is already included in the final expression ignore that prime-
 implicant and go to the next multidot column, otherwise include
 the corresponding prime implicant in the final expression.

Prime implicant chart:

Prime Implicants		Minterms										
		m_1	m_2	m_3	dm_4	m_5	dm_6	m_7	dm_{11}	m_{12}	m_{14}	m_{15}
✓ $\bar{A}\bar{C}D$	1, 5	⊙				⊙						
✓ $\bar{A}\bar{B}C$	2, 3		⊙	⊙								
$\bar{A}B\bar{C}$	4, 5				•	•						
$B\bar{C}\bar{D}$	4, 12				•					•		
$A\bar{B}\bar{C}$	8, 9						•	•				
$A\bar{C}\bar{D}$	8, 12						•			•		
✓ $AB\bar{D}$	12, 14									⊙	⊙	
ACD	11, 15								•			
✓ ABC	14, 15										•	•
✓ $\bar{B}D$	1, 3, 9, 11	⊙		⊙			⊙	⊙		⊙	⊙	

$$\therefore Y(A, B, C, D) = \bar{A}\bar{C}D + \bar{A}\bar{B}C + AB\bar{D} + ABC + \bar{B}D$$

- 5) obtain the simplified SOP form function using Quine McCluskey method
 for the function $F(A, B, C, D, E) = \sum m(0, 1, 2, 8, 9, 15, 17, 21, 24, 25, 27, 31)$

Sol> Given boolean function

$$F(A, B, C, D, E) = \sum m(0, 1, 2, 8, 9, 15, 17, 21, 24, 25, 27, 31)$$

Minterm	Binary representation
	A B C D E
m_0	0 0 0 0 0
m_1	0 0 0 0 1
m_2	0 0 0 1 0
m_8	0 1 0 0 0
m_9	0 1 0 0 1
m_{15}	0 1 1 1 1
m_{17}	1 0 0 0 1
m_{21}	1 0 1 0 1
m_{24}	1 1 0 0 0
m_{25}	1 1 0 0 1
m_{27}	1 1 0 1 1
m_{31}	1 1 1 1 1

Minterm	Binary representation
	A B C D E
m_0	0 0 0 0 0 ✓
m_1	0 0 0 0 1 ✓
m_2	0 0 0 1 0 ✓
m_8	0 1 0 0 0 ✓
m_9	0 1 0 0 1 ✓
m_{17}	1 0 0 0 1 ✓
m_{24}	1 1 0 0 0 ✓
m_{21}	1 0 1 0 1 ✓
m_{25}	1 1 0 0 1 ✓
m_{15}	0 1 1 1 1 ✓
m_{27}	1 1 0 1 1 ✓
m_{31}	1 1 1 1 1 ✓

Minterms	Binary representation A B C D E
m_0, m_1	0 0 0 0 - ✓
m_0, m_2	0 0 0 - 0
m_0, m_8	0 - 0 0 0 ✓
m_1, m_9	0 - 0 0 1 ✓
m_1, m_{17}	- 0 0 0 1 ✓
m_8, m_9	0 1 0 0 - ✓
m_8, m_{24}	- 1 0 0 0 ✓
m_9, m_{25}	- 1 0 0 1 ✓
m_{17}, m_{25}	1 0 - 0 1
m_{17}, m_{25}	1 - 0 0 1 ✓
m_{24}, m_{25}	1 1 0 0 - ✓
m_{25}, m_{27}	1 1 0 - 1
m_{15}, m_{31}	- 1 1 1 1
m_{27}, m_{31}	1 1 - 1 1

minterms	Binary representation A B C D E
m_0, m_1, m_8, m_9	0 - 0 0 -
m_0, m_8, m_1, m_9	0 - 0 0 -
m_1, m_9, m_{17}, m_{25}	- - 0 0 1
m_1, m_{17}, m_9, m_{25}	- - 0 0 1
m_8, m_9, m_{24}, m_{25}	- 1 0 0 -
m_8, m_{24}, m_9, m_{25}	- 1 0 0 -

Prime implicants	Minterms	Binary representation A B C D E
$A \bar{B} \bar{C} \bar{E}$	m_0, m_2	0 0 0 - 0
$A \bar{B} \bar{D} E$	m_{17}, m_{21}	1 0 - 0 1
$A B \bar{C} E$	m_{25}, m_{27}	1 1 0 - 1
$B C D E$	m_{15}, m_{31}	- 1 1 1 1

Table
Continues...

(20)

Prime implicants	Minterms	Binary representation A B C D E
$ABDE$	m_{27}, m_{31}	1 1 - 1 1
$\bar{A}\bar{C}\bar{D}$	m_0, m_1, m_8, m_9	0 - 0 0 -
$\bar{C}\bar{D}E$	m_1, m_9, m_{17}, m_{25}	- - 0 0 1
$B\bar{C}\bar{D}$	m_8, m_9, m_{24}, m_{25}	- 1 0 0 -

From the above prime implicants select minimum number of prime implicants that cover all the minterms using the following procedure

1) draw the prime implicant chart and place '•' against the prime implicants under the respective minterm columns.

2) search for single dot columns and select the prime implicant corresponding to that by putting a check mark in front of it

3) search for multi dot columns one by one, if the corresponding minterm is already included in the final expression ignore that prime implicant and go to the next multidot column, otherwise include the prime implicant in the final expression. The prime implicant chart is given below.

prime implicant chart:

Prime Implicants		Minterms											
		m_0	m_1	m_2	m_8	m_9	m_{15}	m_{17}	m_{21}	m_{24}	m_{25}	m_{27}	m_{31}
✓ $\bar{A}\bar{B}\bar{C}\bar{E}$	m_0, m_2	⊙		⊙									
✓ $\bar{A}\bar{B}\bar{D}\bar{E}$	m_{17}, m_{21}						⊙		⊙				
$\bar{A}\bar{B}\bar{C}E$	m_{25}, m_{27}												
✓ $\bar{B}\bar{C}\bar{D}E$	m_{15}, m_{31}					⊙					•	•	
✓ $\bar{A}\bar{B}\bar{D}E$	m_{27}, m_{31}												⊙
✓ $\bar{A}\bar{C}\bar{D}$	m_0, m_1, m_8, m_9	⊙	⊙		⊙	⊙						⊙	⊙
$\bar{C}\bar{D}E$	m_1, m_9, m_{17}, m_{25}		•			•		•					
✓ $B\bar{C}\bar{D}$	m_8, m_9, m_{24}, m_{25}			⊙	⊙					⊙	⊙		

$$\therefore F(A, B, C, D, E) = \bar{A}\bar{B}\bar{C}\bar{E} + \bar{A}\bar{B}\bar{D}\bar{E} + \bar{B}\bar{C}\bar{D}E + \bar{A}\bar{B}\bar{D}E + \bar{A}\bar{C}\bar{D} + B\bar{C}\bar{D}$$

H.W Simplify the following using Quine Mc cluskey method

(i) $F(A, B, C, D) = \sum m(3, 7, 8, 12, 13, 15) + \sum d(9, 14)$ (ii) $F(A, B, C, D, E) = \sum m(0, 4, 8, 12, 16, 20, 24, 28) + \sum d(1, 5, 7, 23)$

EX-OR Function:

The EX-OR function is denoted by the symbol \oplus . The EX-OR operation between two variables x and y is given by the expression

$$x \oplus y = x'y + xy'$$

It is equal to 1 if only x is equal to 1 or if only y is equal to 1 but not when both x and y are equal to 1.

When x and y are same i.e., $x=y=0$ (or) $x=y=1$, the EX-OR function is equal to '0'.

The identities of EX-OR function are given below

i) $x \oplus 0 = x$

ii) $x \oplus 1 = x'$

iii) $x \oplus x = 0$

iv) $x \oplus x' = 1$

v) $x' \oplus y = x \oplus y' = (x \oplus y)' = x \odot y = x'y' + xy$

The Exclusive-OR function satisfies the following two laws

i) $x \oplus y = y \oplus x$ (Commutative Law)

ii) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ (Associative Law)

A two input EX-OR function is constructed with basic logic gates as shown in below.

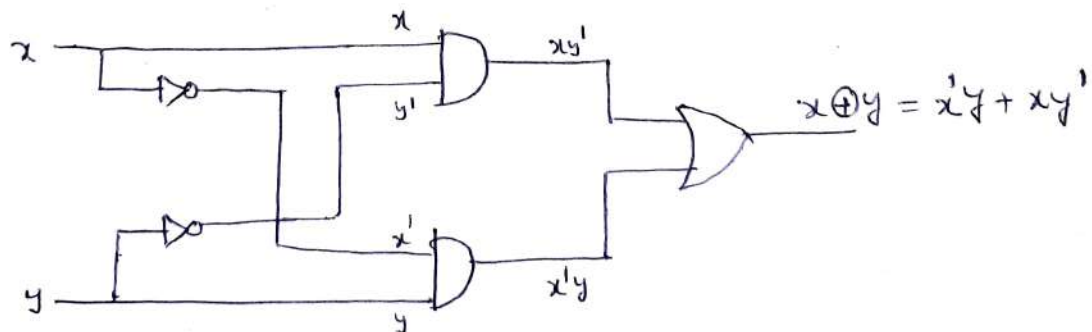


fig: Two input EX-OR function with basic logic gates.

In general the EX-OR gates with multiple inputs do not exist as they are very difficult to fabricate. A two variable EX-OR function with minimum number of NAND gates is as shown.

in below figure.

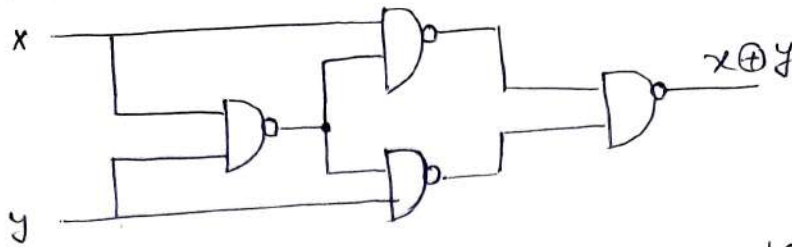


fig: EXOR function with minimum number of NAND Gates.

EX-OR function as an odd function:

A multiple input EX-OR function is equal to 1, when there are odd number of 1's in the inputs. Consider the following truth table of 3-variable EX-OR function.

X	Y	Z	$X \oplus Y \oplus Z$
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

fig: Truth table of 3 variable EX-OR function.

In Particular, a three-variable EX-OR function can be written into a boolean expression as follows

$$\begin{aligned}
 X \oplus Y \oplus Z &= (X \oplus Y) \oplus Z \\
 &= (X \oplus Y) \cdot Z' + (X \oplus Y)' \cdot Z \\
 &= (X'Y + XY') Z' + (X \odot Y) \cdot Z \\
 &= X'Y Z' + XY' Z' + (X'Y' + XY) Z \\
 &= X'Y Z' + XY' Z' + X'Y' Z + XYZ
 \end{aligned}$$

$$= m_2 + m_4 + m_1 + m_7$$

$$= \sum m(1, 2, 4, 7)$$

$$\therefore X \oplus Y \oplus Z = \sum m(1, 2, 4, 7)$$

Therefore from the above equation we can say that the EX-OR function of three variables can be expressed as a sum of min terms.

The EX-OR function can be used in the circuits like Parity Generator and Parity checker as discussed below.

Ex-OR function in parity Generator and in parity checker circuits:

The Ex-OR function is very useful in the systems that perform error detection and error correction. Consider a parity generator circuit of even parity having the output 'P'. The output of the parity generator circuit is equal to '1', when there are odd number of 1's in the input.

Therefore the parity generator circuit can be implemented with EX-OR function as the EX-OR function also produces the output '1' for odd number of 1's in the input.

The parity generator circuit with three inputs X, Y, Z and the output 'P' is as shown in below. The figure (b) shows the truth table for the parity generator.

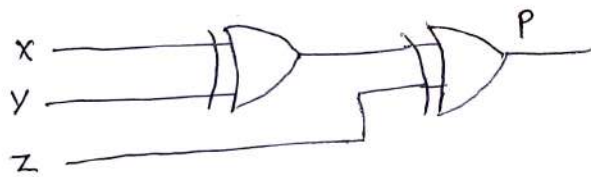


fig a) Parity generator circuit using EX-OR function.

X	Y	Z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

fig(b) Truth table for parity Generator

To check whether there is an error in the received binary number, we use the Parity checker circuit at the receiver. The receiver receives the binary number along with the Parity bit.

If we receive the binary number with odd number of 1's in the received binary, it is treated as there is an error in the received binary and the Parity checker circuit produces the output '1'.

If even number of 1's are available in the received binary, the parity checker produces output '0'.

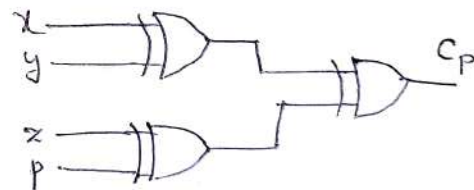
The Exclusive OR function also produces '1' when there are odd number of 1's in the input and produces '0' when there are even number of 1's in the input. So parity checker also generated by using Exclusive-OR function.

The Parity checker receives the message bits x, y, z along with the Parity bit ' p '. Let the output of the parity checker as C_p . The following truth table gives the output of the parity checker for various combinations of input.

x	y	z	p	C_p
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

fig (a) Truth table for Parity checker

The following circuit shows the Parity checker, with x, y, z, p as inputs and C_p as output



$$C_p = x \oplus y \oplus z \oplus p$$

fig: Parity checker circuit using EX-OR function.

Two level and multilevel implementations :

* We know that the desired logic gates can be connected in series to achieve a Particular Boolean Function.

* The maximum number of gates that are connected in between an input and output of a logic circuit represents the level of a gate implementation.

* If there are two gates between the input and output in maximum then it is a two level gate implementation.

* The SOP form function and POS form functions can be implemented by using two level gate implementations.

* The SOP form function can be implemented by using AND-OR logic circuit, the POS form function can be implemented by using OR-AND logic circuit.

Ex: SOP form function $F = AB' + A'B$

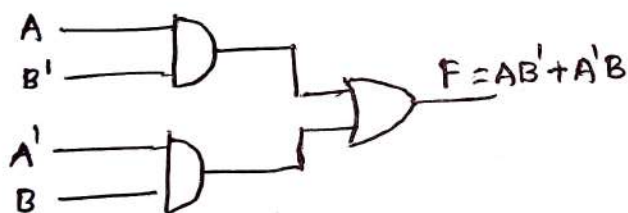


Fig: AND-OR logic circuit

Ex: POS form function $F = (A+B')(A'+B)$

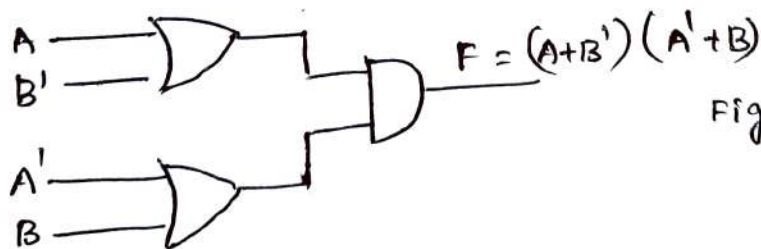


Fig: OR-AND logic circuit

* In any two level (or) multi level gate implementations the inputs are assumed to be readily available. That means any variable either in it's normal form or in it's complement form, it is assumed that they are readily available. No need of using separate NOT gates to get A' , B' , C' from A , B , C respectively.

- * Any SOP form function can be implemented by using two level AND-OR network. The two level AND-OR network is very easily implemented using NAND gates.
- * Any POS form function can be implemented by using a two level OR-AND network. The two level OR-AND network is very easily implemented using NOR gates.
- * Similarly if there are three 3 gates in maximum, between an input and output of a logic circuit, it is known as a three-level gate implementation.
- * Generally if there are more than two gates between an input and output of a logic circuit, it is called as a multi level gate implementation.

Example for multi level gate implementation.

$$F = AD(B+c') + AB(c+d')$$

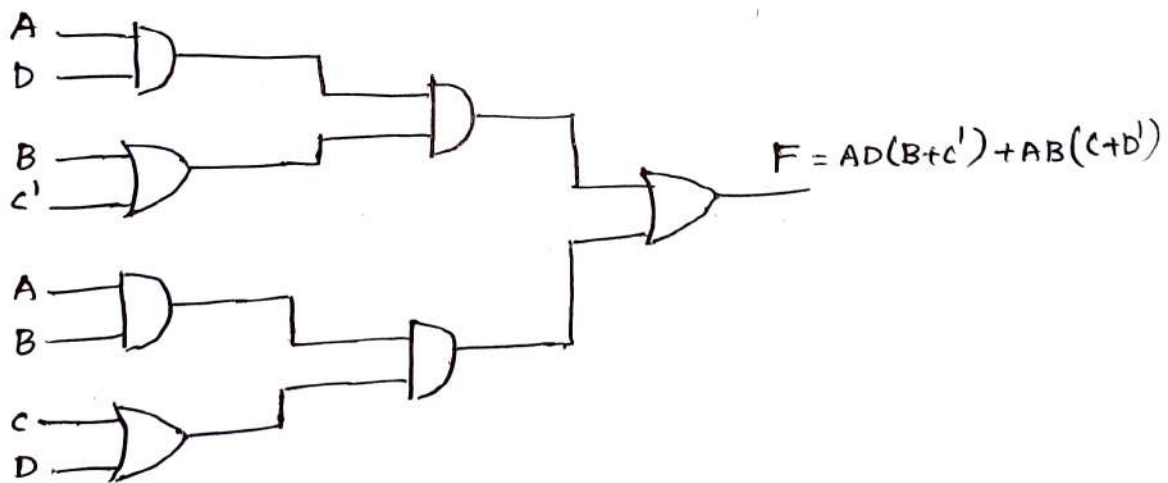


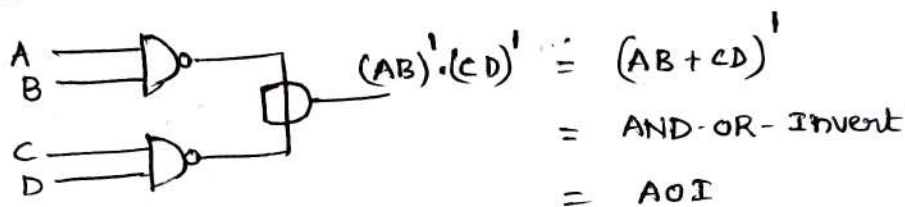
Fig: multi level (3-level) gate implementation.

In this example there are 3 - logic gates between any input and the final output, so this is a three level logic circuit.

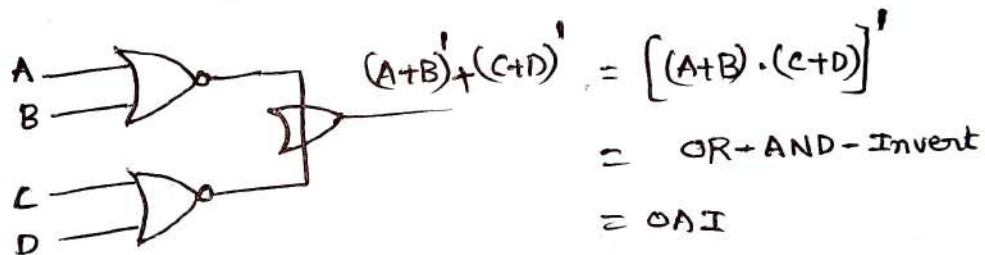
Other two level implementations: Wired Logic:

* Some NAND gates and NOR gates allow the possibility of wired connections between their outputs to provide a specific logic function. This type of logic is called as wired logic.

* For example open collector TTL NAND gates, when their outputs are tied together they perform wired AND logic. The wired AND Logic is not a Physical AND gate, it is represented as shown in below.



* Similarly the NOR gates of ECL logic family produces the wired OR logic, when their outputs are tied together.



ii) Degenerate and non degenerate forms:

Let us consider the logic gates AND, OR, NAND, NOR. When we assign any one of these four gates in the first level and any one of them in the second level there are 16 possible two level gate implementations.

Among these 16 possible two level gate implementations 8 of them are said to be degenerate forms and the remaining 8 are non degenerate forms.

In Degenerate forms there is only a single operation i.e.

performed at the output of the two level logic circuit, that can be either AND (or) OR.

In non-Degenerate form, at the output of their two level logic circuits SOP, POS, AOI (or) OAI operations are performed.

Under Degenerate forms we have the following 8 combinations.

AND - AND }
NAND - NOR } AND
OR - NOR } operation
NOR - AND }

OR - OR }
NOR - NAND } OR
AND - NAND } operation.
NAND - OR }

Under non degenerate forms we have the following 8 combinations

AND - OR } SOP
NAND - NAND }

AND - NOR } AND OR Invert
NAND - AND } (AOI)

OR - AND } POS
NOR - NOR }

OR - NAND } OR AND Invert
NOR - OR } (OAI)

* A boolean function can also be implemented by using AND-OR-Invert form (or) OR AND Invert form.

* To get AND-OR-Invert (AOI) form of a function, first find F' in SOP form by grouping 0's in the K-map and then complement it.

* To get OR-AND-Invert (OAI) form of a function, first find F' in POS form by grouping 1's in the K-map and then complement it.

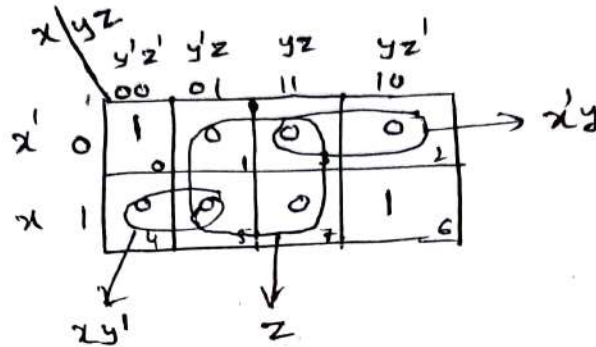
Ex: Implement the following functions with AOI, OAI forms.

i) $F(x, y, z) = \sum m(0, 6)$

H.W ii) $F(w, x, y, z) = \sum m(1, 3, 4, 5, 6, 7, 9, 11, 13, 15)$

i) sol given $F(x, y, z) = \sum m(0, 6)$

AOI :



$$\therefore F'(x, y, z) = z + x'y + xy'$$

$$\Rightarrow F(x, y, z) = (z + x'y + xy')'$$

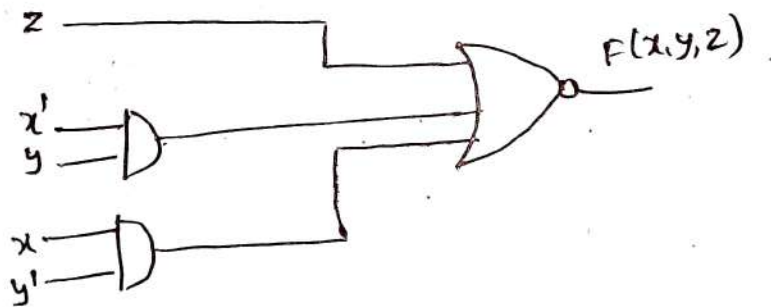


fig: Implementation of AOI form using AND-NOR network

The above figure can be modified as below.

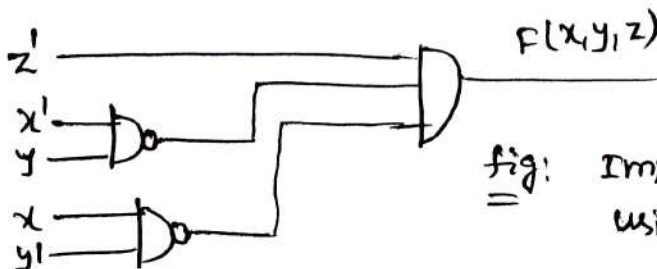
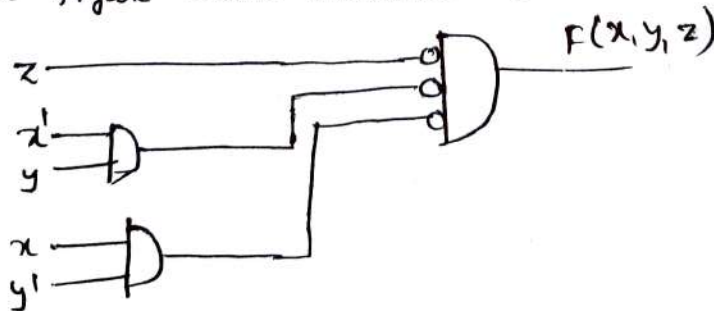


fig: Implementation of AOI form using NAND-AND network

OAI:

Given $F(x,y,z) = \sum m(0,6) = \prod M(1,2,3,4,5,7)$

		$x \backslash yz$			
		yz	yz'	$y'z$	$(y'z')$
		00	01	11	10
① $x+y+z$	x 0	1	0	0	0
	x' 1	0	0	0	1
		0	1	3	2
		4	5	7	

② $x'+y'+z$

$$F(x,y,z) = (x+y+z)(x'+y'+z)$$

$$\Rightarrow F(x,y,z) = [(x+y+z)(x'+y'+z)]'$$

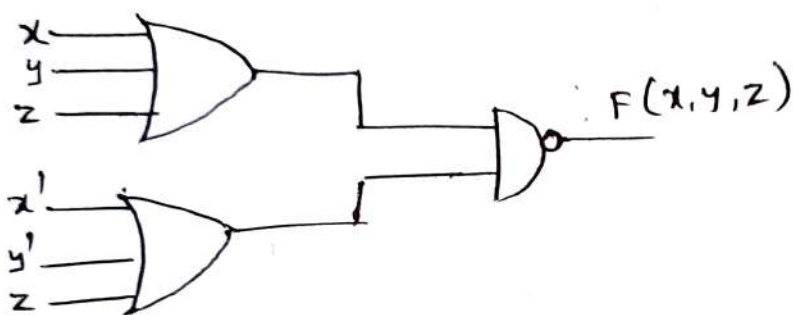


fig: OAI implementation using OR-NAND network

The above figure can be modified as shown in below.

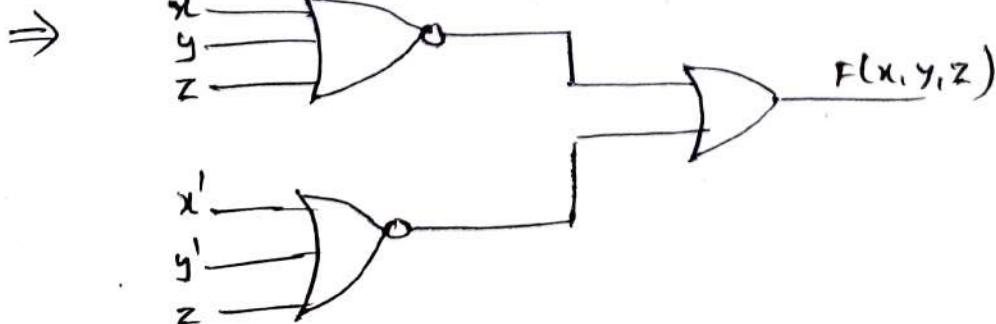
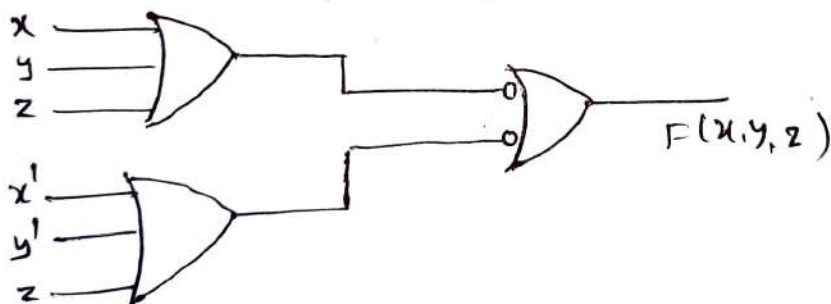


fig: OAI implementation using NOR-OR network

3. Combinational logic circuits

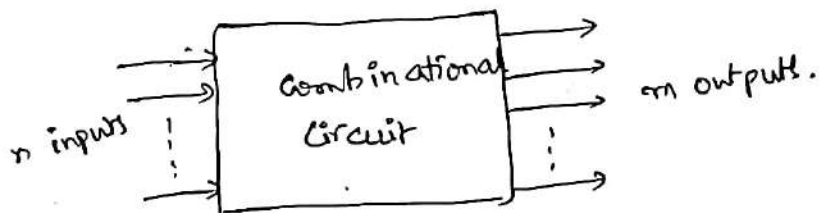
① Logic circuits for digital systems may be combinational or sequential. A combinational circuit consists of logic gates whose outputs at any time are determined from only the present combination of inputs.

In contrast, sequential circuits employ storage elements in addition to logic gates. Their outputs are a function of the inputs and the state of the storage elements. Because the state of the storage elements is a function of previous inputs, the outputs of a sequential circuit depend not only on present values of inputs but also on past inputs.

Combinational circuits :-

A combinational circuit consists of input variables, logic gates and output variables. Combinational logic gates react to the values of signals at their inputs and produce the value of output signal by transforming binary information from the given input data to a required output data.

A block diagram of a combinational circuit is as shown in fig.



The n input binary variables come from an external source. The m output variables are produced by the internal combinational logic circuit and go to an external destination.

For n input variables, there are 2^n possible binary input combinations. For each possible input combination, there is one possible output value. Thus, a combinational circuit can be specified with a truth table that lists the output values for each combination of input variables.

(b) A combinational circuit can also be described by m boolean functions, one for each output variable. Each output function is expressed

in terms of the n input variables.

③ Analysis procedure:-

The analysis of a combinational circuit starts with a given logic diagram and culminates with a set of boolean functions (or) a truth table (or) possibly an explanation of the circuit operation.

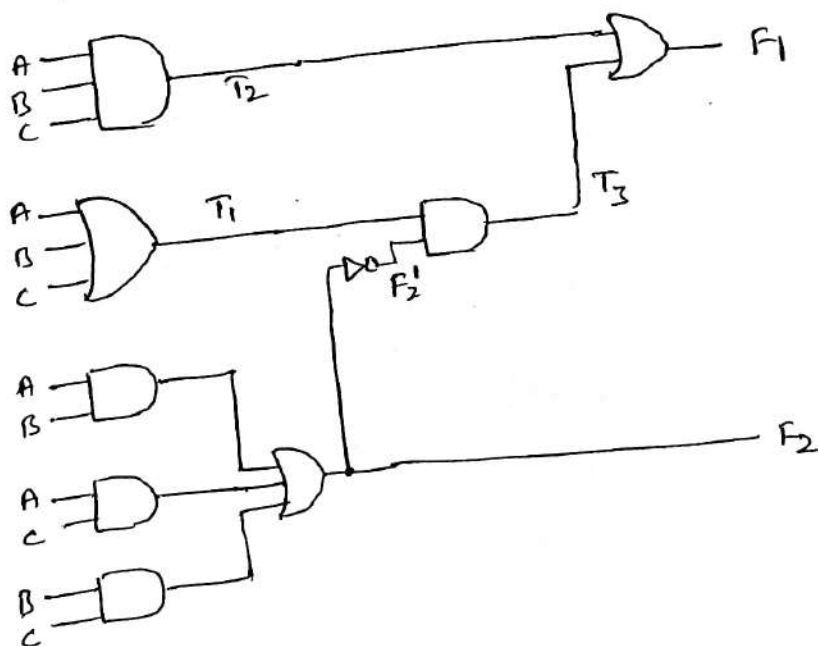
The first step in the analysis is to make sure that the given circuit is a combinational. The combinational circuit has logic gates with no feedback paths (or) memory elements.

Once the logic diagram is verified to be that of a combinational circuit, one can proceed to obtain the o/p boolean functions (or) the truth table.

To obtain the o/p boolean functions from a logic diagram, we proceed as follows.

- ① Label all gate outputs ^{that are a function of input variables} with arbitrary symbols and determine the boolean functions for each gate o/p.
- ② Label the gates that are a function of input variables and previously labeled gates with other arbitrary symbols. Find the boolean functions for these gates.
- ③ Repeat the process outlined in step ② until the o/p's of the circuit are obtained.

Ex:- Analyse the following given circuit.



Sol: From the diagram

(3)

$$T_1 = A+B+C$$

$$T_2 = ABC$$

$$F_2 = AB+AC+BC$$

Next, we consider outputs of gates that are a function of already defined symbols

$$T_3 = T_1 F_2'$$

$$F_1 = T_3 + T_2$$

To obtain F_1 as a function of A, B, C , we substitute expressions of previously defined symbols. we substitute expressions of we are substituting

$$\begin{aligned} F_1 &= ABC + (A+B+C)(AB+AC+BC)' \\ &= ABC + (A+B+C)[(AB)'(AC)'(BC)'] \\ &= ABC + (A+B+C)[(A'+B')(A'+C')(B'+C')] \\ &= ABC + (A+B+C)[(A'+B')(A'B' + A'C' + B'C' + C')] \\ &= ABC + (A+B+C)[A'B' + A'C' + A'B'C' + A'C' + A'B' + A'B'C' + B'C' + B'C'] \\ &= ABC + (A+B+C)[A'B' + A'C' + A'B'C' + B'C'] \\ &= ABC + AA'B' + AA'C' + AA'B'C' + AB'C' + A'B'B + A'BC' + A'BB'C' + BB'C' + A'B'C + A'C'C + A'B'C'C + B'CC \\ F_1 &= ABC + AB'C' + A'BC' + A'B'C \\ F_1 &= \Sigma(1, 2, 4, 7) \end{aligned}$$

Now let us express the boolean functions in truth table form.

A	B	C	T_1	T_2	T_3	F_2	F_2'	F_1
0	0	0	0	0	0	0	1	0
0	0	1	1	0	1	0	1	1
0	1	0	1	0	1	0	1	1
0	1	1	1	0	0	1	0	0
1	0	0	1	0	1	0	1	1
1	0	1	1	0	0	1	0	0
1	1	0	1	0	0	1	0	0
1	1	1	1	1	0	1	0	1

Design procedure:-

The design of combinational circuits starts from the specification of the design objective and culminates in a logic diagram or a set of boolean functions from which the logic diagram can be obtained. The procedure involves the following steps.

- ① From the specifications of the circuits, determine the required number of inputs and outputs and assign a symbol to each.
- ② Define the truth table that defines the required relationship b/w inputs and outputs.
- ③ Obtain the simplified boolean functions for each o/p as a function of the input variables.
- ④ Draw the logic diagram and verify the correctness of the design.

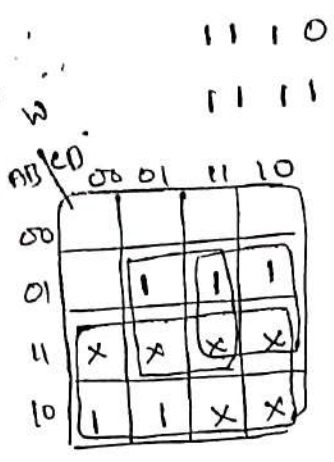
A truth table for a combinational circuit consists of i/p columns and o/p columns. The input columns are obtained from the 2^n binary numbers for the n input variables. The binary values for the o/p's are determined from the stated specifications.

Ex:- Design a logic circuit that converts a BCD code into an Excess-3 code.

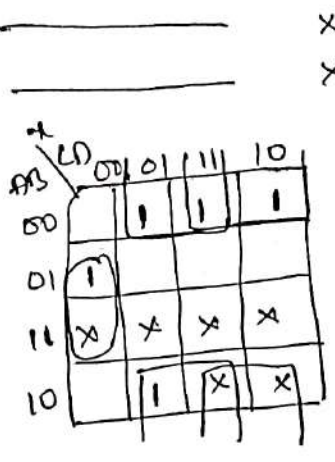
Sol:-

inputs (BCD)				Outputs (Excess-3 code)			
A	B	C	D	w	x	y	z
0	0	0	0	0	0	1	1
0	0	0	1	0	1	0	0
0	0	1	0	0	1	0	1
0	0	1	1	0	1	1	0
0	1	0	0	0	1	1	1
0	1	0	1	1	0	0	0
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	0
1	0	0	0	1	0	1	1
1	0	0	1	1	1	0	0
1	0	1	0	x	x	x	x
1	0	1	1	x	x	x	x
1	1	0	0	x	x	x	x
1	1	0	1	x	x	x	x

5



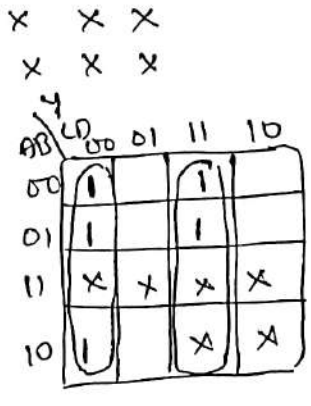
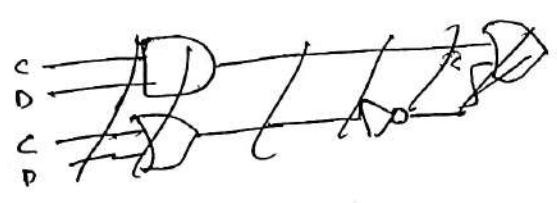
$w = A + BD + BC$



$x = BC'D' + B'D + B'C$

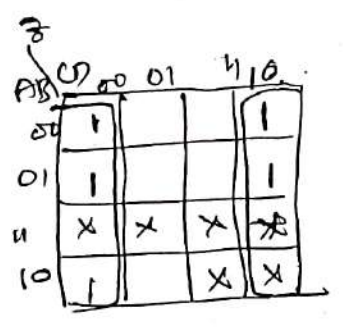
$z = A + B(C+D)$
 $x' = B'(C+D) + B(C+D)'$
 $= B \oplus (C+D)$

Logic diagram

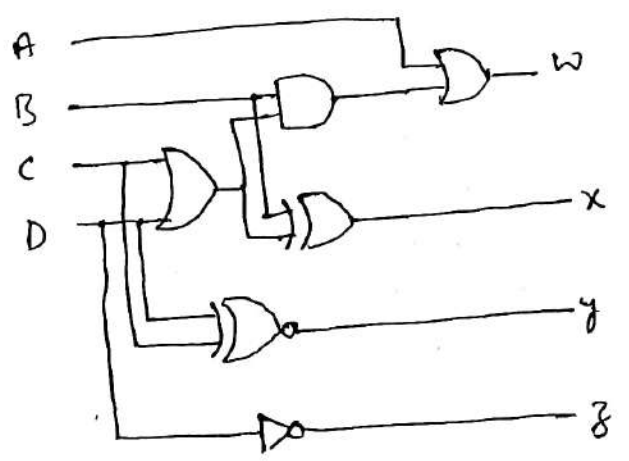


$y = C'D' + CD$

$y = C'D' + CD$
 $= (C \oplus D)'$



$z = D'$



Binary Adder-subtractor:-

Digital computers perform a variety of arithmetic operations. The most basic arithmetic operation is the addition of two binary digits. The simple addition consists of 4 possible elementary operations

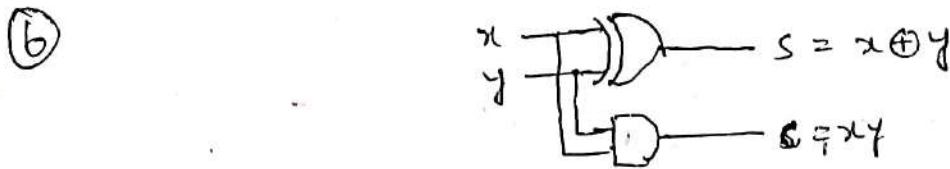
- $0+0=0$
- $0+1=1$
- $1+0=1$
- $1+1=10$

Half adder:- It is a combinational circuit that performs the addition of two one bit inputs and it produces two outputs: sum and carry. We assign symbols x and y to the two inputs and S (for sum) and C (for carry) to the outputs.

x	y	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

From truth table
 $S = x'y + xy'$
 $= x \oplus y$
 $C = xy$

The half adder circuit is implemented as follows

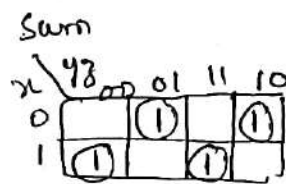


Full adder:-

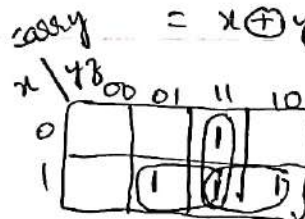
Full adder is a combinational circuit that performs addition on 3 one-bit inputs and it produces two outputs: sum and carry. Two of the input variables denoted by x and y , represent two significant bits to be added. The 3rd input, z represents the carry from the previous lower significant position.

The truth table of full adder is as follows.

x	y	z	Sum	Carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



$$\begin{aligned}
 \text{Sum} &= x'y'z + x'y z' + xy'z' + xy z \\
 &= x'(y'z + y z') + x(y'z' + y z) \\
 &= x'(y \oplus z) + x(y \oplus z)' \\
 &= x \oplus y \oplus z
 \end{aligned}$$



$$\text{Carry} = yz + xz + xy$$

The logic diagram for the full adder implemented in sum of products form is shown in fig.

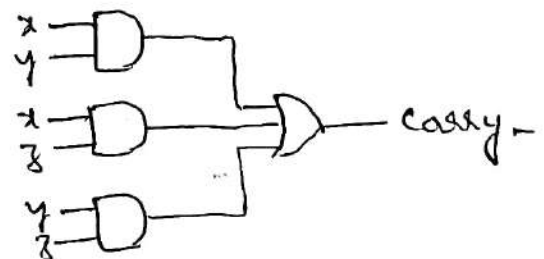
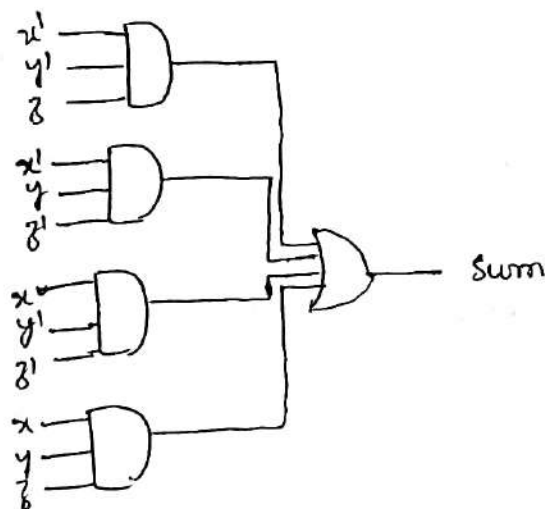
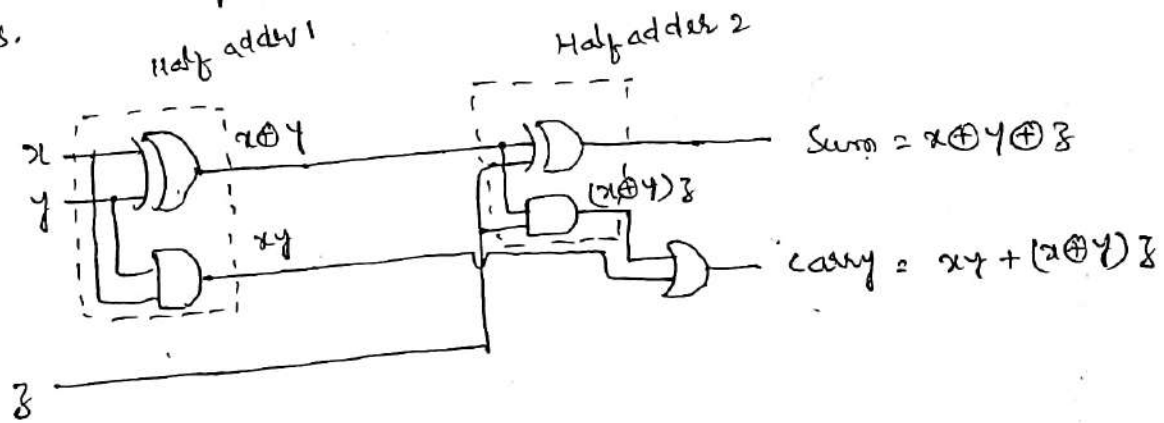


Fig. Implementation of full adder in sum-of-products form

It can also be implemented with two half adders and one OR gate as follows.



Half Subtractor:-

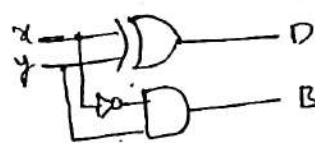
Half Subtractor is a combinational circuit that has two inputs and two outputs. The 2 inputs x and y form the minuend and the subtrahend. The two OP's are difference (D) and borrow (B).

x	y	D	B
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Truth table

$$D = x'y + xy' = x \oplus y$$

$$B = x'y$$



This circuit performs " $x-y$ " operation

Fig:- Half subtractor

Full Subtractor:-

A full subtractor has 3 inputs and two outputs. The inputs are designated by x, y and z and the outputs are D (Difference) and B (borrow). The following table explains the functionality of full subtractor

x	y	z	D	B
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

x	y	z	D
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

$$D = x'y'z + x'y'z' + x'y'z' + x'yz$$

$$= x'(y'z + y'z') + x'(y'z' + yz)$$

x	y	z	B
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

$$B = x'z + x'y + yz$$

Q6

The logic diagram for the full ~~adder~~ ^{subtractor} implemented in sum of products form is as shown in fig.

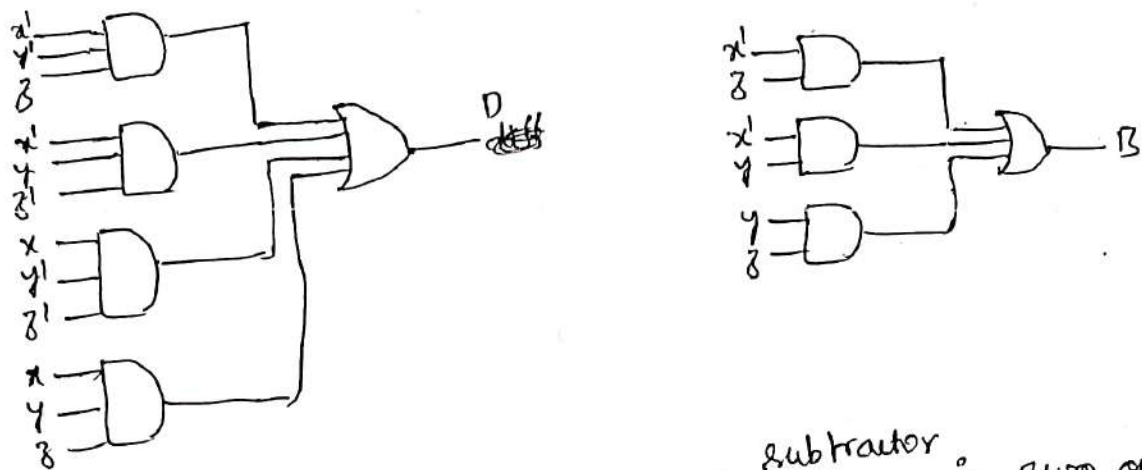
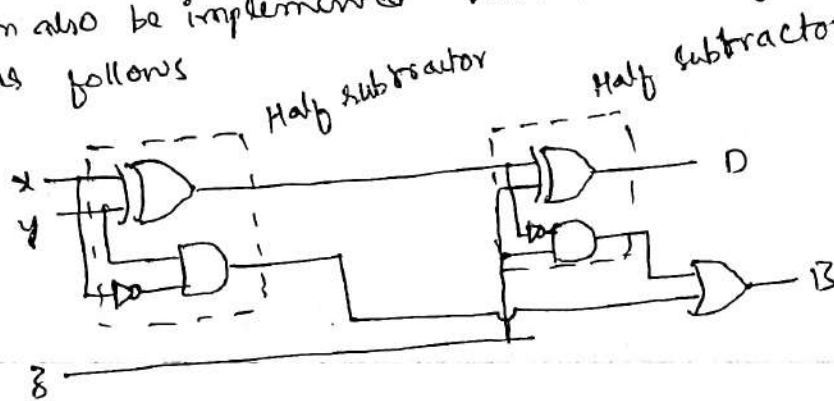


fig:- Implementation of full ~~adder~~ ^{subtractor} in sum of products form

It can also be implemented with two half subtractors and one OR gate as follows



Binary adder (or) parallel adder (or) Ripple Carry adder:-

9

A binary adder is a digital ckt that produces the arithmetic sum of two binary numbers. It can be constructed with full adders connected in cascade, with the o/p carry from each full adder connected to the i/p carry of the next full adder in the chain.

Following fig shows the interconnection of four full adder circuits to provide a 4-bit binary ripple carry adder.

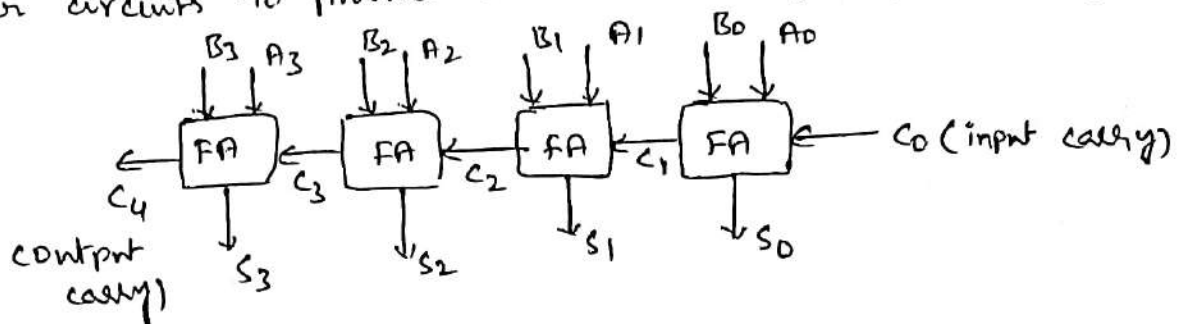


fig:- 4-bit adder

The input carry to the adder is C_0 and it ripples through the full adders to the o/p carry C_4 . The S outputs generate the required sum bits. An n -bit adder requires n -full adders with each o/p carry connected to the i/p carry of the next higher order full adder.

Let us consider the two binary numbers $A = 1011$ and $B = 0011$. Their sum $S = 1110$ is formed with the 4-bit adder as follows.

Augend (A)	1 0 1 1
Addend (B)	0 0 1 1
input carry	0 1 1 0
	1 1 1 0
Sum (S) -	1 1 1 0
o/p carry -	0 0 1 1

→ The sum bits are generated starting from the right most position and are available as soon as the corresponding previous carry bit is generated. All the carries must be generated for the correct sum bits to appear at the o/p's.

Look ahead carry generator:-

The parallel adder is ripple carry type in which the carry o/p of each full adder stage is connected to the carry i/p

(10) of the next higher order stage. Therefore, the sum and carry out of any stage cannot be produced until the input carry occurs. This leads to a time delay in the addition process. This delay is known as carry propagation delay. It can be explained by considering the following addition

$$\begin{array}{r} 0101 \\ + 0011 \\ \hline 1000 \end{array}$$

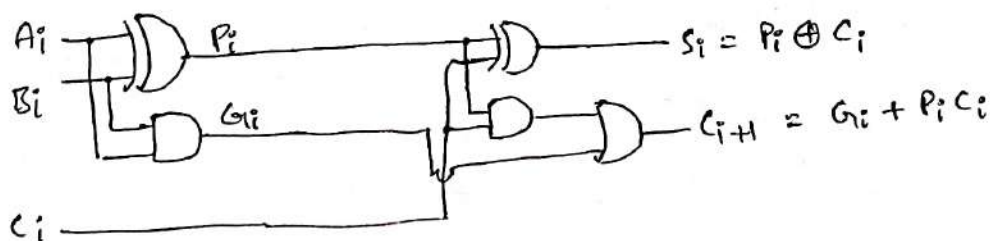
Addition of the LSB position produces a carry into the second position. This carry, when added to the bits of the second position produces a carry into the 3rd position. The latter carry, when added to the bits of the 3rd position, produces a carry into the last position. The key thing to notice is that the sum bit generated in the last position depends on the carry that was generated by the addition in the previous positions. This means that, adder will not produce correct result until LSB carry has propagated through the intermediate full-adders.

This represents a time delay that depends on the propagation delay produced in an each full adder. For example, if each full adder is considered to have

An obvious solution for reducing the carry propagation delay time is to employ faster gates with reduced delays.

Another solution is to increase the complexity of the circuit in such a way that the carry delay time is reduced. The most widely used technique employs the principle of carry lookahead logic.

Consider the circuit of the full adder as shown in fig.



Here we define two new binary variables $P_i = A_i \oplus B_i$
 $G_i = A_i B_i$

the old sum and carry can be expressed as

$$S_i = P_i \oplus C_i$$

$$C_{i+1} = G_i + P_i C_i$$

- ① G_i is called a carry generate and it produces a carry of "1" when both A_i and B_i are "1", regardless of the i/p carry.
 P_i is called a carry propagate because it determines whether a carry in stage "i" will propagate into stage i+1.

Now we write the boolean functions for the carry outputs of each stage as follows.

$$C_0 = \text{input carry}$$

$$C_1 = G_0 + P_0 C_0$$

$$C_2 = G_1 + P_1 C_1 = G_1 + P_1 (G_0 + P_0 C_0) \Rightarrow G_1 + P_1 G_0 + P_1 P_0 C_0$$

$$C_3 = G_2 + P_2 C_2 = G_2 + P_2 (G_1 + P_1 G_0 + P_1 P_0 C_0) \\ = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_0$$

Since the Boolean function for each o/p carry is expressed in sum of products form, each function can be implemented with one level of AND gates followed by an OR gate. The 3 boolean functions for C_1 , C_2 and C_3 are implemented in the carry lookahead generator as shown in fig.

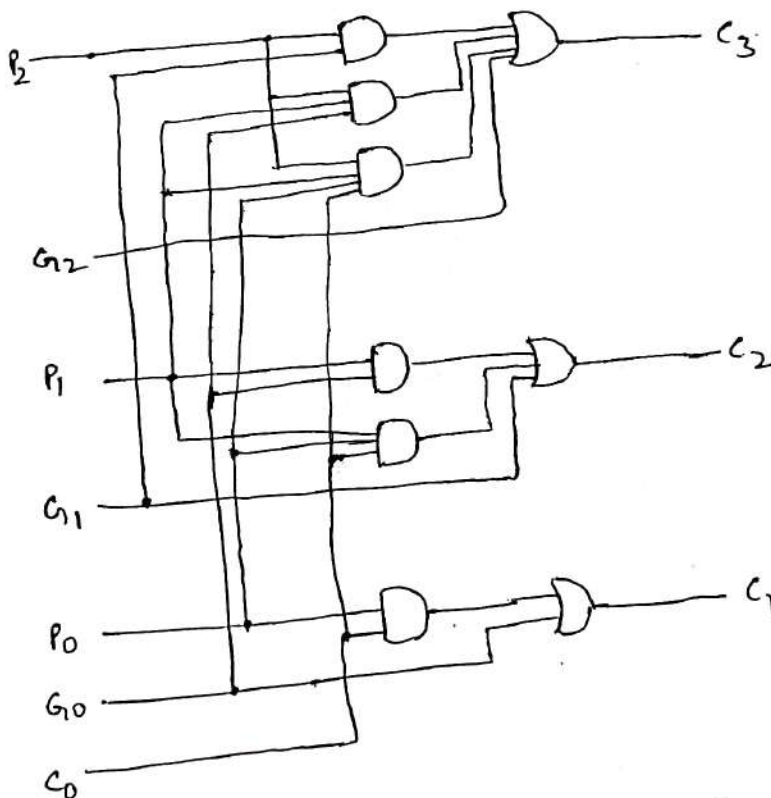


Fig:- Logic diagram of carry look ahead generator

(12)

Note that this circuit can add in less time because c_3 does not have to wait for c_2 and c_1 to propagate. In fact, c_3 is propagated at the same time as c_1 and c_2 . This gain in speed of operation is achieved at the expense of additional complexity.

The construction of a four-bit adder with a carry lookahead scheme is as shown in fig.

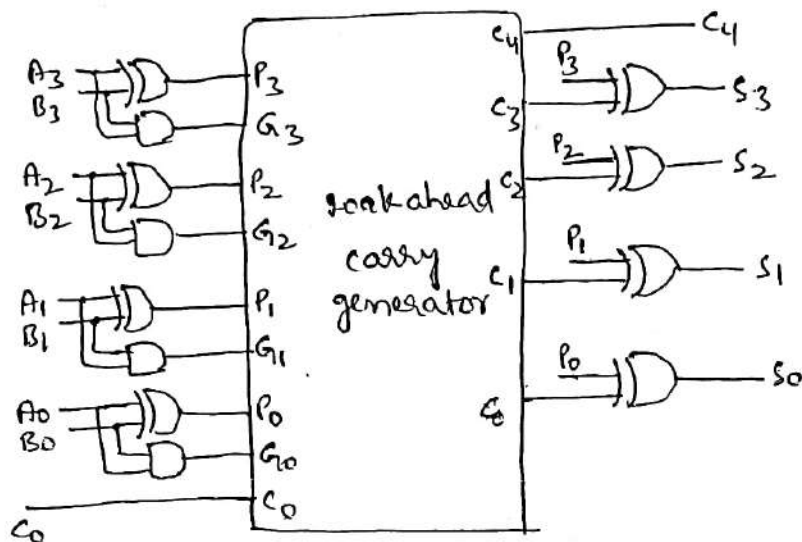
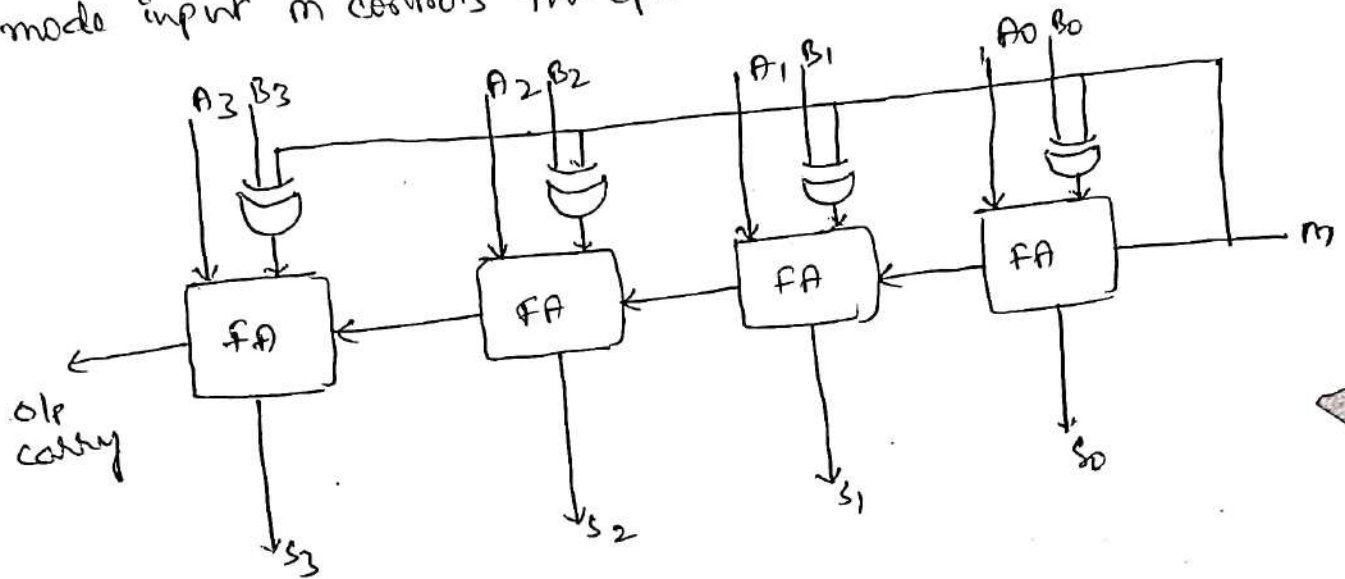


fig:- four bit adder with carry lookahead logic

Binary adder-subtractor:-

The subtraction $A-B$ can be performed by taking 2's complement of B and adding it to A . The 2's complement can be obtained by taking 1's complement and adding '1' to LSB of 1's complement. The 1's complement can be implemented with inverters. Thus $A-B = A + 1's \text{ complement of } B + 1$.

A 4-bit adder-subtractor circuit is shown in fig. the mode input m controls the operation.



When $m=0$, we have $B \oplus 0 = B$. The full adders receive the value of B , the old carry is '0' and the circuit performs $A+B$.
When $m=1$, we have $B \oplus 1 = B'$ and old carry is '1'. The B inputs are all complemented and '1' is added through the old carry. The circuit performs the operation $A+B'+1$ ($A + 2's \text{ complement of } B$).

Decimal adder:-

13) Computers (as) calculators that perform arithmetic operations directly in the decimal number system represent decimal numbers in binary coded form. An adder for such a computer must employ arithmetic circuits that accept coded decimal numbers and present results in the same code.

For binary addition, it is sufficient to consider a pair of significant bits together with a previous carry.

A decimal adder requires a min. of 9 inputs and five outputs, since 4 bits are required to code each decimal digit and the circuit must have an input and output carry. There is a wide variety of possible decimal adder circuits, depending upon the code used to represent the decimal digits. Here we examine a decimal adder for the BCD code.

BCD adder:- (8421 adder)

Consider the arithmetic addition of two decimal digits in BCD, together with an i/c carry from a previous stage. Since each input digit does not exceed 9, the o/p sum cannot be greater than $9+9+1=19$, the "1" in the sum being an i/c carry. Suppose we apply two BCD digits to a 4-bit binary adder. The adder will form the sum in binary and produce a result that ranges from 0 to 19. These binary numbers are listed in Table 1. and are labeled by symbols K, Z_8, Z_4, Z_2 and Z_1 . K is carry and the subscripts under 'Z' represent the weights 8, 4, 2 and 1 that can be assigned to the 4 bits in BCD code.

The columns under the binary sum list the binary value that appears in the o/p's of 4-bit binary adder. The o/p sum of two decimal digits must be represented in BCD and should appear in the form listed in the columns under "BCD sum".

The problem here is to find a rule by which the binary sum is converted to the correct BCD digit.

In examining the contents of the table, it is found that when the binary sum is equal to or less than 1001, the corresponding BCD number is identical and therefore no correction is needed. When binary sum is greater than 1001, we obtain an invalid BCD

(14)

Table 1

Binary Sum					BCD Sum					Decimal
K	z_8	z_4	z_2	z_1	C	s_8	s_4	s_2	s_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

representation. The addition of binary 6 (0110) to the binary sum converts it to the correct BCD representation and also produces an o/p carry.

From the table, it is found that a correction is needed when the binary sum has an o/p carry $K=1$. The other six combinations from 1010 through 1111 that need a correction have a 1 in position z_8 , z_4 and z_2 have a 1.

The condition for a correction and an o/p carry can be expressed by

$$C = K + z_8 z_4 + z_8 z_2$$

When $C=1$, it is necessary to add 0110 to the binary sum and provide an o/p carry for the next stage.

Now let us see a BCD adder that adds two BCD digits and produces a sum digit in BCD as shown in fig.

15

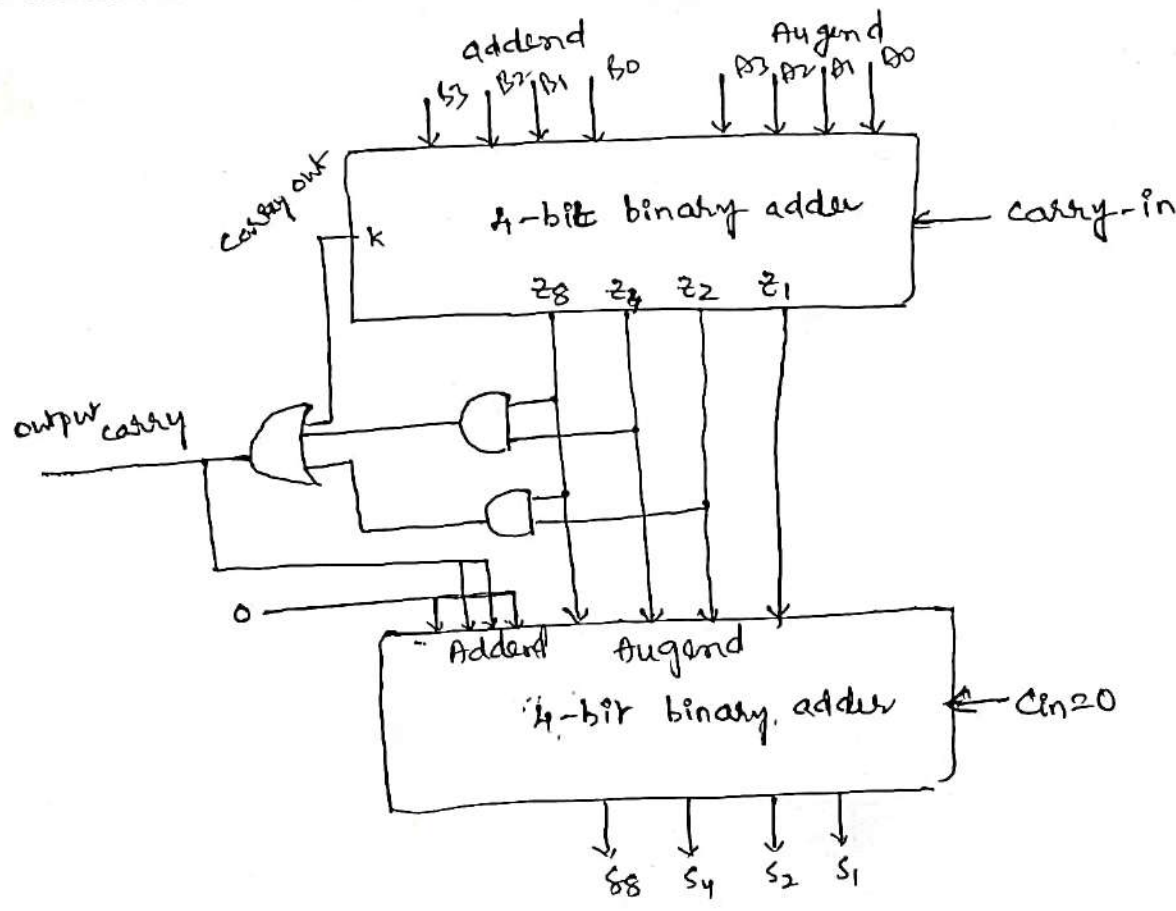


Fig:- Block diagram of a BCD adder

Binary multiplier:-

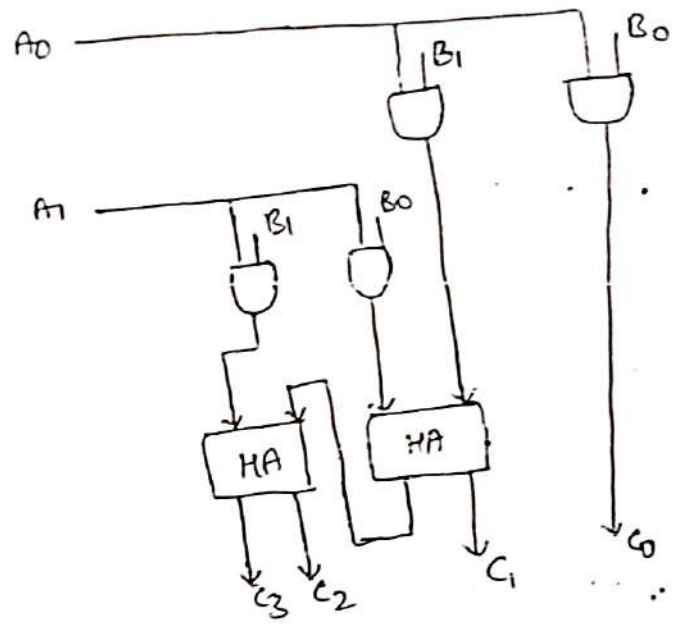
Multiplication of binary numbers is performed in the same way as multiplication of decimal numbers. The multiplicand is multiplied by each bit of the multiplier, starting from the LSB. Each such multiplication forms a partial product. Successive partial products are shifted one position to the left. The final product is obtained from the sum of the partial products.

Consider the multiplication of two 2-bit numbers as shown in fig. The multiplicand bits are B_1 and B_0 , the multiplier bits are A_1 and A_0 and the product is $C_3 C_2 C_1 C_0$. The first partial product is formed by multiplying $B_1 B_0$ by A_0 . The multiplication of two bits such as A_0 and B_0 produces a '1' if both bits are '1' otherwise, it produces '0'. This is identical to an AND operation. Therefore, the partial product can be implemented with AND gates as shown in fig. The second partial product is formed by multiplying $B_1 B_0$ by A_1 and shifting one

$$\begin{array}{r}
 B_1 B_0 \\
 \times A_1 A_0 \\
 \hline
 B_1 A_0 \quad B_0 A_0 \\
 B_1 A_1 \quad B_0 A_1 \\
 \hline
 C_3 \quad C_2 \quad C_1 \quad C_0
 \end{array}$$

16

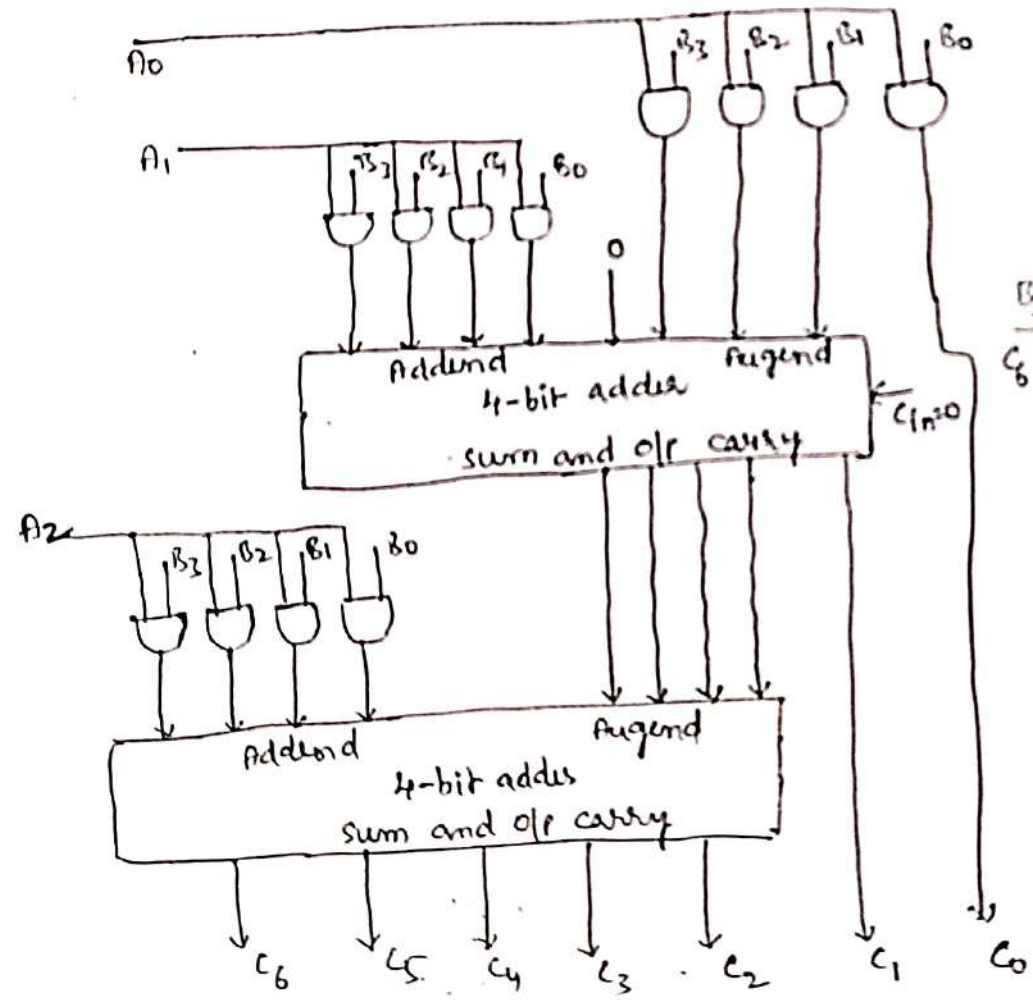
position to the left. The two partial products are added with two half adder (HA) circuits as shown in fig.



For "J" multiplier bits and "K" multiplicand bits, we need (JK) AND gates and (J-1) K-bit adders to produce a product of J+K bits.

fig:- Two bit by two bit multiplier

As a second example, consider a multiplier circuit that multiplies a binary number represented by 4 bits by a number represented by 3 bits. Let the multiplicand be $B_3B_2B_1B_0$ and the multiplier by $A_2A_1A_0$. Since $J=3$ and $K=4$, we need 12 AND gates and 2 4-bit adders to produce a product of seven bits. The logic diagram of the multiplier is as shown in fig.



$$\begin{array}{r}
 B_3 B_2 B_1 B_0 \\
 \times A_2 A_1 A_0 \\
 \hline
 B_3 A_0 \ B_2 A_0 \ B_1 A_0 \ B_0 A_0 \\
 B_3 A_1 \ B_2 A_1 \ B_1 A_1 \ B_0 A_1 \\
 B_3 A_2 \ B_2 A_2 \ B_1 A_2 \ B_0 A_2 \\
 \hline
 C_6 \ C_5 \ C_4 \ C_3 \ C_2 \ C_1 \ C_0
 \end{array}$$

fig:- 4-bit by 3-bit binary multiplier

Exclusive-OR function:-

The exclusive-OR is equal to 1 if only x is equal to 1

(b) if only 'y' is equal to '1' but not when both are equal to '1' (c) when both are equal to '0'. The exclusive-OR is denoted by \oplus . It performs the following boolean operation

$$x \oplus y = xy' + x'y$$

The following identities apply to the exclusive-OR operation

$$x \oplus 0 = x$$

$$x \oplus 1 = x'$$

$$x \oplus x = 0$$

$$x \oplus x' = 1$$

$$x \oplus y' = x' \oplus y = (x \oplus y)'$$

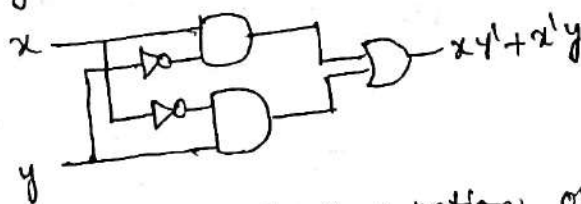
The exclusive-OR operation is both commutative and associative i.e.

$$x \oplus y = y \oplus x$$

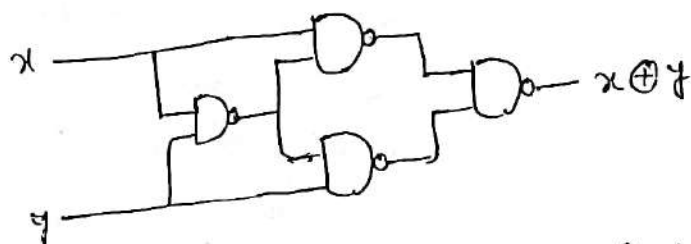
$$\text{and } (x \oplus y) \oplus z = x \oplus (y \oplus z) = x \oplus y \oplus z$$

This means that the two inputs to an exclusive-OR gate can be interchanged without affecting the operation.

A two input exclusive OR gate is constructed with conventional gates using two inverters, two AND gates and an OR gate as shown in fig.



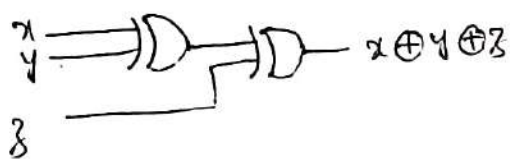
The following fig shows the implementation of exclusive OR with four NAND gates.



→ Exclusive OR function is equal to 1 when it has odd no. of '1's at its IP - otherwise the functional value is equal to 0 so that

3/9
18

it is called odd function
the 3-input odd function is implemented by means of two input XOR gates as shown in fig.



Parity generation and checking:-

Exclusive-OR functions are very useful in systems requiring error detection and correction codes. A parity bit is used for the purpose of detecting errors during the transmission of binary information.

A parity bit is an extra bit included with a binary message to make the no. of 1's either odd or even. The message including the parity bit is transmitted and then checked at the receiving end for errors. An error is detected if the checked parity does not correspond with the one transmitted.

Parity generator:- The circuit that generates the parity bit in the transmitter is called a parity generator.

Parity checker:- The circuit that checks the parity in the receiver is called a parity checker.

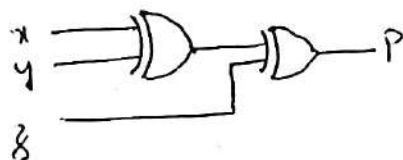
for example, consider a three-bit message to be transmitted together with an even parity bit. The following truth table ~~shows~~ is for parity generator.

x	y	z	P
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

The 3 bits - x, y, z constitute the message and are the inputs to the circuit. The parity bit P is the O/P. For even parity, the bit P must be generated to make the total no. of 1's even.

$$\therefore P = \cancel{x \oplus y} \oplus z$$

The logic diagram for the parity generator is shown in fig



(a) 3-bit even parity generator

3/10
19

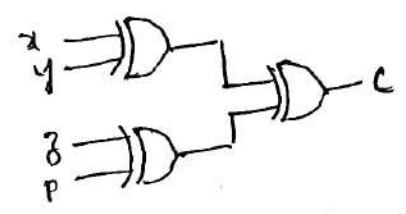
The three bits in the message together with the parity bit are transmitted to their destination where they are applied to a parity checker circuit to check for possible errors in the transmission.

Since the information was transmitted with even parity, the four bits received must have an even no. of 1's. The output of the parity checker, denoted by c , will be equal to 1 if an error occurs i.e. if the four bits received have an odd no. of 1's.

The following table is the truth table for even parity checker.

x	y	z	p	c
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

The parity checker can be implemented with XOR gates
 $c = x \oplus y \oplus z \oplus p$



4-bit even parity checker.

20 Magnitude Comparator:-

The comparison of two numbers is an operation that determines whether one no. is greater than, less than (&) equal to the other number.

A magnitude comparator is a combination circuit that compares two numbers A and B and determines their relative magnitudes. The outcome of the comparison is specified by 3 binary variables that indicate whether $A > B$, $A = B$ (&) $A < B$.

Consider two numbers A and B with four bits each. i.e.

$$A = A_3 A_2 A_1 A_0 \quad \& \quad B = B_3 B_2 B_1 B_0$$

The two numbers are equal if all pairs of significant digits are equal. i.e. $A_3 = B_3$, $A_2 = B_2$, $A_1 = B_1$ and $A_0 = B_0$

Each bit is either 1 (&) 0, and the equality of each pair of bits can be expressed logically with an EX-NOR function as

$$x_i = A_i B_i + A_i' B_i' \quad \text{where } i = 0, 1, 2, 3$$

where $x_i = 1$ only if the pair of bits in position i are equal.

$$(A=B) = (A_3=B_3) \text{ and } (A_2=B_2) \text{ and } (A_1=B_1) \text{ and } (A_0=B_0) \\ = x_3 x_2 x_1 x_0$$

To determine whether A is greater (&) less than B, we inspect the relative magnitudes of pairs of significant digits starting from MSB. If MSBs are equal, we compare the next lower significant pair of bits. The comparison continues until a pair of unequal bits is reached.

If the corresponding bit of A is 1 and that of B is 0, we conclude that $A > B$. If the corresponding bits of A is 0 and that of B is 1, we have $A < B$.

The boolean functions for $A < B$ and $A > B$ are as follows.

$$(A < B) = A_3' B_3 + x_3 A_2' B_2 + x_3 x_2 A_1' B_1 + x_3 x_2 x_1 A_0' B_0$$

$$(A > B) = A_3 B_3' + x_3 A_2 B_2' + x_3 x_2 A_1 B_1' + x_3 x_2 x_1 A_0 B_0'$$

Following fig shows the logic diagram of 4-bit magnitude comparator.

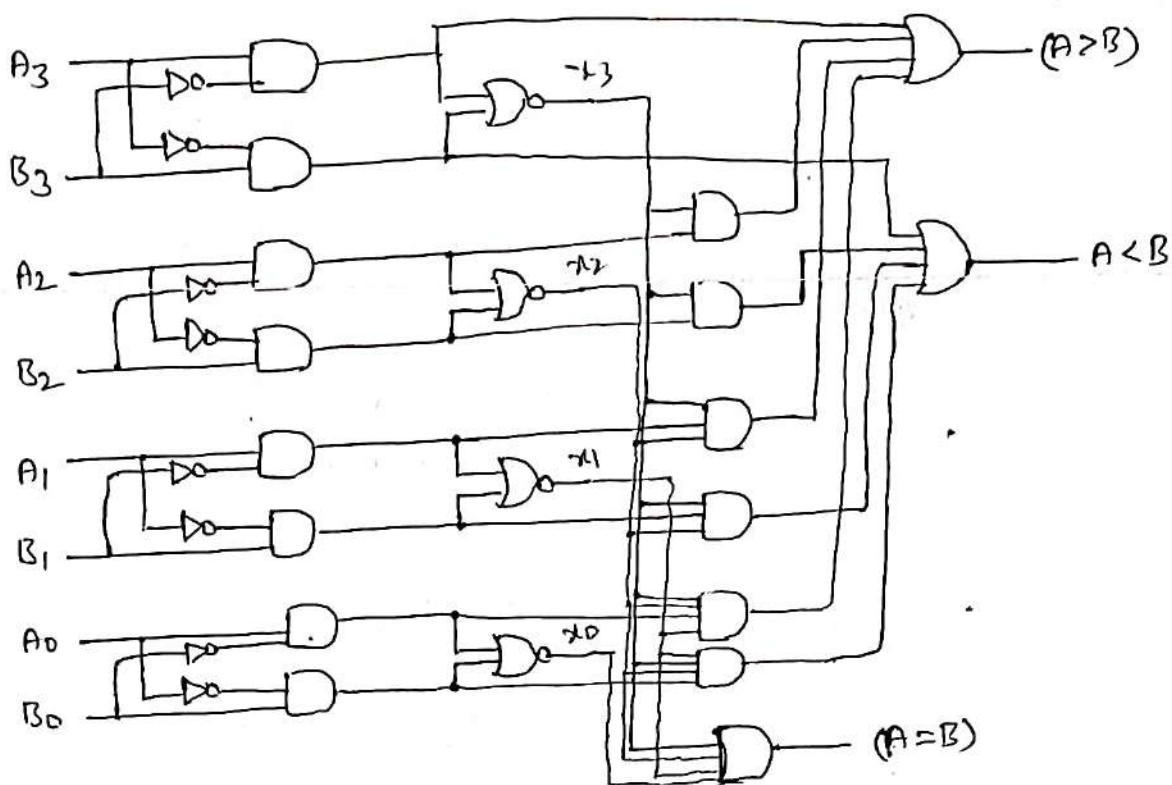


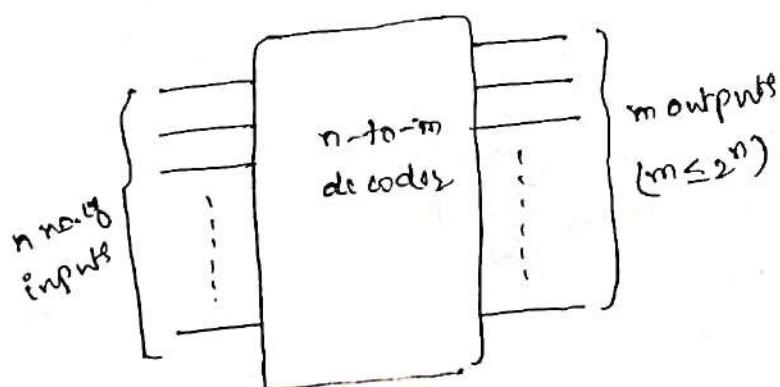
fig:- 4-bit magnitude comparator

Decoders:-

A decoder is a combinational circuit that converts binary ~~input~~ information from n input lines to a max. of 2^n unique ~~old~~ lines.

If the n -bit coded information has unused combinations the decoder may have fewer than 2^n outputs.

The decoder presented here is called n to m line decoder where $m \leq 2^n$ as shown in fig.



The main purpose of decoder is to generate the 2^n (fewer) minterms of n input variables.

For example, consider 3 to 8 line decoder circuit as shown in fig.

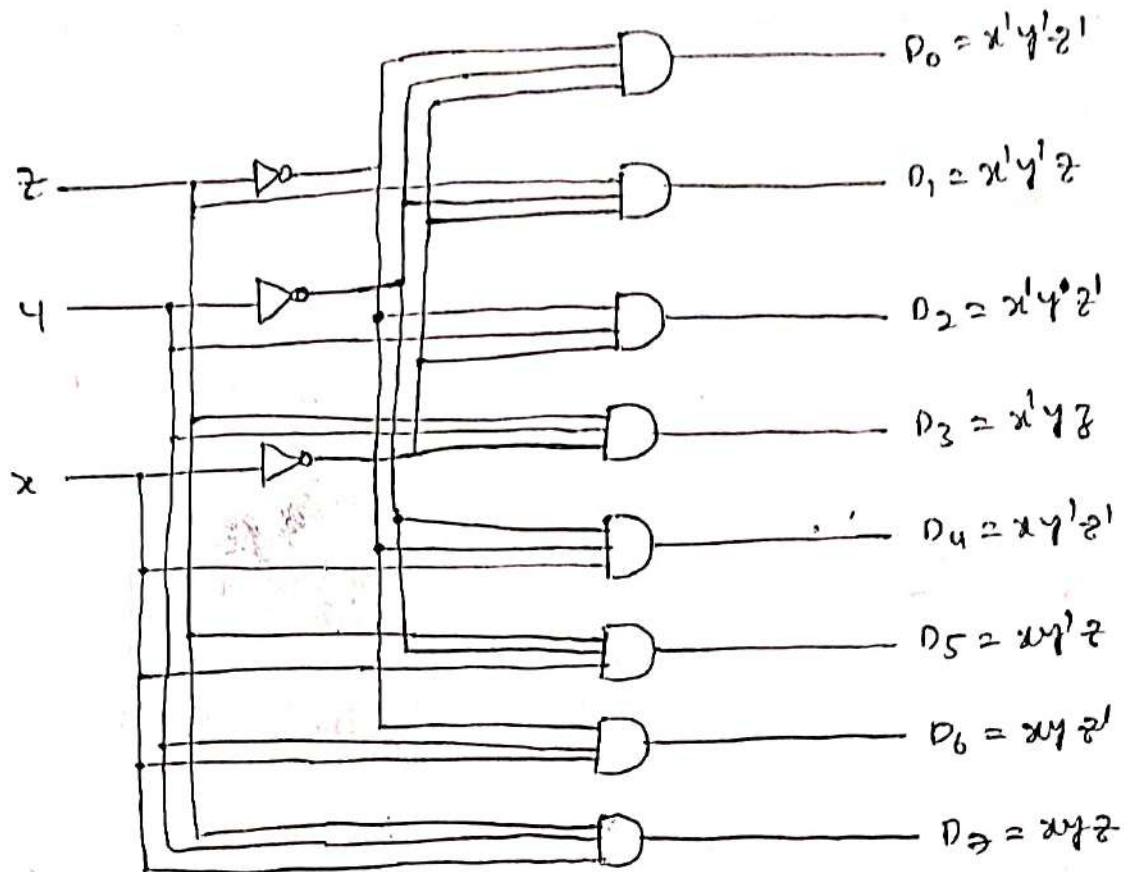


fig. 3-to-8 line decoder

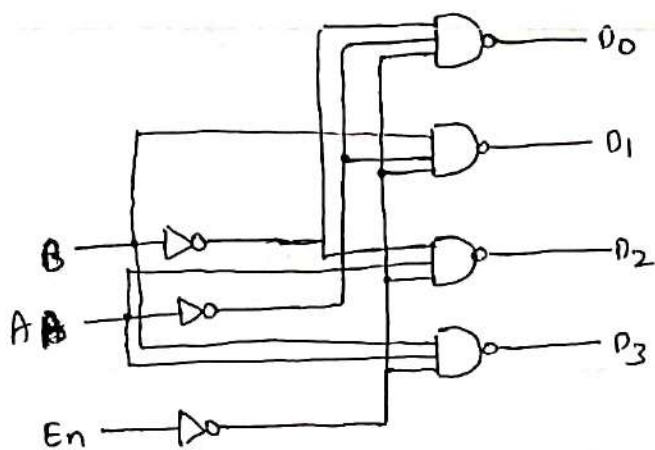
The 3 inputs are decoded into 8 o/p's, each representing one of the minterms of the 3 input variables. A particular application of this decoder is binary to octal conversion.

The operation of the decoder may be clarified by the truth table. For each possible i/p combination, there are 7 o/p's that are equal to "0" and only one that is equal to '1'. The o/p whose value is equal to '1' represents the minterm equivalent of the binary number currently available in the i/p lines.

inputs			output							
x	y	z	P ₀	P ₁	P ₂	P ₃	P ₄	P ₅	P ₆	P ₇
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

23

Furthermore, decoders include one or more enable inputs to control the circuit operation. Some decoders are constructed with NAND gates. A 2-to-4 line decoder with an enable input constructed with NAND gates is as shown in fig. The circuit operates with complemented o/p's and a complement enable input.



En	A	B	D ₀	D ₁	D ₂	D ₃
1	X	X	1	1	1	1
0	0	0	0	1	1	1
0	0	1	1	0	1	1
0	1	0	1	1	0	1
0	1	1	1	1	1	0

(b) Truth table

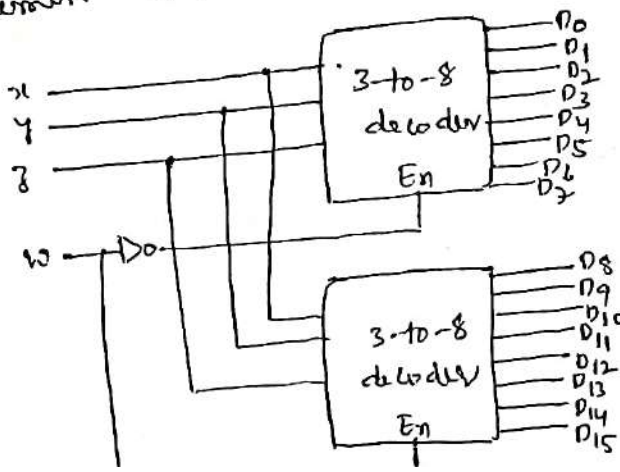
(a) 2-to-4 line decoder with enable i/p.

The decoder is enabled when E is equal to '0' (i.e. active low enable). As indicated by the truth table, only one o/p can be equal to 0 at any given time, all other o/p's equal to 1. The o/p whose value is equal to 0 represents the minterm selected by inputs A and B. The circuit is disabled when E is equal to '1' regardless of the values of other two inputs.

In general, a decoder may operate with complemented or (b) uncomplemented o/p's.

Note:- If decoders are constructed with AND gates, we use active high enable inputs. If decoders are constructed with NAND gates, we use active low enable i/p's.

Ex:- Implement 4 to 16 decoder with 3 to 8 decoder



Decoders with enable inputs can be connected together to form a larger decoder circuit. Fig shows two 3 to 8 line decoders with enable inputs connected to form a 4 to 16 line decoder.

② When $w=0$, the top decoder is enabled and the other is disabled. The bottom decoder o/p's are all 0's and the top eight o/p's generate minterms 0000 to 0111. When $w=1$, the enable conditions are reversed.

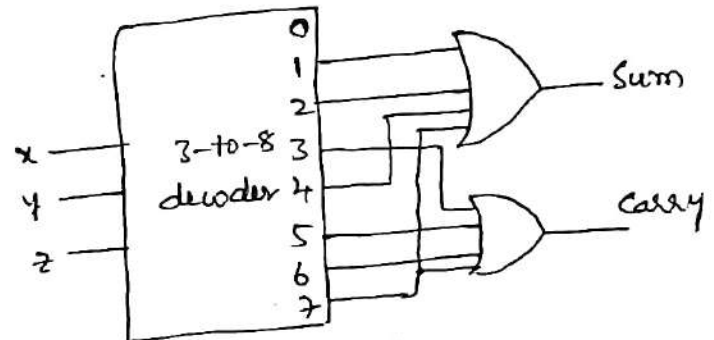
Ex:- Design a full adder circuit using decoder and logic gates.

Sol:- Truth table of Full adder

inputs			outputs	
x	y	z	sum	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$$\text{Sum} = \sum m(1, 2, 4, 7)$$

$$\text{Carry} = \sum m(3, 5, 6, 7)$$

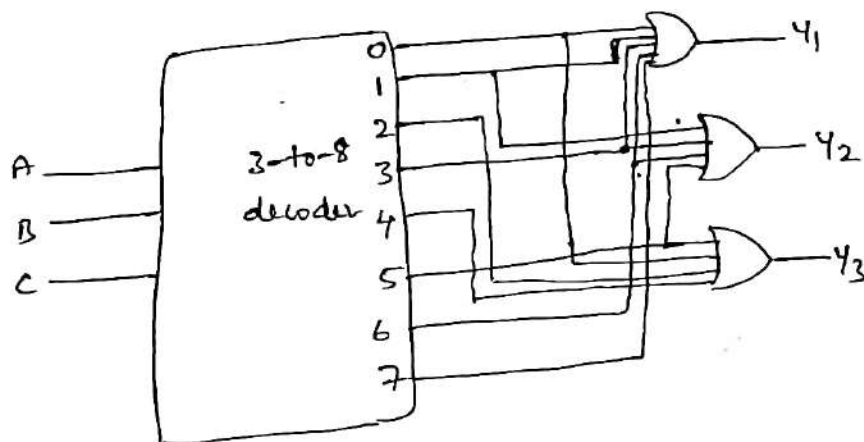


② Implement the following function using decoder logic
 $y_1 = \sum (0, 1, 3, 6, 7)$ $y_2 = \pi(0, 2, 4, 7)$ $y_3 = \pi(1, 3, 6, 7)$

Sol:-

$$y_1 = \sum (0, 1, 3, 6, 7) \quad y_3 = \sum (0, 2, 4, 5)$$

$$y_2 = \sum (1, 3, 5, 6)$$



② Encoder :-

An encoder is a digital circuit that performs the inverse operation of a decoder. An encoder has 2^n (or fewer) input lines and n output lines. The output lines generate the binary code corresponding to the input value.

For example, consider a 8-to-3 encoder whose truth table is given in Table. It has 8 inputs and 3 outputs that generate the corresponding binary number. It is assumed that only one input has a value of '1' at any given time.

inputs								outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

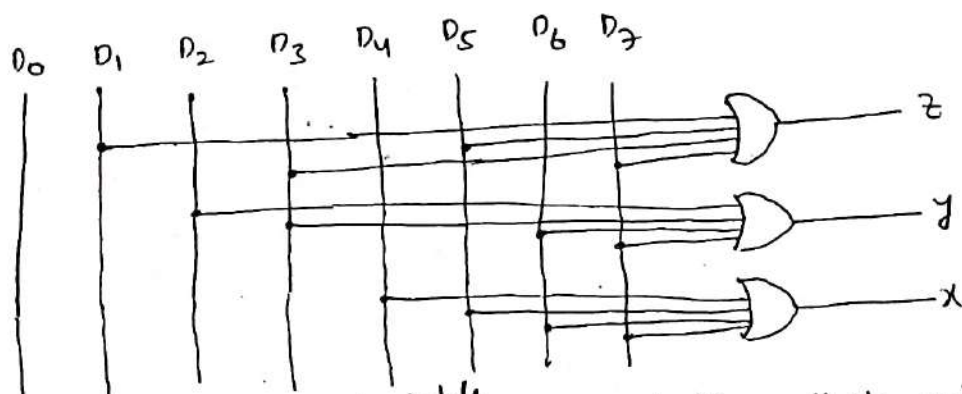
From the truth table, the boolean functions for output variables are

$$x = D_4 + D_5 + D_6 + D_7$$

$$y = D_2 + D_3 + D_6 + D_7$$

$$z = D_1 + D_3 + D_5 + D_7$$

The encoder can be implemented with OR gates.



The encoder defined in table has the limitation that only one input can be active at any given time. If two inputs are active simultaneously, the output produces an undefined combination. For example, if D_3 and D_6 are 1 simultaneously, the output of the encoder will be 111. The output 111 does not represent either binary 3 (or) binary 6.

26 To resolve this ambiguity, encoder circuits must establish an input priority to ensure that only one i/p is encoded. If we establish a higher priority for inputs with higher subscript numbers, and if both D_3 and D_6 are 1 at the same time, the o/p will be 110 because D_6 has higher priority than D_3 .

Another ambiguity in the 8-to-3 encoder is that an o/p with all 0's is generated when all the i/p's are zero. But this o/p is same as when D_0 is equal to 1. This can be resolved by providing one more o/p to indicate whether at least one i/p is equal to '1'.

Priority Encoder:-

A priority encoder is an encoder circuit that includes the priority function. The operation of the priority encoder is such that if two or more inputs are equal to '1' at the same time, the i/p having the highest priority will take precedence. The truth table of a four-input priority encoder is given in table. In addition to the two o/p's x and y , the circuit has a third o/p designated by v . This is a valid bit indicator that is set to 1 when one or more inputs are equal to 1. If all inputs are 0, there is no valid i/p and v is equal to 0. The other two o/p's are not inspected when v equals '0' and are specified as don't care conditions.

Inputs				Outputs		
D_3	D_2	D_1	D_0	x	y	v
0	0	0	0	x	x	0
1	x	x	x	1	1	1
0	1	x	x	1	0	1
0	0	1	x	0	1	1
0	0	0	1	0	0	1
0	0	0	0			

01xx - 0100
0101
0110
0111

001x - 0010
0011

1xxx - 1000 1101
1001 1110
1010 1111
1011
1100

The higher the subscript number, input have the higher priority. Input D_3 has the highest priority, so regardless of the values of the other i/p's, when this input is 1, the o/p for xy is 11. D_2 has next priority level and so on.

22

$$V = D_0 + D_1 + D_2 + D_3$$

x	$D_3 D_2$	$D_1 D_0$	00	01	11	10
00			x			
01			1	1	1	1
11			1	1	1	1
10			1	1	1	1

$$x = D_2 + D_3$$

y	$D_3 D_2$	$D_1 D_0$	00	01	11	10
00					1	1
01						
11			1	1	1	1
10			1	1	1	1

$$y = D_3 + D_2' D_1$$

The priority encoder is implemented in fig.

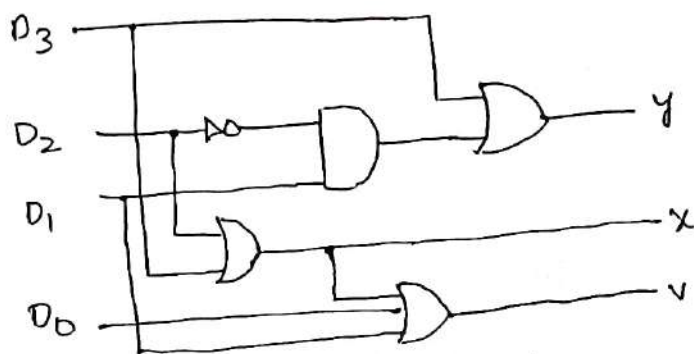


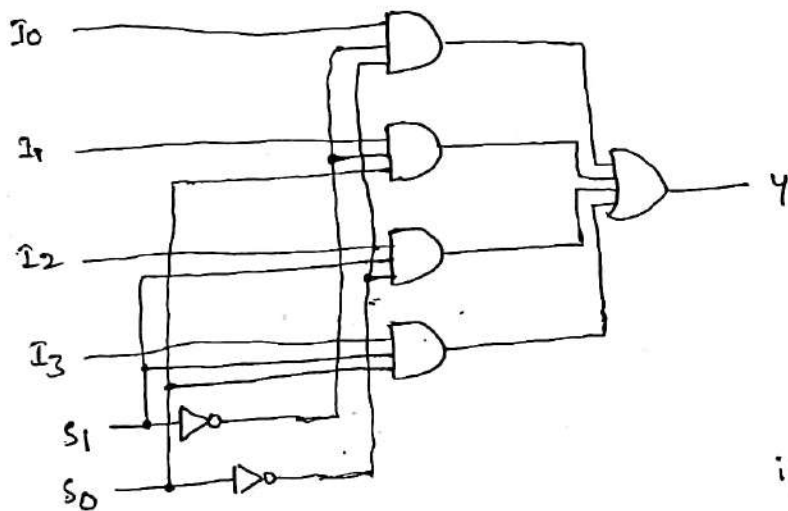
fig:- 4-bit priority encoder

Multiplexers:- (a) Data Selector

28. A digital multiplexer is a combinational circuit that selects binary information from one of many input lines and directs it to a single output line. The selection of a particular input line is controlled by a set of selection lines.

Normally, there are 2^n input lines and n selection lines whose bit combinations determine which input is selected.

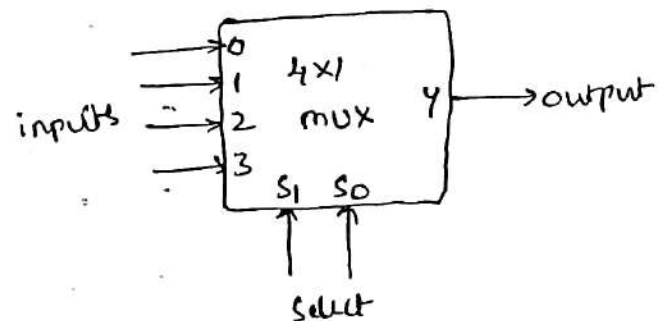
A 4-to-1 line multiplexer is shown in fig (a). Each of the four input lines I_0 to I_3 , is applied to one input of an AND gate. Selection lines S_1 and S_0 are decoded to select a particular AND gate.



(a) logic diagram

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

(b) function table



(c) Block diagram

The function table lists the input to output path for each possible bit combination of the selection lines. It is represented in block diagram form as shown in fig (c)

To demonstrate the ckt operation, consider the case when $S_1 S_0 = 10$. The AND gate associated with I_2 has two of its inputs equal to 1 and the 3rd input connected to I_2 . The other 3 AND gates have at least one input equal to 0, which makes their OR equal to '0'. The OR gate OR is now equal to the value of I_2 . That means a path is provided from selected I_2 to the output.

(29)

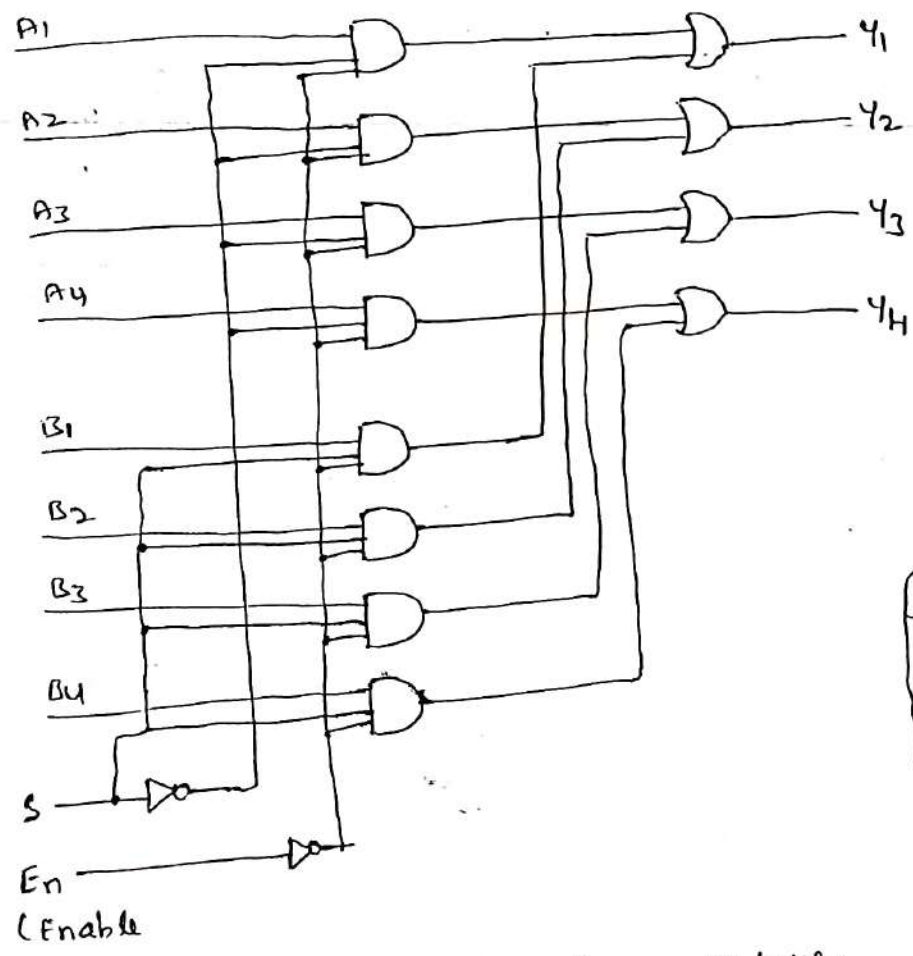
A multiplexer is also called a data selector, since it selects one of many inputs and directs the binary information to the output line.

The AND gates and inverters in the multiplexer resemble a decoder circuit and they decode the input selection lines. In general, a 2^n to 1 line multiplexer is constructed from an n -to- 2^n decoder by adding to it 2^n input lines, one to each AND gate. The o/p's of the AND gates are applied to a single OR gate to provide the 1-line o/p.

Multiplexers may have an enable i/p to control the operation of the unit. When the enable input is in a given binary state, the o/p's are disabled and when it is in the other state the circuit functions as a normal multiplexer.

Quadruple 2 to 1 line multiplexer:-

It has four ^{2 to 1} multiplexers, each capable of selecting one of two input lines, as shown in fig (a). Output Y_1 can be selected



S	En	Y
X	1	all 0's
0	0	Select A
1	0	Select B

(b) function table

fig (a) Quadruple 2 to 1 line multiplexer

to be either A_1 or B_1 . Similarly, output Y_2 may have the value of A_2 or B_2 and so on. The control input "E" enables the multiplexers in the '0' state and disables them in the '1' state.

Boolean function implementation

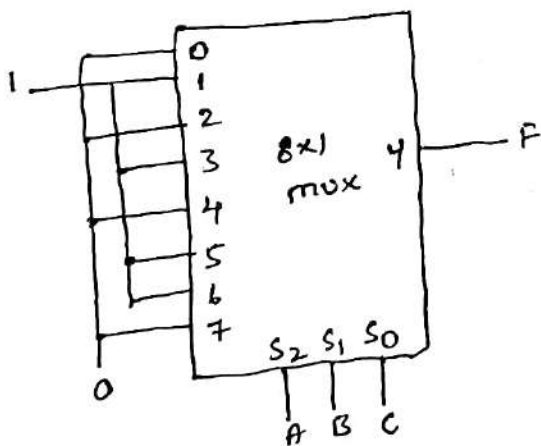
A multiplexer consists of a set of AND gates whose outputs are connected to single OR gate. Because of this construction any boolean function in a SOP form can be easily realized using mux. Each AND gate in the multiplexer represents a minterm. In 8 to 1 multiplexer, there are 3 select inputs and 2^3 minterms. By connecting the function variables directly to the select inputs, a multiplexer can be made to select the AND gate that corresponds to the minterm in the function.

If a minterm exists in a function, we have to connect the AND gate data input to logic 1 otherwise we have to connect it to logic 0.

If we have a boolean function of $n+1$ variables, we take n of these variables and connect them to the selection lines of a mux. The remaining single variable of the function is used for the input of the mux. If A is this single variable, the inputs of the multiplexer are chosen to be either A or A' or 1 or 0. In this way, it is possible to generate any function of $n+1$ variables with a 2^n to 1 mux.

Ex:- Implement the following boolean function using 8x1 mux.

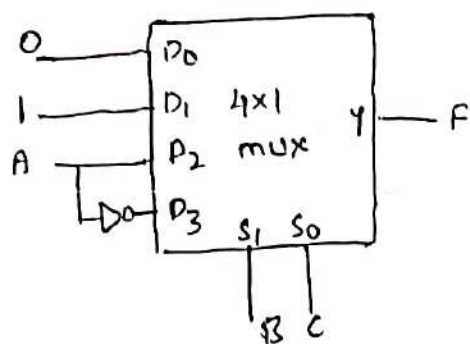
$$F(A,B,C) = \sum m(1, 3, 5, 6)$$



Ex:- Implement the following boolean function using 4 to 1 mux

(2)

$$F(A, B, C) = \sum m(1, 3, 5, 6)$$



Here, two of the variables B and C are applied to the selection lines. The data inputs for mux are derived from the implementation table.

Implementation table is nothing

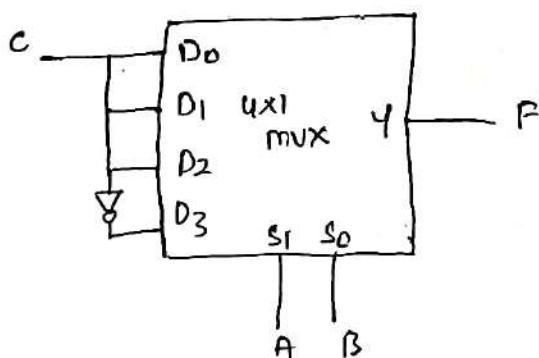
but the list of the inputs of the mux and under them list all the minterm in two rows. The minterms given in

the function are circled and then each column is inspected separately as follows.

- ① If the two minterms in a column are not circled, 0 is applied to the corresponding multiplexer input.
- ② If the two minterms in a column are circled, 1 is applied to the corresponding multiplexer input.
- ③ If the minterm in the second row is circled and minterm in the 1st row is not circled, A is applied to the corresponding mux input.
- ④ If the minterm in the first row is circled and minterm in the second row is not circled, \bar{A} is applied to the corresponding multiplexer i/p.

	D ₀	D ₁	D ₂	D ₃
\bar{A}	0	①	2	③
A	4	⑤	⑥	7

Another way



	D ₀	D ₁	D ₂	D ₃
\bar{C}	0	2	4	⑥
C	①	③	⑤	7

52) Demultiplexer

A demultiplexer is a circuit that receives information on a single line and transmits this information on one of 2^n possible o/p lines. The selection of specific o/p line is controlled by the values of n selection lines as shown in fig.

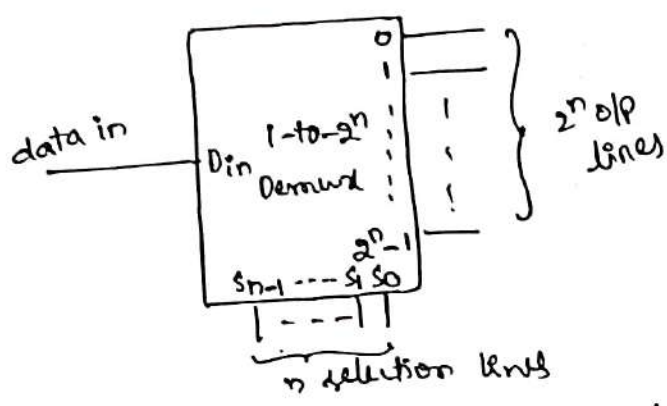


Fig:- Block diagram of 1- 2^n demultiplexer

Following fig shows the block diagram and logic diagram of 1-4 demultiplexer. The single input variable D_{in} has a path to all four o/p's, but the input information is directed to only one of the 4 o/p lines.

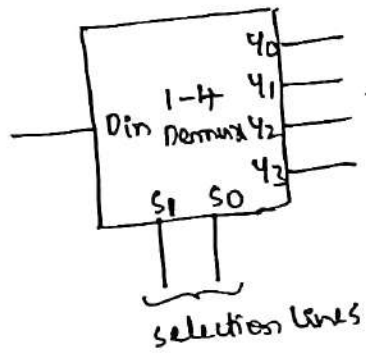


Fig:- Block diagram of 1-4 demultiplexer

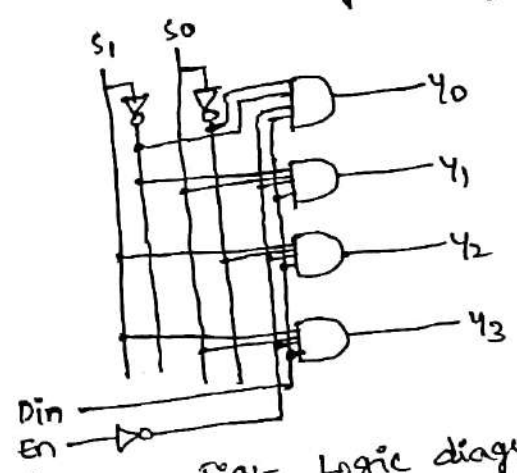


Fig:- Logic diagram

the operation of demultiplexer is described by the following functional table

inputs			outputs			
En-L	s ₁	s ₀	y ₀	y ₁	y ₂	y ₃
0	0	0	Din	0	0	0
0	0	1	0	Din	0	0
0	1	0	0	0	Din	0
0	1	1	0	0	0	Din
1	x	x	0	0	0	0

4. Sequential circuits

Sequential circuits :-

A block diagram of a sequential circuit is as shown in fig. It consists of a combinational circuit to which storage elements are connected to form a feedback path.

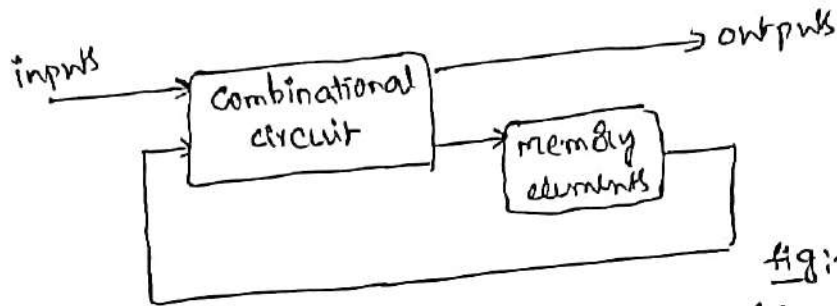


fig:- Block diagram

The storage elements are devices capable of storing binary information. The binary information stored in these elements at any given time defines the state of sequential circuit at that time.

The sequential circuit receives binary information from external inputs together with the present state of the storage elements to determine the binary value of the outputs.

→ The block diagram demonstrates that the op's in a sequential circuit are a function not only of the inputs, but also of the present state of the storage elements.

The next state of the storage elements is also a function of external inputs and the present state.

Sequential circuits are classified into

- clocked or synchronous sequential circuit
- asynchronous sequential circuit.

→ A Synchronous sequential circuit is a system whose behavior can be defined from the knowledge of its signals at discrete instants of time.

→ The behavior of an asynchronous sequential circuit depends upon input signals at any instant of time and the order in which the inputs change.

The storage elements commonly used in asynchronous sequential circuits are time delay devices or latches.

② In practice, the inherent propagation delay of logic gates is of sufficient duration to produce the needed delay, so that actual delay units may not be necessary.

The storage elements used in clocked or synchronous sequential circuits are called flip-flops. A flip-flop is a binary storage device capable of storing one bit of information. In a stable state, the output of a flip-flop is either 0 or 1. A sequential circuit may use many flip-flops to store as many bits as necessary. The state of flip-flops are affected at only discrete instants of time or with the arrival of each pulse.

Storage elements: latches

A storage element in a digital circuit can maintain a binary state indefinitely until disturbed by an input signal to switch states.

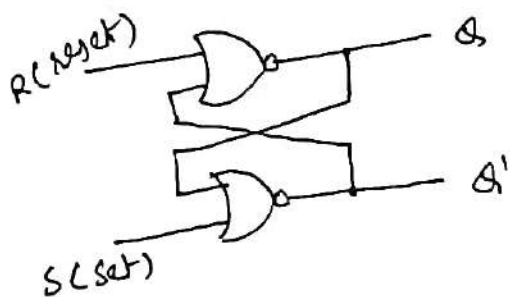
The major difference among various types of storage elements are in the no. of inputs they possess and in the manner in which the inputs affect the binary state.

Storage elements that operate with signal levels are referred to as latches and those controlled by a clock transition are flip-flops. Latches are said to be level sensitive devices and flip-flops are edge sensitive devices. The two types of storage elements are related because latches are the basic ckt from which all flip-flops are constructed.

SR latch:- The SR latch is a ckt with two inputs coupled NOR gates (or) two cross coupled NAND gates. The SR latch constructed with 2 cross coupled NOR gates is shown in fig.

The latch has 2 useful states.

When $Q=1$ & $Q'=0$, the latch is said to be in set state. When $Q=0$ & $Q'=1$, the latch is said to be in reset state. Outputs Q and Q' are normally the complement of each other.



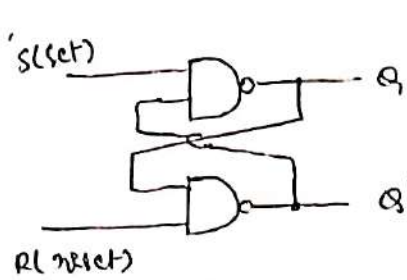
③ However, when both inputs are equal to 1 at the same time, a condition is produced in which both Q 's are equal to '0' occurs. This state is known as unpredictable (or) undefined (or) a metastable state. Consequently, in practical applications, setting both inputs to 1 is forbidden.

Under normal conditions, both inputs of the latch remain at '0' unless the state has to be changed.

The application of a momentary '1' to the 'S' input causes the latch to go to the set state. The 'S' input must go back to '0' before any ^{other} changes take place, in order to avoid the occurrence of an undefined next state. The functional table of SR latch is shown in fig.

S	R	Q	Q'
0	0	No change	
0	1	0	1
1	0	1	0
1	1	0	0 (forbidden)

(A) The SR latch with two cross-coupled NAND gates is as shown in fig.



(a) Logic Diagram

		Next state of				Next state of		
S	R	Q	Q'	S	R	Q	Q'	
0	0	1	1	1	0	0	1	(After S=1, R=0)
0	1	1	0	1	1	0	0	(After S=0, R=1)
1	0	0	1	0	1	1	0	(After S=0, R=1)
1	1	No change	No change	0	0	1	1	(Forbidden)

(b) Function Table

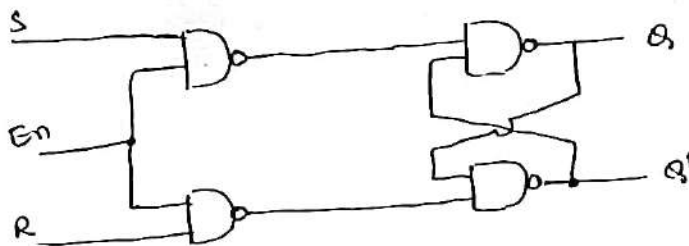
It operates with both inputs normally at 1, unless the state of latch has to be changed. The application of '0' to the "S" input causes output 'Q' to go to 1, putting the latch in set state. When the S input goes back to 1, the circuit remains in set state.

After both inputs go back to 1, we are allowed to change the state of latch by placing a "0" in the R input. This causes the circuit to go to the reset state and stay there even after both inputs return to 1. The forbidden condition for the NAND latch is both inputs being equal to 0. At the same time, an input combination that should be avoided.

Because the NAND latch requires "0" signal to change its state, it is sometimes referred to as S'R' latch.

Gated SR Latch:-

The operation of the basic SR latch can be modified by providing an additional input signal that determines (controls) when the state of the latch can be changed. An SR latch with a control input is as shown in fig.

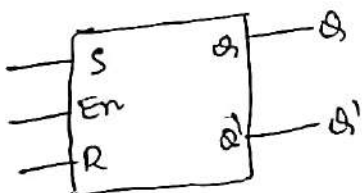


En	S	R	Next state of Q
0	x	x	No change
1	0	0	No change
1	0	1	Q=0, reset state
1	1	0	Q=1, set state
1	1	1	Indeterminate

function table.

It consists of the basic SR latch and two additional NAND gates. The "En" input acts as control input for the other two inputs. The ~~state~~ extended function table is as follows.

En	S	R	Q _n	Q _{n+1}	state
0	x	x	0	0	No change
0	x	x	1	1	
1	0	0	0	0	No change
1	0	0	1	1	
1	0	1	0	0	Reset
1	0	1	1	0	
1	1	0	0	1	Set
1	1	0	1	1	

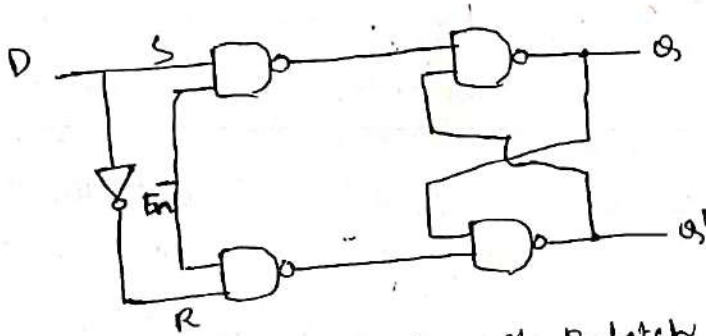


⑤

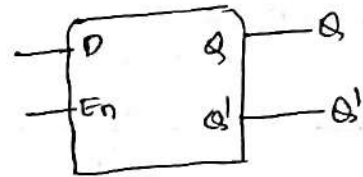
1 1 1 0 x } Indeterminate
1 1 1 1 x

Grated D latch:-

Looking at the truth table of SR latch we can find that when both inputs are same the o/p either does not change (as it is invalid). In many practical applications, these two conditions are not required. These input conditions can be avoided by making them complement of each other. This modified SR latch is known as D latch (or) Delay latch.



Logic Diagram of D-latch



Logic Symbol

En	D	Qn	Qnt+1	state
0	x	x	Qn	No change
1	0	x	0	reset
1	1	x	1	set

En	D	next state of Q Qnt+1
0	x	Qn
1	0	0
1	1	1

Qnt+1	Qn	0	1
0	0	0	0
1	0	1	1

characteristic equation
 $Q_{nt+1} = D$

characteristic equation of SR latch

Qnt+1	S	Qn	01	11	10
0	0	0	1		
1	0	1	1	x	x

$$Q_{nt+1} = S + \bar{R} Q_n$$

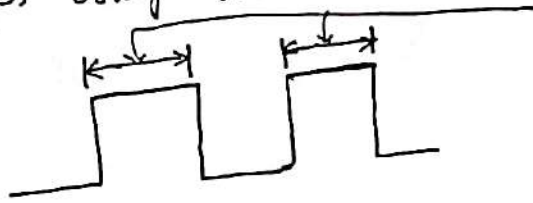
→ Latches and flip flops are the basic building blocks of most sequential circuits. The main difference b/w latches and f/f's is that the method ~~using~~ used for changing their state.

We have seen SR latch and D latch with enable i/p. Latches are controlled by enable signal and they are level triggered either +ve level triggered (or) negative level triggered. The o/p state is free to change according to S and R input values, when active level is maintained at the enable i/p.

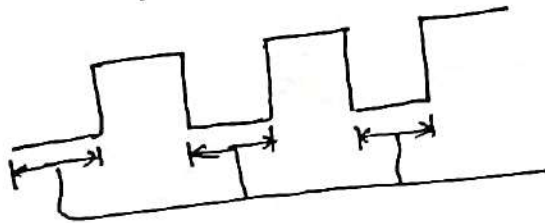
⑥ Flip-flops are different from latches. FF's are edge triggered instead of level-triggered.

Level triggering:-

Positive level triggered:- The output of latch responds to the input changes only when its enable i/p is '1' (High) only when $En=1$ otherwise it is disabled.



Negative level triggered:- The output of latch responds to the input changes only when its enable input is '0' (Low).



latch is enabled only when $En=0$. otherwise it is disabled

The logic diagram for -ve level triggered SR latch is shown in fig

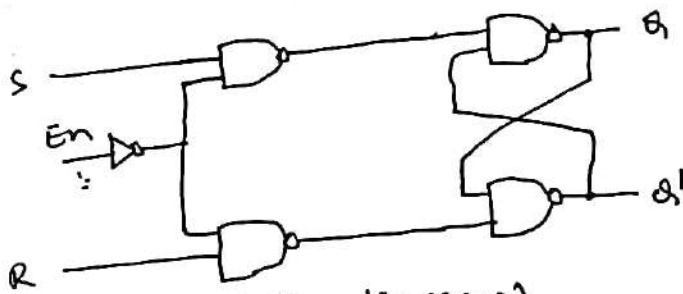


fig:- logic diagram)

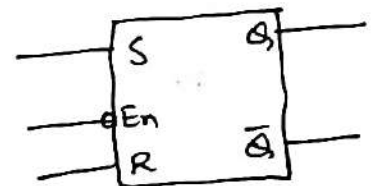


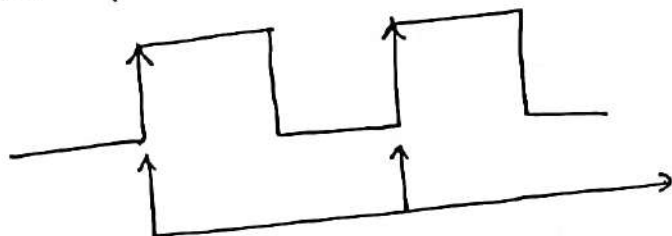
fig:- logic symbol)

characteristic table

En	S	R	Next state of	
			Q	Q'
1	x	x	No change	
0	0	0	No change	
0	0	1	0	1
0	1	0	1	0
0	1	1	undefined state	

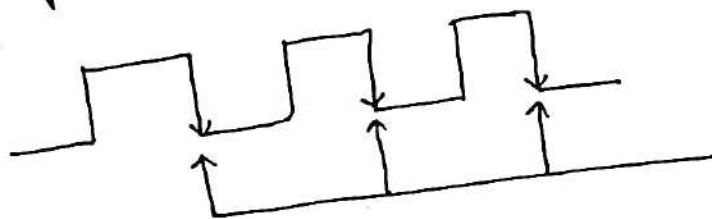
⑦ Edge triggering:- In the edge triggering, the o/p responds to changes in the i/p only at positive edge (or) negative edge of the clock input. There are two types of edge triggering.

Positive edge triggering:- In this case, the output responds to the changes in input only at positive edge (0 to 1 transition) of clock pulse.



o/p responds only at +ve edge of the clock pulse.

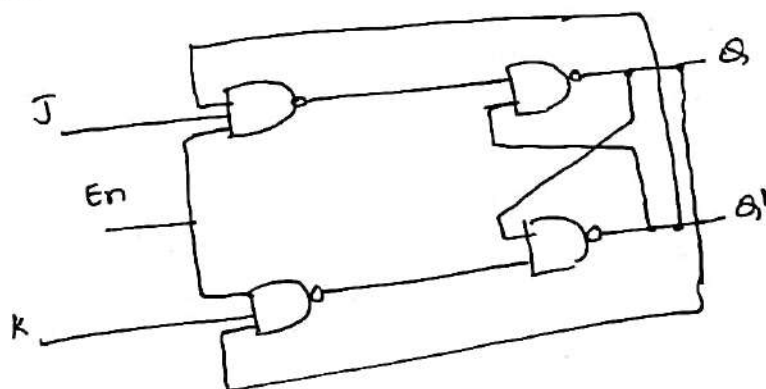
Negative edge triggering:- Here, the o/p responds to the changes in input only at -ve edge (1 to 0 transition) of the clock pulse.



output responds only at -ve edge of clock pulse.

Gated JK latch

The undefined state of SR latch when $S=R=1$ can be eliminated by converting it into JK latch. The logic diagram of JK latch is shown in fig.



Characteristic table			Next state
En	J	K	Q
0	X	X	No change
1	0	0	No change
1	0	1	0
1	1	0	1
1	1	1	Toggle

When $En=0$, one of the i/p of 1st level NAND gates is '0', so they result '1' as o/p. and these will act as input for basic SR latch. When $S=R=1$, there is no change in latch state.

③ When $En=1$, the latch is enabled and the latch state depends on J & K input values.

case i:- when $J=K=0$, there is no change in the latch state

case (ii):- when $J=1, K=0$ is applied, irrespective of present state output is set to '1'

case (iii):- when $J=0, K=1$ is applied, irrespective of present state output is set to '0'

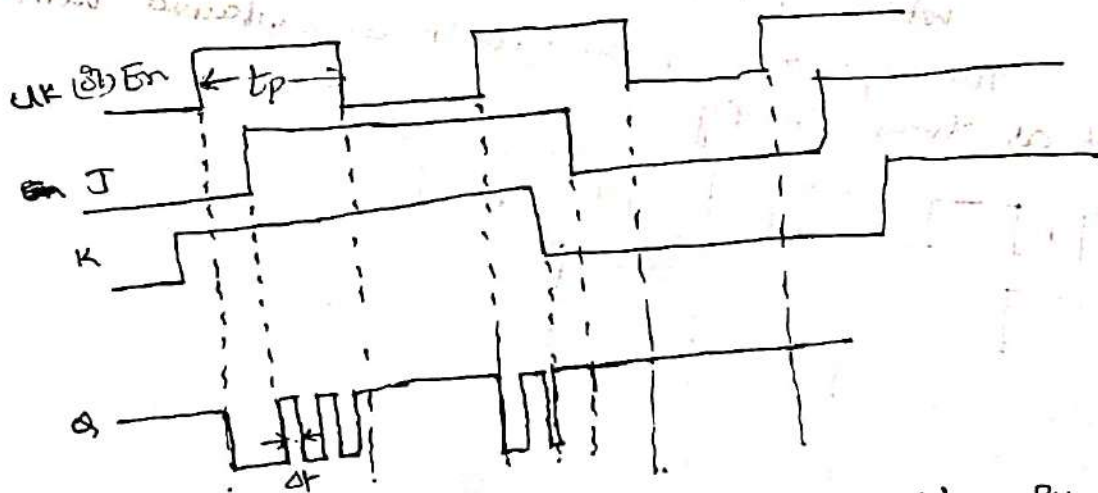
case (iv):- when $J=1, K=1$ is applied, the output is complemented

Race around condition:-

In JK ~~flip-flop~~ latch

when $J=K=1$, the o/p toggles.

Consider that initially $Q=0$ and $J=K=1$. After a time interval Δt equal to the propagation delay through 2 NAND gates in series, the o/p will change to $Q=1$ and after another time interval of Δt the o/p will change back to $Q=0$. This toggling will continue until the ~~latch~~ is enabled and $J=K=1$. At the end of clock pulse the latch is disabled and the value of Q is uncertain. This situation is referred to as race-around condition. This is shown in fig.



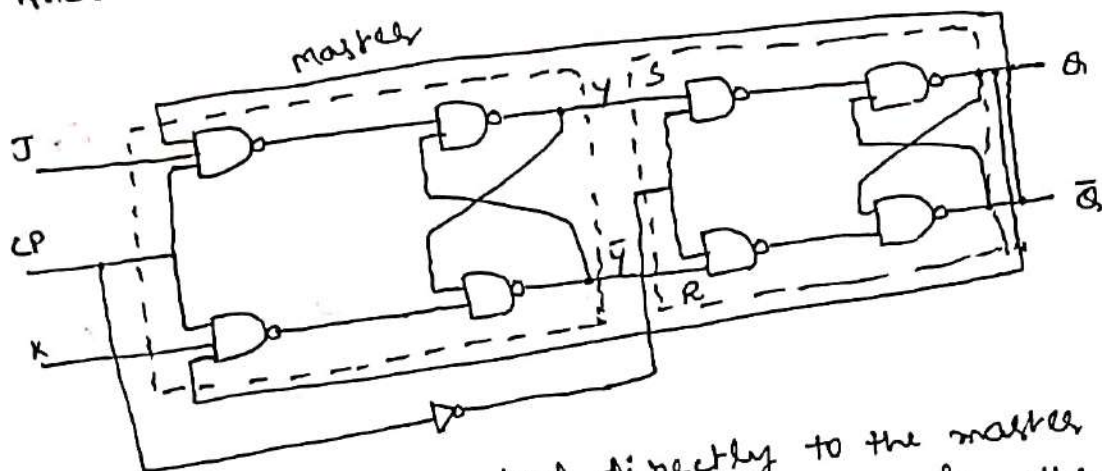
This condition exists when $t_p \geq \Delta t$. By keeping $t_p < \Delta t$ we can avoid race around condition.

(or) A more practical method for overcoming this difficulty is the use of master-slave flip-flop.
(or) using edge triggered flip-flops.

⑨ master slave JK flip-flop:-

A master slave flip-flop is constructed from two latches. one circuit serves as a master and the other as a slave. The overall circuit is referred to as master-slave ff.

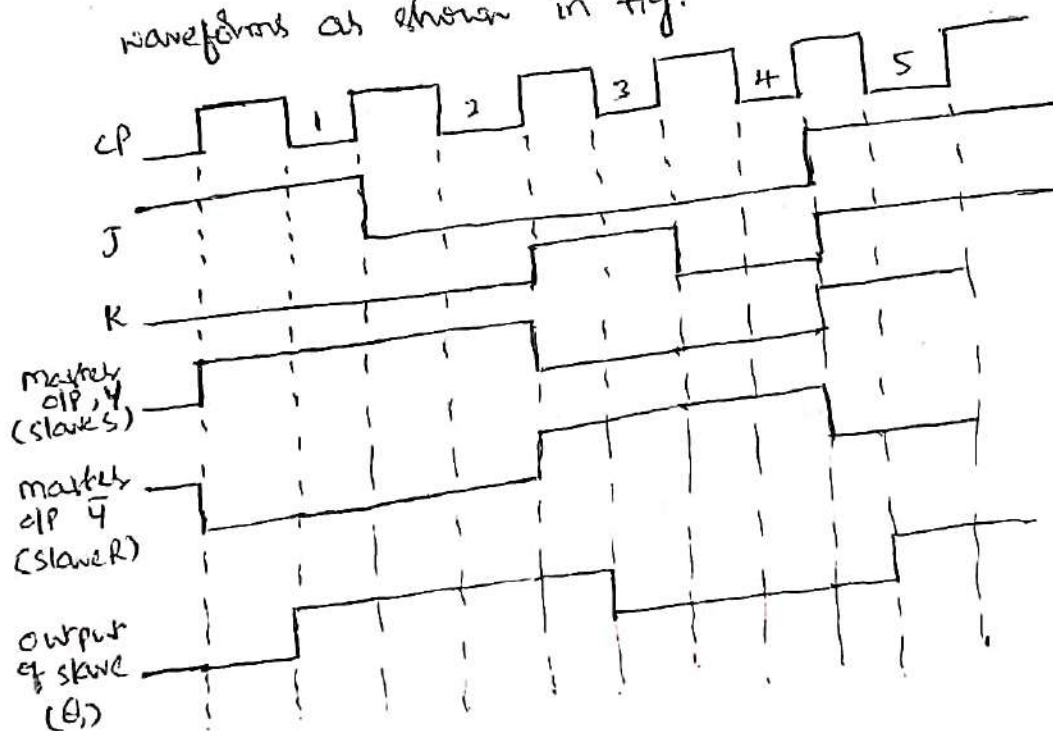
Following fig shows JK master-slave ff. It consists of JK latch as a master and gated SR latch as a slave. The ^{or the} op master is fed as an ip to the slave.



clock signal is connected directly to the master but it is connected through inverter to the slave. Therefore the information present at J and K ip's is transmitted to the op of master when $CP=1$ and it is held there until the $CP=0$, after which it is allowed to pass through to the op of slave.

~~When J=1 and K=1, master~~

The operation of the circuit is explained with timing waveforms as shown in fig.



⑩ characteristic equation for JK:-

The expanded truth table for JK is shown below.

characteristic table

E_n	J	K	Q_n	Q_{n+1}
0	x	x	0	0
0	x	x	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	0
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

Q_{n+1}	Q_n			
J \ K	00	01	11	10
0		1		
1	1	1		1

$$Q_{n+1} = J\bar{Q}_n + K'Q_n$$

clocked T flip-flop:- (or) positive edge triggering T flip-flop

T flip-flop is also known as Toggle flip-flop. It is a modification of JK flip-flop. As shown in fig(a), the T flip-flop is obtained from a JK f/f by connecting both inputs J & K together. The logic symbol is shown in fig(b)

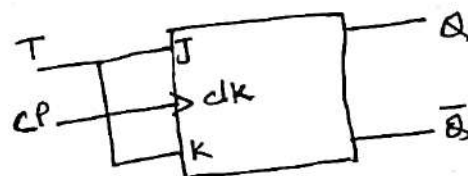
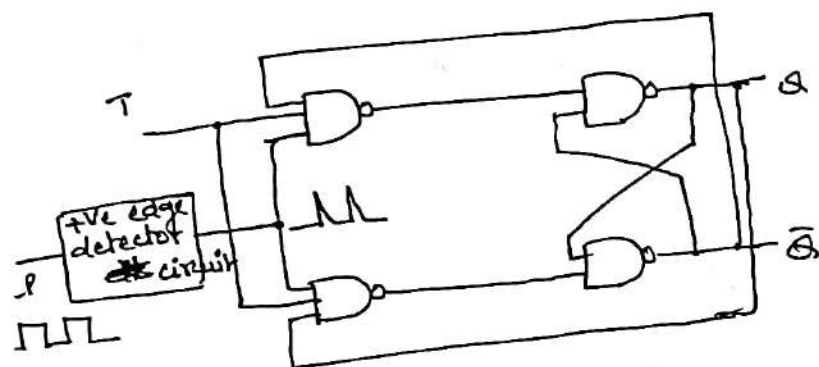


fig:- Logic Symbol

fig(a): logic diagram

When $T=0$, $J=K=0$ and hence there is no change in the CP. When $T=1$, $J=K=1$ and hence CP toggles. The truth table (or) characteristic table is shown below.

CP	T	Q_{n+1}
\uparrow	0	Q_n
\uparrow	1	\bar{Q}_n
\uparrow	x	Q_n

(not a edge)

To get the characteristic equation, expand the truth table.

②

CP	T	Q_n	Q_{n+1}
↑	0	0	0
↑	0	1	1
↑	1	0	1
↑	1	1	0
⌋	x	0	0
⌋	x	1	1

Q_{n+1}	$T \backslash Q_n$	0	1
0	0	0	1
1	1	1	0

$$Q_{n+1} = T'Q_n + TQ_n'$$

$$= T \oplus Q_n$$

Negative edge triggering T flip-flop:-

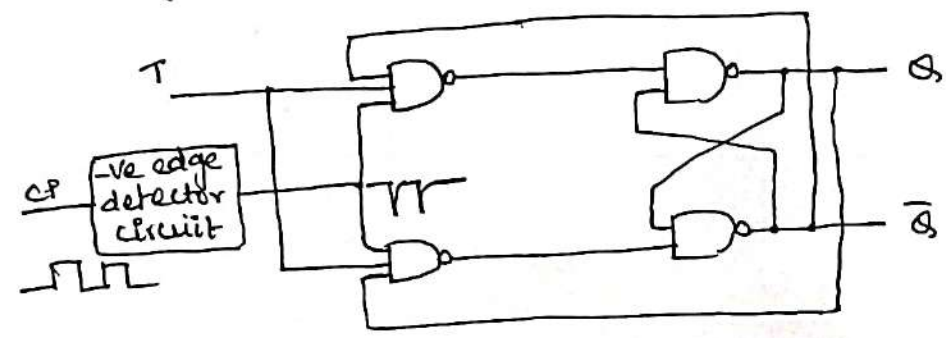


fig:- Logic diagram

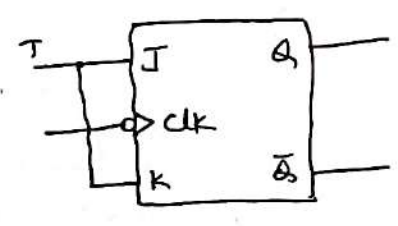


fig:- logic symbol

Truth table

CP	T	Next state of Q Q_{n+1}
↓	0	Q_n
↓	1	$\overline{Q_n}$
⌋	x	Q_n

(not a negative edge)

⑫ F/F Excitation table:- (or) transition table

During the design process, we know the sequence of states i.e. the transition from each present state to its corresponding next state. From this information we wish to find the f/f input conditions that will cause the required transition. For this reason, we need a table that lists the required inputs for a given change of state. Such a table is known as excitation table of the f/f. It can be derived from their truth table.

RS f/f:-

Truth table

S	R	Q_n	Q_{n+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

Excitation Table

Q_n	Q_{n+1}	S	R
0	0	0	X
0	1	1	0
1	0	0	1
1	1	X	0

JK f/f:-

Truth Table

J	K	Q_n	Q_{n+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

Excitation table

Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

D f/f:-

Truth table

D	Q_n	Q_{n+1}
0	0	0
0	1	0
1	0	1

Excitation table

Q_n	Q_{n+1}	D
0	0	0
0	1	1
1	0	0
1	1	1

13

T FF:-

Truth Table

T	Q_{n+1}
0	Q_n
1	\bar{Q}_n

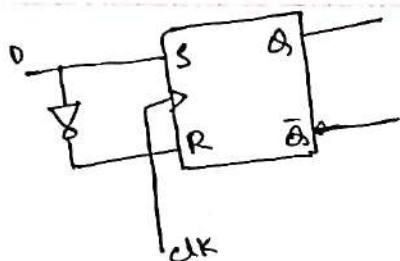
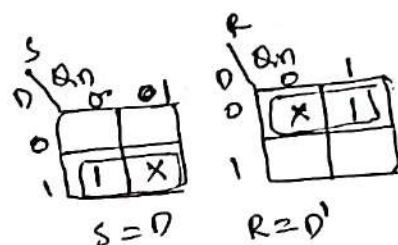
Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

Conversion of flip-flops :-

It is possible to convert one flip-flop into another flip-flop with some additional gates.

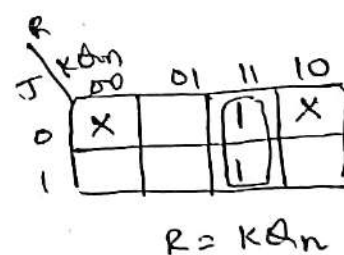
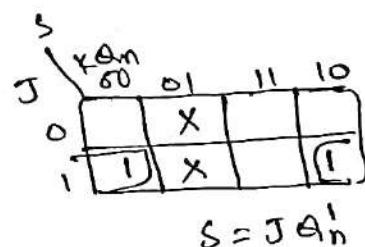
SR flip to D flip:-

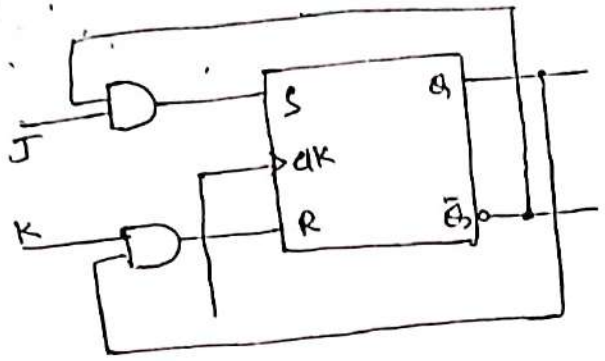
Input	Present State	Next State	Flip-flop's	
	Q_n	Q_{n+1}	S	R
0	0	0	0	X
0	1	0	0	1
1	0	1	1	0
1	1	1	X	0



SR flip to JK flip

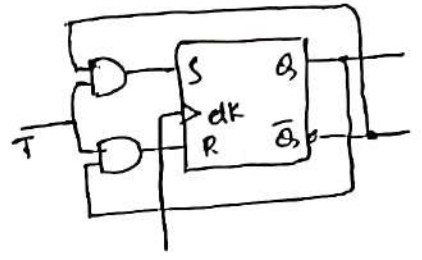
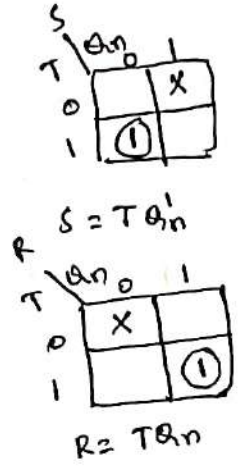
Inputs	Present State	Next State	Flip-flop's	
J K	Q_n	Q_{n+1}	S	R
0 0	0	0	0	X
0 0	1	1	X	0
0 1	0	0	0	X
0 1	1	0	0	1
1 0	0	1	1	0
1 0	1	1	X	0
1 1	0	1	1	0
1 1	1	0	0	1





SR fls to T-fls

input	present state	Next state	fls inputs	
T	Q_n	Q_{n+1}	S	R
0	0	0	0	X
0	1	1	X	0
1	0	1	1	0
1	1	0	0	1



Asynchronous (or) Direct inputs

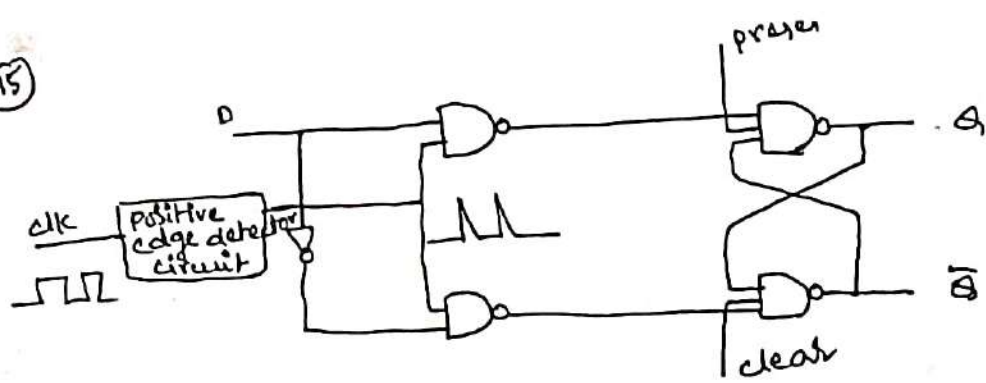
For the clocked flip-flops, the S-R, D, T and J-K inputs are called asynchronous inputs because their effect on the fls o/p is synchronized with the clock input.

Flip-flops available in IC packages sometimes provide special inputs for setting (preset) or clearing (clear) the fls asynchronously. These inputs are called asynchronous or direct inputs. These inputs are connected directly into the latch portion of fls so that they override the effect of synchronous inputs and clock.

When power is turned on to a digital system, the state of fls is unknown. The direct inputs are useful for bringing all fls in the system to a known state prior to clocked operation.

A positive edge triggered D fls with active low preset and clear inputs is shown in fig.

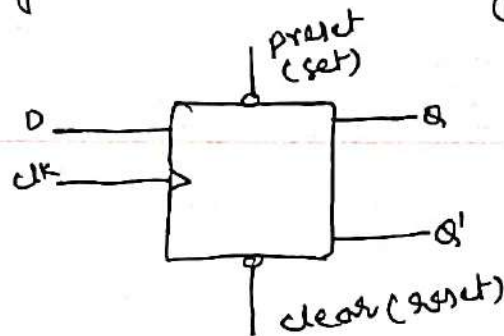
15



The characteristic table is as follows.

preset	clear	clk	D	Next state of Q, Q'	
0	1	x	x	1	0
1	0	x	x	0	1
0	0	x	x	1	1
1	1	↑	0	0	1
1	1	↑	1	1	0
1	1	↑ (not a +ve edge)	x	No change	

The logic symbol is shown in fig.



(16)

Analysis of clocked sequential circuits

Analysis describes what a given circuit will do under certain operating conditions. The behavior of a clocked sequential circuit is determined from the inputs, outputs and the state of its flip-flops. The outputs & the next state are both a function of the inputs and the present state.

The analysis of a sequential circuit consists of obtaining a ^{state} table (or) a ^{state} diagram for the time sequence of inputs, outputs and internal states.

A logic diagram is recognized as a clocked sequential circuit if it includes flip-flops with clock inputs. The flip-flops may be of any type, and the logic diagram may (or) may not include combinational circuit gates.

State equations:- The behavior of a clocked sequential circuit can be described algebraically by means of state equations.

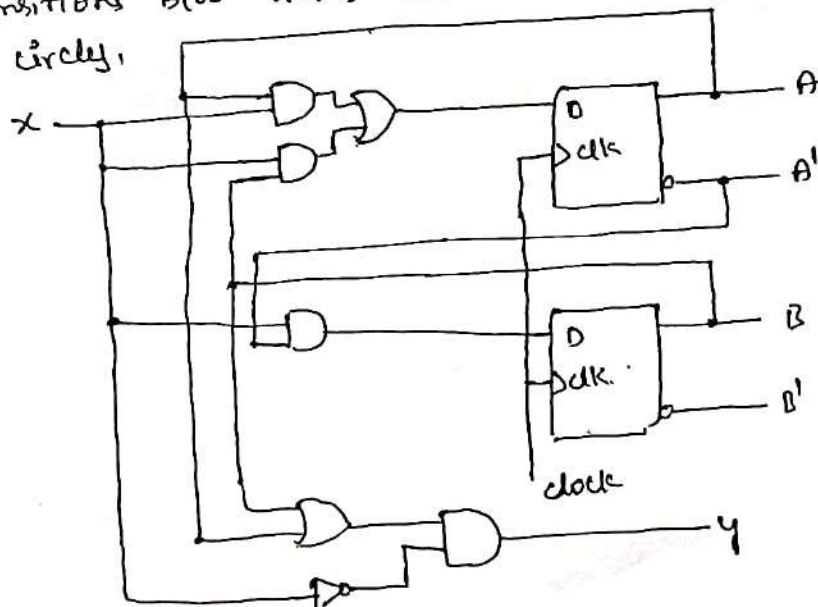
A state equation specifies the next state as a function of present state and inputs.

State Table:- The time sequence of inputs, outputs and flip-flop states can be enumerated in a state table.

State diagrams:-

The information available in a state table can be represented graphically in the form of a state diagram. In this type of diagram, a state is represented by a circle and the transitions b/w states are indicated by directed lines connecting the circles.

Ex:-



(12)

For From the diagram, $D_A = A(t) \oplus B(t) \oplus x(t)$

$$D_B = A'(t) \oplus x(t)$$

Since the characteristic equation of D flip-flop is $Q(t+1) = D$, it is possible to write a set of state equations for the circuit

$$A(t+1) = A(t) \oplus B(t) \oplus x(t)$$

$$B(t+1) = A'(t) \oplus x(t)$$

A state equation is an algebraic expression that specifies the condition for a flip state transition.

Since all the variables in the boolean expression are a function of present state, we can omit the designation (t) after each variable.

$$A(t+1) = Ax + Bx$$

$$B(t+1) = A'x$$

Also, the present state value of the output can be expressed algebraically as

$$y(t) = [A(t) + B(t)] x'(t)$$

By removing the symbol (t) for the present state, we obtain the output boolean expression

$$y = (A + B) x'$$

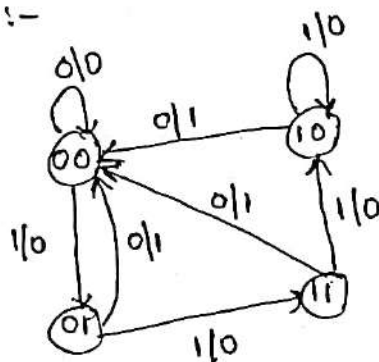
state table

Present state		Input x	Next state		Output y
A	B		A^+	B^+	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	1
0	1	1	1	1	0
1	0	0	0	0	1
1	0	1	1	0	0
1	1	0	0	0	1
1	1	1	1	0	0

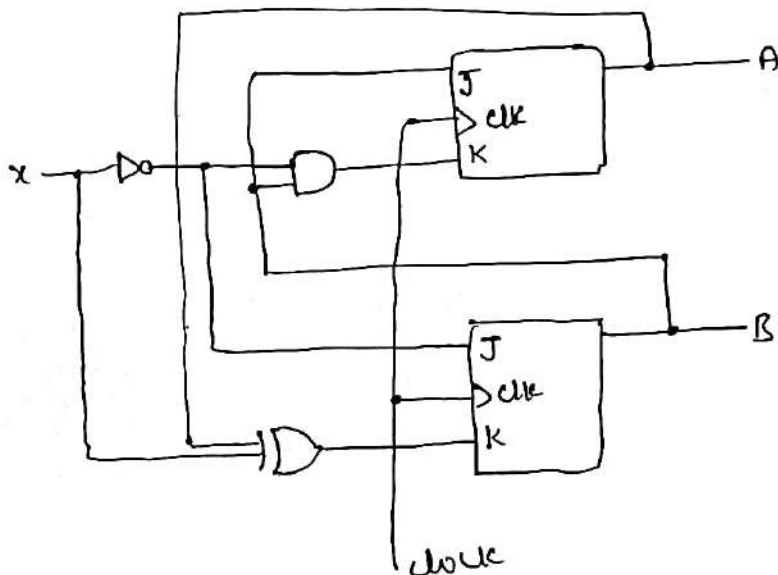
Another form of the state table is

present state		Next state				output	
		x=0		x=1		x=0	x=1
A	B	A ⁺	B ⁺	A ⁺	B ⁺	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

State diagram :-



Analysis with JK flip-flops



from fig, $J_A = B$, $K_A = x'B$
 $J_B = x'$, $K_B = A \oplus x$

characteristic equation of JK is
 $Q(t+1) = JQ' + K'Q$

$$A(t+1) = J_A A' + K_A' A = BA' + (x'B)'A = BA' + (x+B)'A = BA' + xA + AB'$$

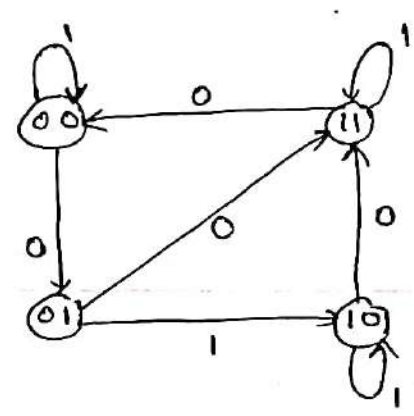
$$B(t+1) = J_B B' + K_B' B = x'B' + (A \oplus x)'B = x'B' + A'B + AB$$

19

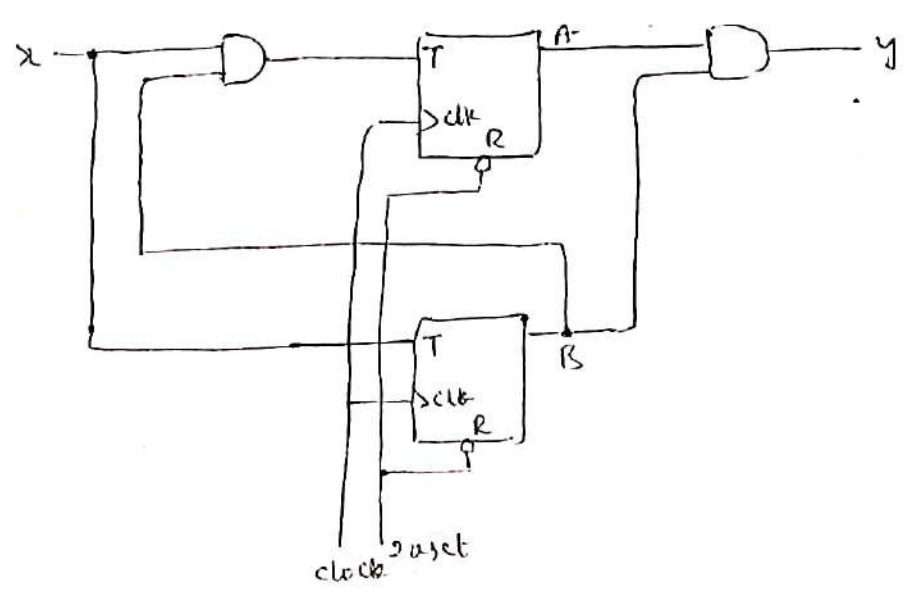
State table

Present state		input	Next state		Flip flop inputs			
A	B		A	B	J _A	K _A	J _B	K _B
0	0	0	0	1	0	0	1	0
0	0	1	0	0	0	0	0	1
0	1	0	1	1	1	1	1	0
0	1	1	1	0	1	0	0	1
1	0	0	1	1	0	0	1	1
1	0	1	1	0	0	0	0	0
1	1	0	0	0	1	1	1	1
1	1	1	0	0	1	0	0	0
1	1	1	1	1	1	0	0	0

state diagram



analysis with T-flip-flops:-



20

$T_0 = Bx$ & $T_1 = x$

characteristic equation of $T(x)$ $T(x) = T(x)A + T(x)B$

Next state equations

$$A(x+1) = BxA' + (Bx)'A$$

$$= A'Bx + (x' + B')A = A'Bx + x'A + B'A$$

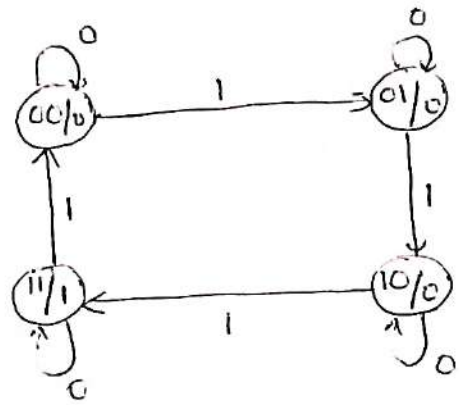
$$B(x+1) = xB' + x'B = xCB$$

state output expression $y = A'B$

State table

Present state		input x	next state		output y	flip-flop inputs	
A	B		A	B		T ₀	T ₁
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1
0	1	0	0	1	0	0	0
0	1	1	1	0	0	1	1
1	0	0	1	0	0	0	0
1	0	1	1	1	0	0	1
1	1	0	1	1	1	0	0
1	1	1	0	0	1	1	1

State diagram



Mealy and Moore models of finite state machines

In synchronous (or clocked sequential networks), clocked flip-flops are used as memory elements which change their individual states in synchronism with the periodic clock signal. Therefore, the change in states of flip-flops and change in state of the entire circuit occurs at the transition of clock signal.

Clocked sequential networks are represented by two models

- (i) Moore model
- (ii) Mealy model

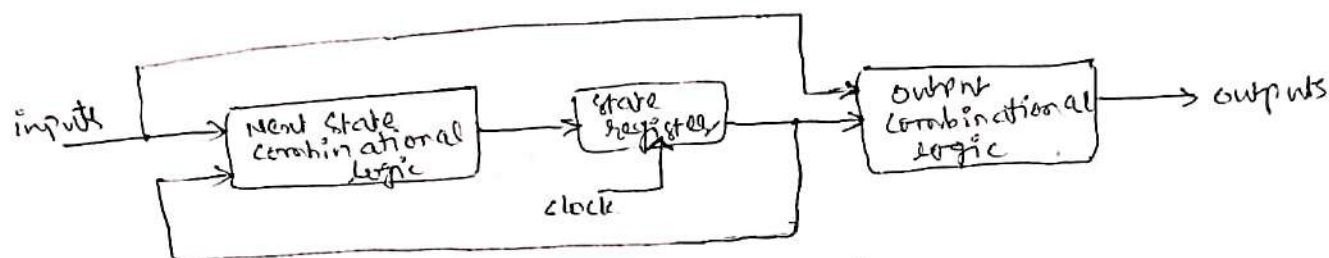
(21)

mealy model:- In the mealy model, the output depends only on the present state of the flip-flops.

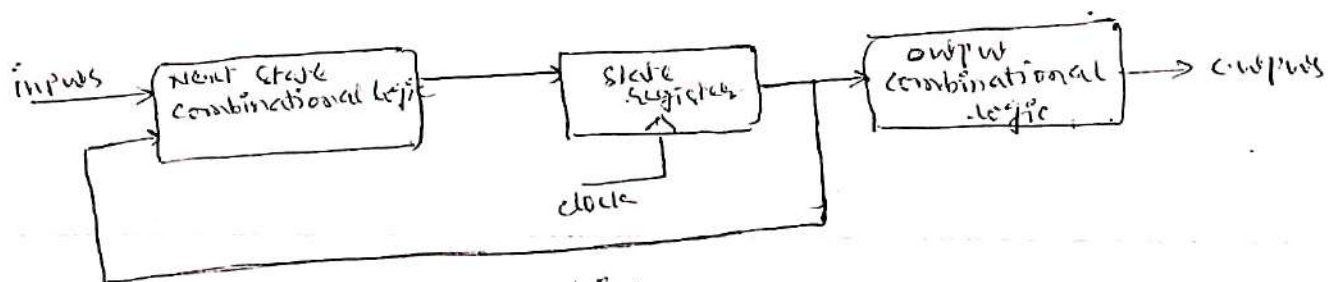
moore model:- In the moore model, the output depends on both the present state of the flip-flops and the inputs.

The two models of a sequential circuit are commonly referred to as a finite state machine (FSM).

The following diagrams show the block diagrams of mealy and moore models.



(a) mealy machine



(b) moore machine

Fig:- Block diagrams of mealy & moore state machines.

In a moore model, the outputs of the sequential circuit are synchronized with the clock because they depend only on flip-flop outputs that are synchronized with the clock.

In a mealy model, the outputs may change if the inputs change during the clock cycle.

⇒ State Reduction and Assignment:-

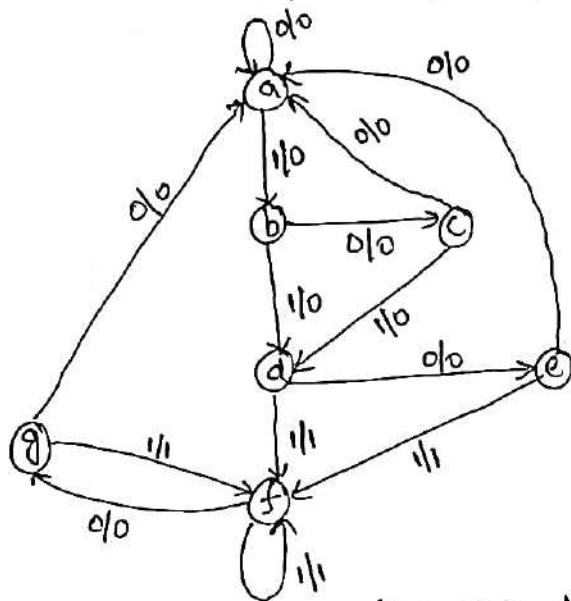
→ The design (synthesis) of a sequential circuit starts from a set of specifications and culminates in a logic diagram. Two sequential circuits may exhibit the same input-output behavior, but have a different no. of internal states in their state diagram.

State reduction:-

The state reduction technique basically avoids the introduction of redundant states. The reduction in redundant states reduce the no. of required fls's and logic gates, therefore reducing the cost of the final circuit.

The two states are said to be equivalent, if every possible set of inputs generate exactly same o/p and same next state. When two states are equivalent, one of them can be removed without altering the i/p - output relationship.

We will illustrate the state-reduction procedure with an example. We start with a sequential circuit whose specification is given in the state diagram of fig.



The states are denoted by letter symbols instead of their binary values, because in state reduction technique internal states are not important but only input-o/p sequences are important.

fig:- state diagram

There are an infinite no. of input sequences that may be applied to the circuit, each results in a unique o/p sequence. For ex. consider the i/p sequence 01010110100 starting from the initial state a. Each input of 0 & 1 produces an o/p of 0 & 1 and causes the circuit to go to next state.

From the state diagram, we obtain the o/p and state sequence for the given i/p sequence as follows.

23

state	a	a	b	c	d	e	f	f	g	f	g	a
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

The state table for the given state diagram is as follows

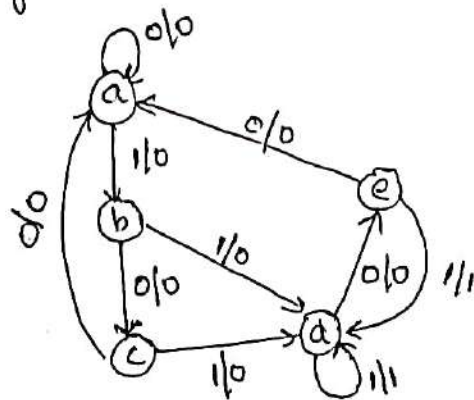
present state	Next state		Output	
	x=0	x=1	x=0	x=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	g	f	0	1
g	a	f	0	1

e and g states are equivalent to each other because they produce same output for the same inputs and the circuit enters into same next state so eliminate one state. The row with present state g is removed and state g is replaced by state "e". Each time it occurs in the next state column.

present state	Next state		Output	
	x=0	x=1	x=0	x=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	f	0	1
e	a	f	0	1
f	e	f		

present state	Next state		Output	
	x=0	x=1	x=0	x=1
a	a	b	0	0
b	c	d	0	0
c	a	d	0	0
d	e	d	0	1
e	a	d	0	1

24 The final reduced table is shown and the diagram is as follows.



fig(b) reduced state diagram

This state diagram satisfies the signal ip-ops specifications and will produce the required op sequence for any given ip sequence. The following list derived from the state diagram of fig(b) is for the ip sequence used previously.

state	a	a	b	c	d	e	d	d	e	d	e	a
input	0	1	0	1	0	1	1	0	1	0	0	
output	0	0	0	0	0	1	1	0	1	0	0	

State assignment:- In order to design a sequential circuit with physical components, it is necessary to assign unique coded binary values to the states. For a circuit with m states, the codes must contain n bits where $2^n \geq m$. For example with 3 bits it is possible to assign codes to 8 states denoted by binary numbers 000 through 111.

In case of reduced state table, only five states need binary assignment and we are left with 3 unused states. Unused states are treated as don't care conditions during the design.

Three possible Binary state assignments

state	Assignment 1 Binary	Assignment 2 gray code	Assignment 3 one hot
a	000	000	00001
b	001	001	00010
c	010	011	00100
d	011	010	01000
e	100	110	10000

25) Design Procedure:-

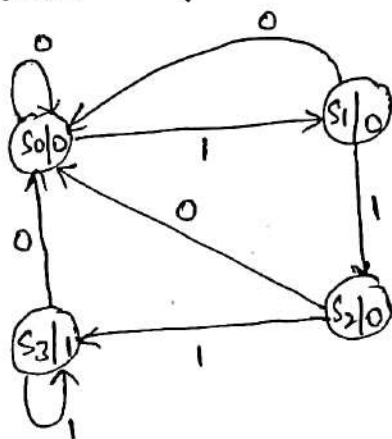
The design of a clocked sequential circuit starts from a set of specifications and culminates in a logic diagram (or) a list of boolean functions from which the logic diagram can be obtained.

The procedure for designing synchronous sequential circuits can be summarized by a list of steps.

- ① From the word description and specifications of the desired operation, derive a state diagram for the circuit.
- ② Reduce the no. of states if necessary
- ③ Assign binary values to the states
- ④ Obtain the binary coded state table
- ⑤ Choose the type of flip-flops to be used
- ⑥ Derive the simplified ff input equations and o/p equations
- ⑦ Draw the logic diagram.

Ex:- Design a circuit that detects a sequence of 3 (or) more consecutive 1's in a string of bits coming through an input line

Sol:- The state diagram for this type of circuit is as shown in fig.



It is derived by starting with state S_0 , the reset state.

If the input is 0, the ckt stays in S_0 , but if the input is 1, it goes to state S_1 to indicate that 1 was detected.

If the next input is 1, the ckt enters into S_2 to indicate the arrival of two

consecutive 1's but if the i/p is 0, the state goes back to S_0 . The 3rd consecutive 1 sends the circuit to state S_3 . If more 1's are detected the circuit stays in S_3 . Any 0 input sends the circuit back to S_0 . When the circuit is in state S_3 , the o/p is 1 otherwise it is equal to '0'.

Assign binary values $S_0 = 00$, $S_1 = 01$, $S_2 = 10$ & $S_3 = 11$

(26)

Synthesis using D flip-flops:-

Once the state diagram has been derived, the rest of the design follows a straightforward synthesis procedure.

The state table is derived from the state diagram with a sequential binary assignment. We choose 2 D flip-flops to represent the 4 states and label their o/p's as A and B. There is one input x and one o/p y .

State table:-

Present state		input x	Next state		output y
A	B		A	B	
0	0	0	0	0	0
0	0	1	0	1	0
0	1	0	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	1	1	1	0
1	1	0	0	0	1
1	1	1	1	1	1

The characteristic equation of the D FF is $Q(t+1) = D$. It means that next state values in the state table specify the D input condition for the FF.

The FF input equations can be obtained directly from the next state columns of A and B as

$A(t+1)$

$A \backslash Bx$	00	01	11	10
0			1	
1		1	1	

$$A(t+1) = D_A = Bx + Ax$$

$$y = AB$$

$B(t+1)$

$A \backslash Bx$	00	01	11	10
0		1		
1		1	1	

$D_B =$

$$B(t+1) = B'x + Ax$$

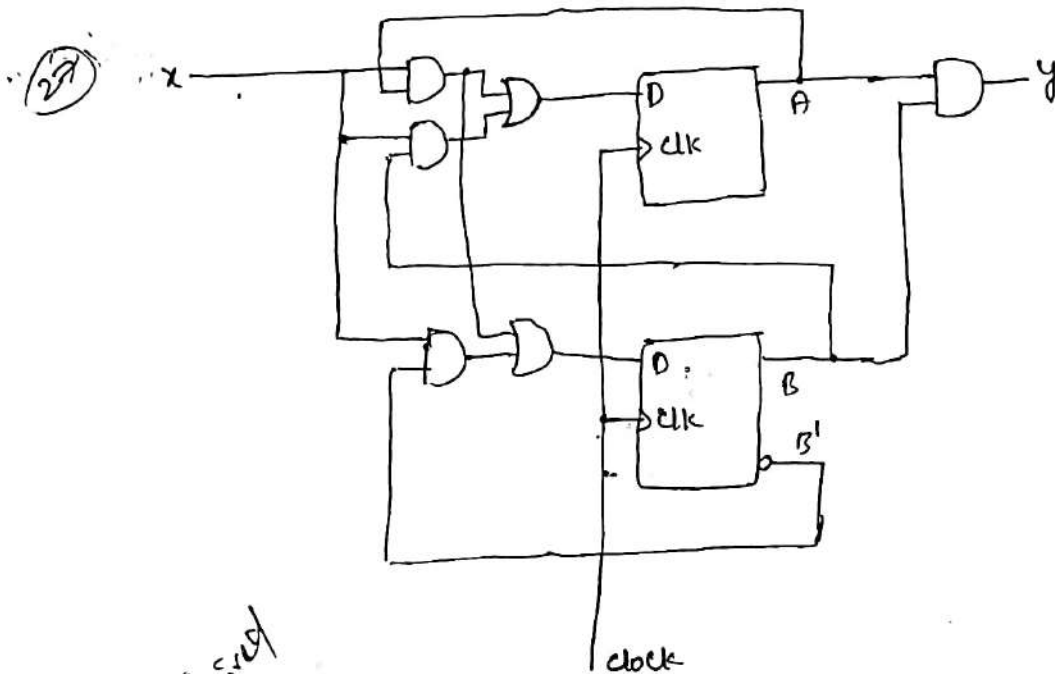


fig:- Logic diagram of Sequence detector

Synthesis using JK flip-flops:-

In this case, the Ht input equations must be evaluated from the present state to the next state transition derived from the excitation table

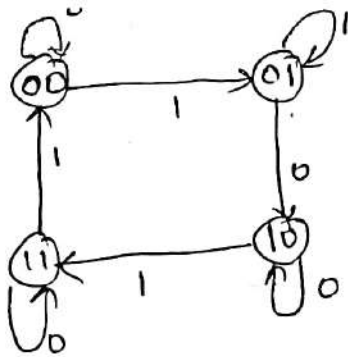
Excitation table of JK f/f is

Q_n	Q_{n+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

State table:-

present state		input x	next state		output y	Flipflop i/p's			
			A	B		J_A	K_A	J_B	K_B
0	0	0	0	0	0	0	X	0	X
0	0	1	0	1	0	0	X	1	X
0	1	0	0	0	0	0	X	X	1
0	1	1	1	0	0	1	X	X	1
1	0	0	0	1	0	X	1	0	X
1	0	1	1	1	0	X	0	1	X
1	1	0	0	0	1	X	1	X	1
1	1	1	1	1	1	X	0	X	0

Ex:-



Not required

Sol:- State table

Present state		input x	Next state		flip-flop inputs			
A	B		A	B	J_A	K_A	J_B	K_B
0	0	0	0	0	0	x	0	x
0	0	1	0	1	0	x	1	x
0	1	0	1	0	1	x	x	1
0	1	1	0	1	0	x	x	0
1	0	0	1	0	x	0	1	x
1	0	1	1	1	x	0	x	0
1	1	0	1	1	x	1	x	1
1	1	1	0	0	x	1	x	1

J_A

A	0	1	1	1
0	x	x	x	1
1	x	x	x	x

$$J_A = Bx'$$

K_A

A	0	1	1	1
0	x	x	x	x
1	x	1	x	x

$$K_A = Bx$$

J_B

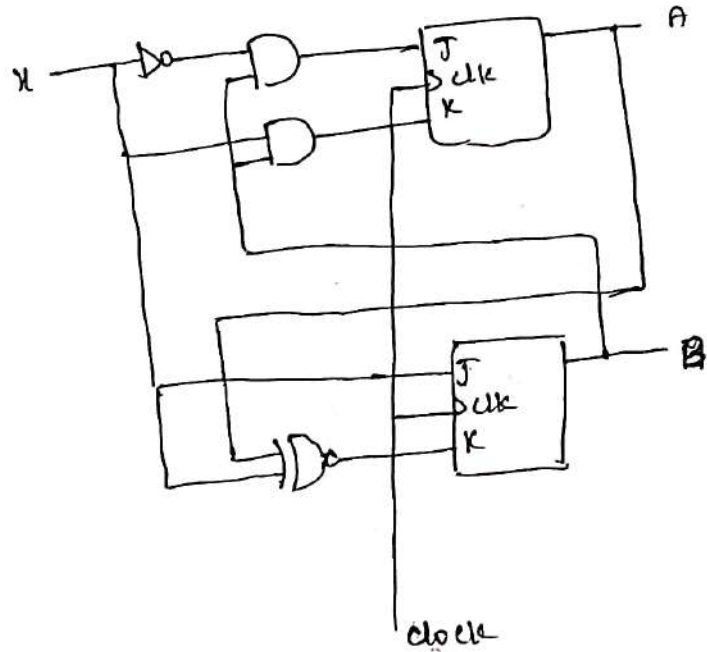
A	0	1	1	1
0	1	x	x	x
1	1	x	x	x

$$J_B = x$$

K_B

A	0	1	1	1
0	x	x	1	1
1	x	x	1	1

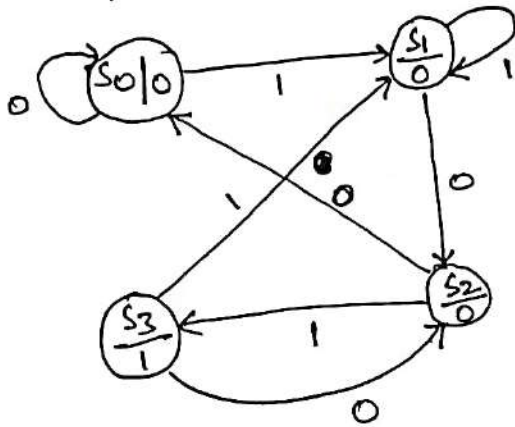
$$K_B = Ax + A'x' = (A \oplus x)'$$



Ex:- Design a mode type sequence detector to detect a serial input sequence of '101'.

Sol:-

The state diagram is derived by starting with state S_0 , the reset state. If the input is 0, the ckt stays in S_0 , but if the input is 1, it goes to state S_1 to indicate that '1' was detected.



If the next i/p is 0, the circuit enters into S_2 to indicate the sequence '10' is detected. When input is '1', the ckt remain in state S_1 because '1' is the 1st bit in sequence.

If the next input is '1', the circuit enters into S_3 to indicate the sequence 101 is detected and o/p must equal to 1. ~~Here, we cannot go back to state S_1 since o/p in state~~

~~S_1 is 0~~) In case if input is 0, the ckt enters into S_0 state to restart checking of input sequence.

State S_3 :- Since the sequence is detected, this is the last state. When i/p is 1, the ckt detects the 1st bit in the next sequence, hence ckt enters into state S_1 .

When input is 0, the ckt detected second bit in the overlapped sequence hence the ckt go to state S_2 .

Assign binary values $S_0=00$, $S_1=01$, $S_2=10$, $S_3=11$

Since there are 4 states, we need 2 flt's (choose JK flip-flops). Label the o/p's of flt's as A and B. There is one i/p & one o/p.

From the above state diagram, determine the state

table	Present State		Input 0 X	Next State A(t+1) B(t+1)		o/p Y	Flip-flop i/p's			
	A	B		A	B		J _A	K _A	J _B	K _B
	0	0	0	0	0	0	0	X	0	X
	0	0	1	0	1	0	0	X	1	X
	0	1	0	1	0	0	1	X	X	1
	0	1	1	0	1	0	0	X	X	0
	1	0	0	0	0	0	X	1	0	X
	1	0	1	0	0	0	X	1	0	X

29

1	0	1	1	1	0	x	0	1	x
1	1	0	1	0	1	x	0	x	1
1	1	1	0	1	1	x	1	x	0

J_A

A \ Bx	00	01	11	10
0				1
1	x	x	x	x

$J_A = Bx'$

K_A

A \ Bx	00	01	11	10
0	x	x	x	x
1	1		1	

$K_A = Bx' + Bx$

J_B

A \ Bx	00	01	11	10
0		1	x	x
1		1	x	x

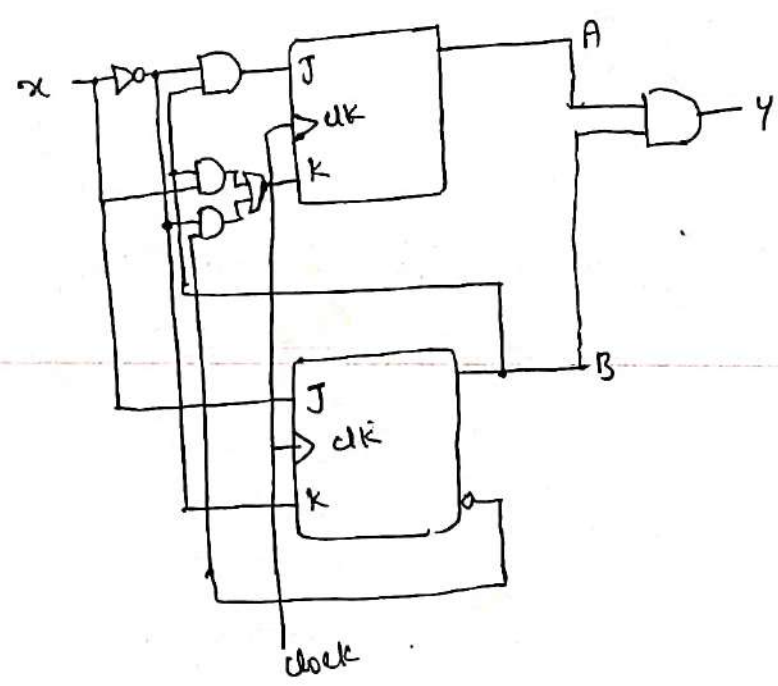
$J_B = x$

K_B

A \ Bx	00	01	11	10
0	x	x		1
1	x	x		1

$K_B = x'$

o/p equation $y = AB$



30 Counters:-

A register that goes through a prescribed sequence of states upon the application of input pulses (clock) is called a counter. The sequence of states may follow the binary number sequence (or) any other sequence of states. A counter that follows the binary number sequence is called a binary counter.

An n -bit binary counter consists of n flip-flops and can count in binary from 0 through $2^n - 1$.

Counters are available in two categories,

- ① ripple counters (or) Asynchronous counters
- ② Synchronous counters

In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops.

In a synchronous counter, the clock inputs of all flip-flops receive the common clock.

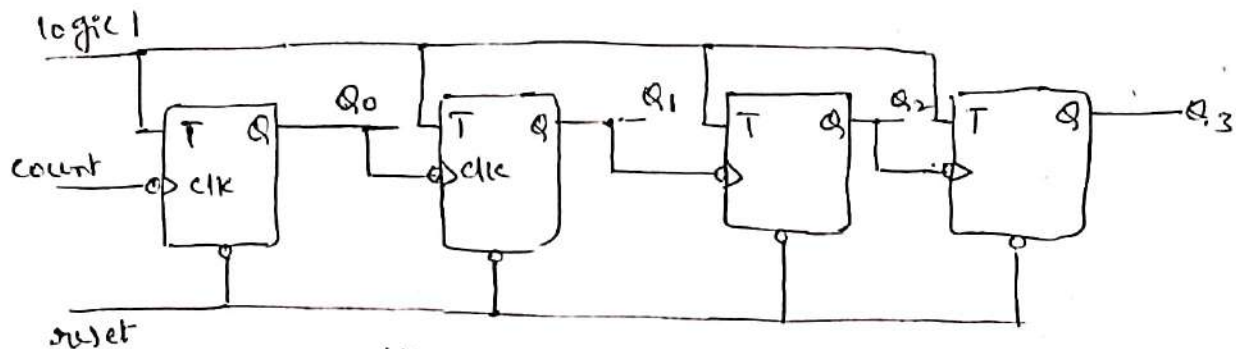
Ripple Counters (Asynchronous Counters):-

A binary ripple/asynchronous counter consists of a series connection of complementing flip-flops, with the output of each flip-flop connected to the clock input of the next higher order flip-flop. The flip-flop holding the LSB receives the incoming count (clock) pulse.

A complementing flip-flop can be obtained from a JK flip-flop with J & K inputs tied together (or) from a T flip-flop. A third alternative is to use a D flip-flop with the complement of Q connected to the D input.

The logic diagram of n -bit binary ripple counter is shown in fig. (a) & (b). The counter is constructed with complementing flip-flops of T-type in fig. (a) and D-type in fig. (b).

31



(a)
fig:- with T flip-flops

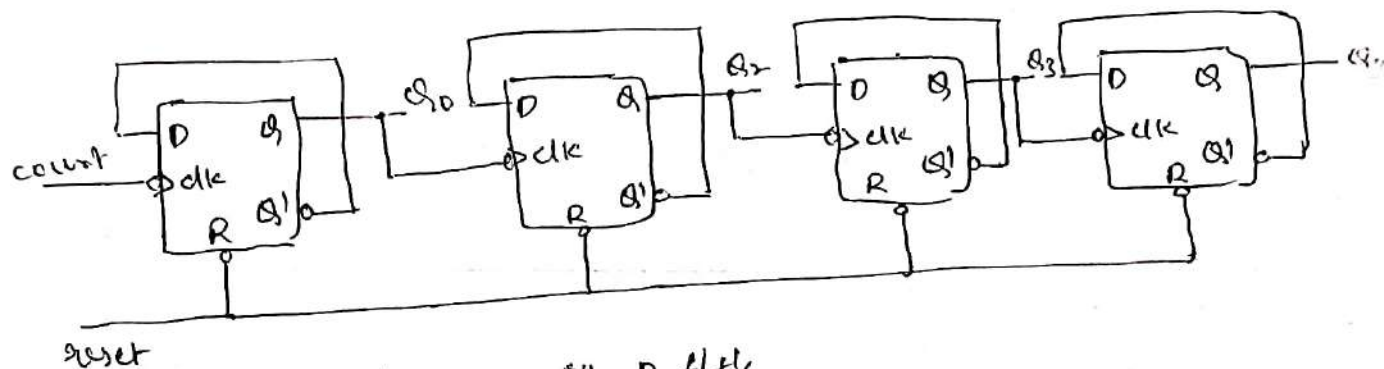


fig (b):- with D flip-flops

fig:- 4-bit binary ripple counter

The output of each flip-flop is connected to the clock input of next flip-flop in sequence. The flip-flop holding the least significant bit receives the incoming count (clock pulse). The T inputs of all the flip-flops are connected to a permanent 1, making each flip-flop complement if the clock input goes through a negative transition.

The counter starts with binary 0 and increments by 1 with each count pulse input. After the count of 15, the counter goes back to "0" to repeat the count. The timing diagram of 4-bit ripple carry adder is as follows.

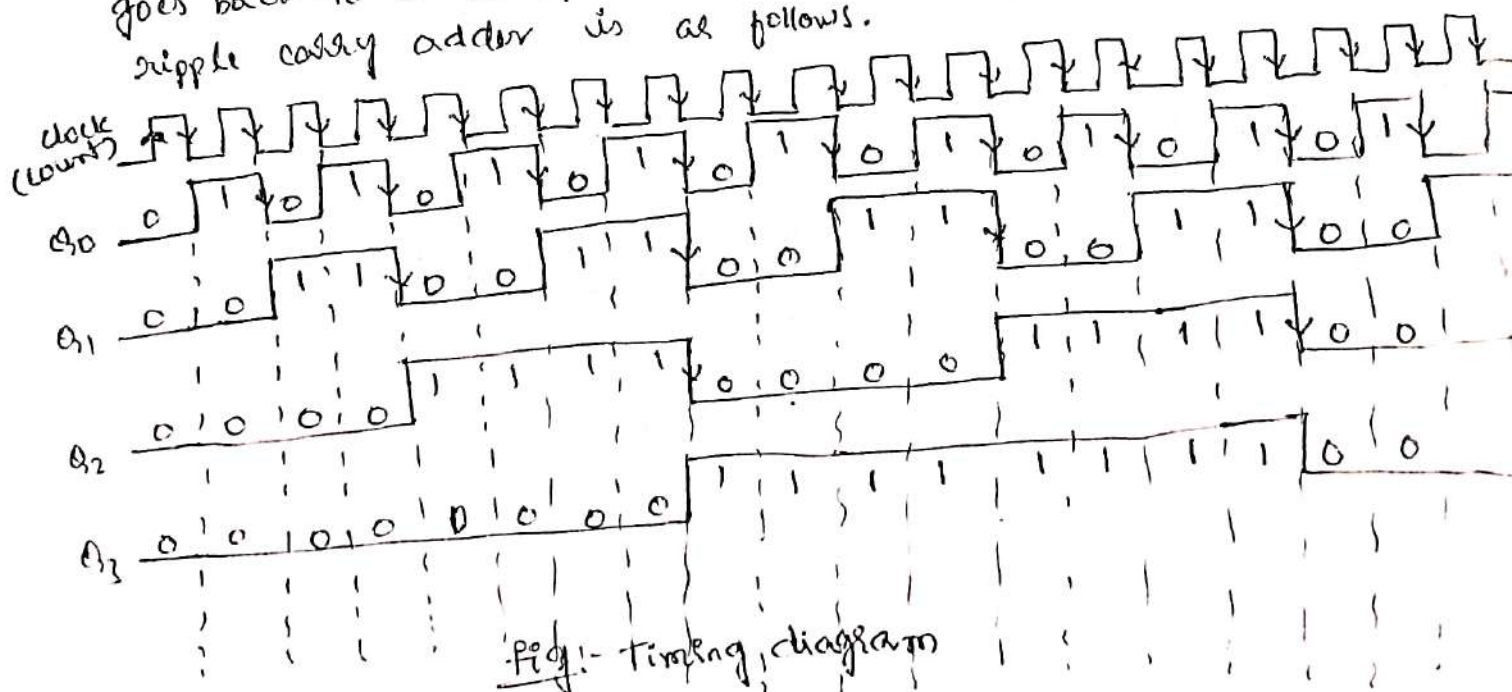
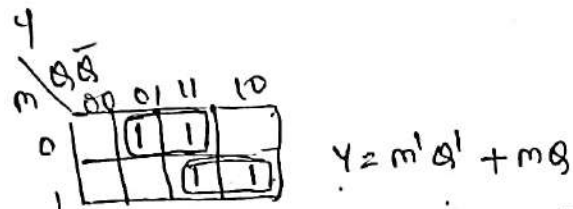
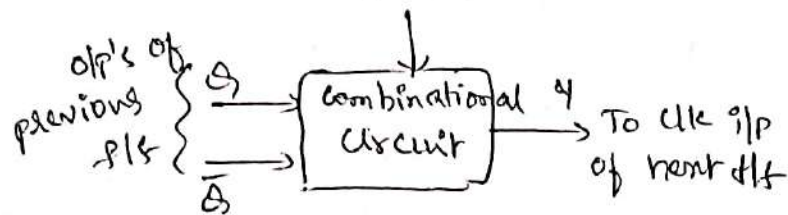


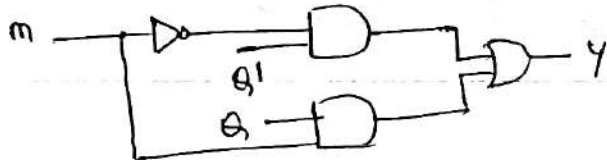
fig:- timing diagram

Asynchronous up/down Counter:-

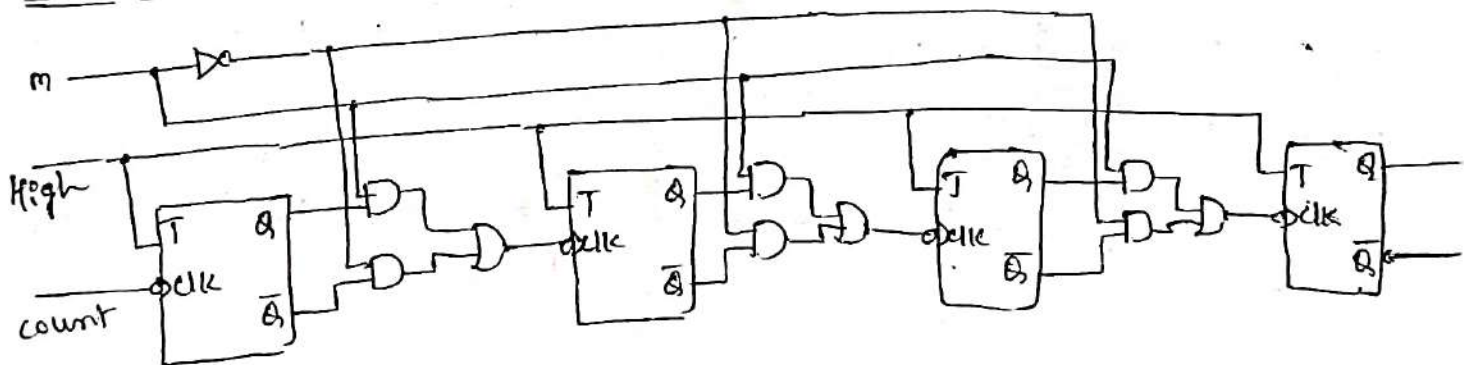
To form an asynchronous up/down counter, one control input say m is necessary to control the operation of up/down counter. When $m=0$, the counter will count up and when $m=1$, the counter will count down. To achieve this, the m input should be used to control whether the normal ff output (or) the inverted ff (\bar{Q}) is fed to drive the clock signal of successive stage ff as shown in fig mode control (m)



inputs			output (Y)	
m	Q	\bar{Q}		
m=0	0	0	0	$Y = \bar{Q}$ for down counting
	0	1	1	
	1	0	0	
	1	1	1	
m=1	0	0	0	$Y = Q$ for up count
	0	1	0	
	1	0	1	
	1	1	1	



4-bit up/down counter



34. Synchronous counters

In synchronous counters, the clock input of all the flip-flops receive a common clock which triggers all the flip-flops simultaneously.

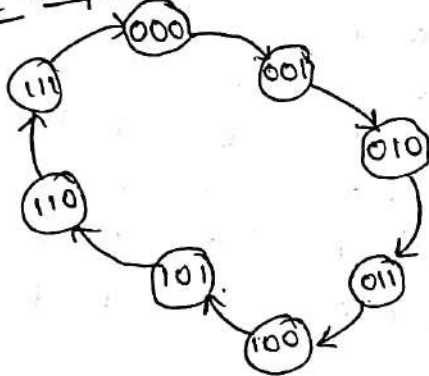
Upcounter:-

In up counter, the count value is incremented by 1 for the occurrence of every clock pulse.

3-bit counter (or) mod-8 counter:-

A 3-bit counter has 3 flip-flops. Let the 3 flip-flops are T flip-flops with output A, B, C. The state diagram of 3-bit counter is as follows:

State diagram



Excitation table of T

Q_n	Q_{n+1}	T
0	0	0
0	1	1
1	0	1
1	1	0

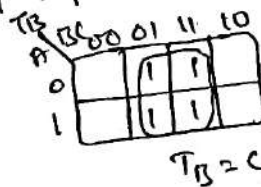
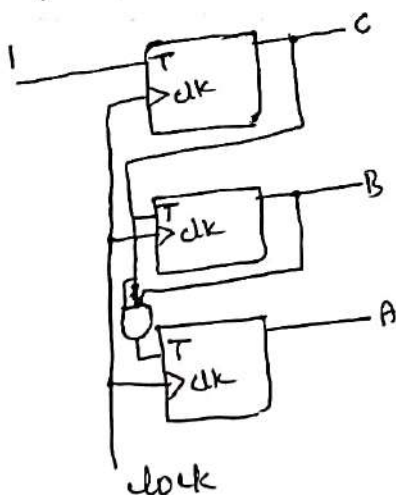
State table

Present State A B C	Next State A(t+1) B(t+1) C(t+1)			Flip inputs $T_A T_B T_C$		
	A(t+1)	B(t+1)	C(t+1)	T_A	T_B	T_C
0 0 0	0	0	1	0	0	1
0 0 1	0	1	0	0	1	1
0 1 0	0	1	1	0	0	1
0 1 1	1	0	0	1	1	1
1 0 0	1	0	1	0	0	1
1 0 1	1	1	0	0	1	1
1 1 0	1	1	1	0	0	1
1 1 1	0	0	0	1	1	1

Flip-flop input equations $T_C = 1$,

$$T_A = A'BC + ABC = BC$$

Logic diagram



(35)

For a 4-bit counter, let the o/p's of flip-flops are A, B, C and D.

Its flip-flop i/p equations can be written as

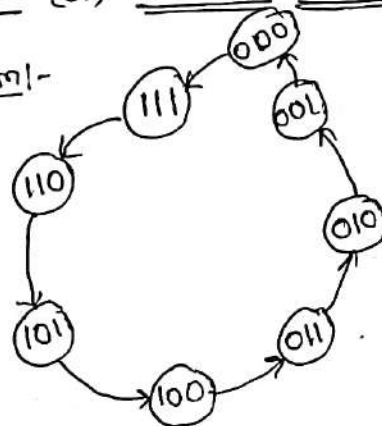
$$T_D = 1, T_C = D, T_B = CD, T_A = BCD$$

Down counter

In down counter, the count value is decremented by 1 for the occurrence of every clock pulse.

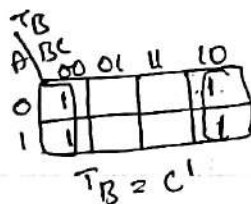
3-bit counter (or) mod-8 counter

State diagram:-



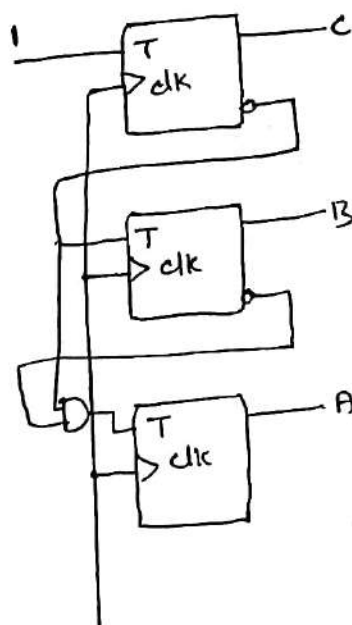
flip-flop input equations

$$T_C = 1$$



$$T_A = ABC' + A'B'C' \\ = (A' + A)B'C' = B'C'$$

Logic diagram



State Table

present state	next state			flf inputs		
	A(t+1)	B(t+1)	C(t+1)	T _A	T _B	T _C
1 1 1	1	1	0	0	0	1
1 1 0	1	0	1	0	1	1
1 0 1	1	0	0	0	0	1
1 0 0	0	1	1	1	1	1
0 1 1	0	1	0	0	0	1
0 1 0	0	0	1	0	1	1
0 0 1	0	0	0	0	0	1
0 0 0	1	1	1	1	1	1

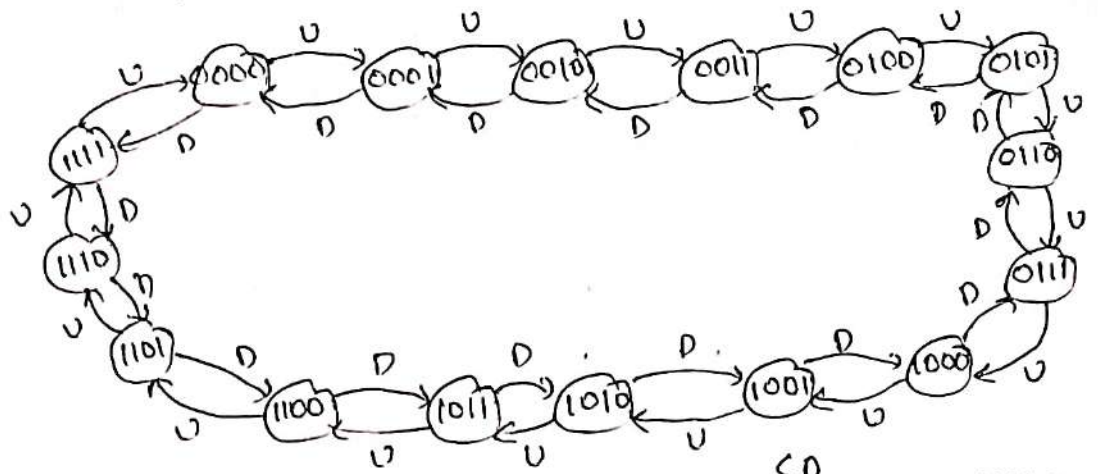
for a 4-bit ^{down} counter (mod-16 counter), let the o/p's of f/f's are A, B, C and D.

Its flip-flop input equations can be written as

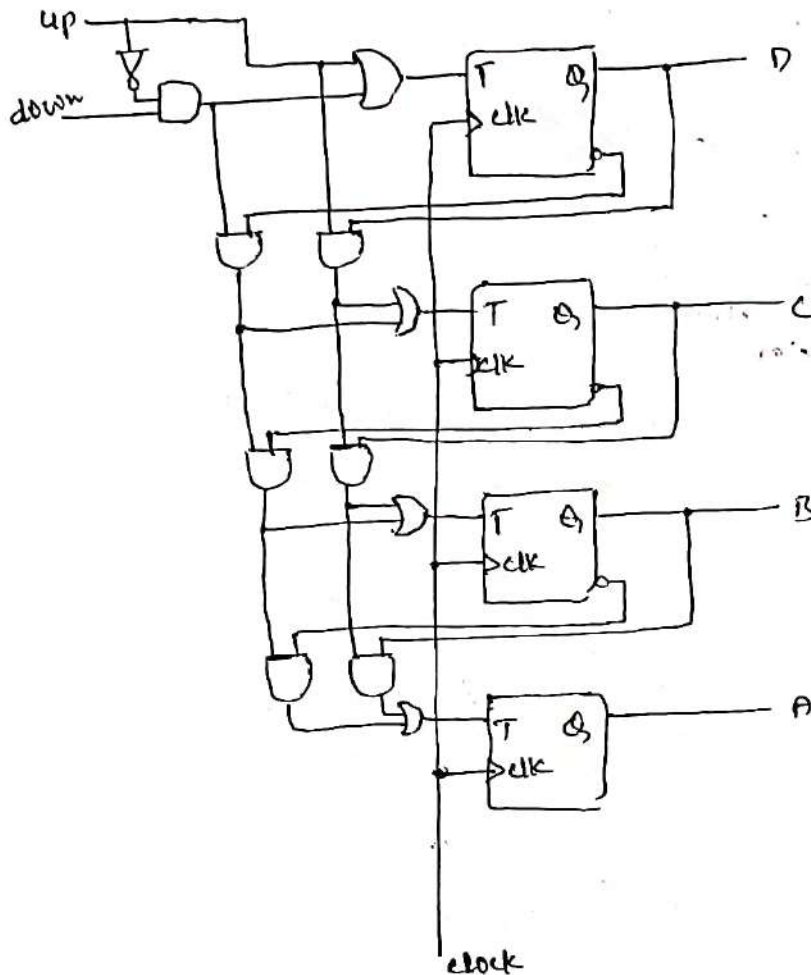
$$T_D = 1, T_C = D', T_B = A'C', T_A = B'C'$$

Ex-1: Design a mod-16 synchronous up/down counter using T flip-flops

36



up counter flt input equations $T_D = 1$, $T_C = D$, $T_B = \overline{C}D$, $T_A = ABCD$
 down counter flt input equations $T_D = 1$, $T_C = \overline{D}$, $T_B = \overline{C}\overline{D}$, $T_A = \overline{B}\overline{C}\overline{D}$



If $up=0$ & $down=0$, $T_A=0$,
 $T_B=0$, $T_C=0$ & $T_D=0$ so
 the o/p does not change in
 next state.

If $up=1$ & $down="X"$ $T_D=1$,
 $T_C=0$, $T_B=CD$ & $T_A=BCD$

If $up=0$ & $down=1$, $T_D=1$,
 $T_C=\overline{D}$, $T_B=\overline{C}\overline{D}$ & $T_A=\overline{B}\overline{C}\overline{D}$

Binary counter with parallel load:-

Counters employed in digital systems often require parallel load capability for transferring an initial binary number into the counter prior to the count operation.

Following fig shows a logic diagram of a 4-bit counter that has a parallel load capability and can operate.

as a counter.

(33)

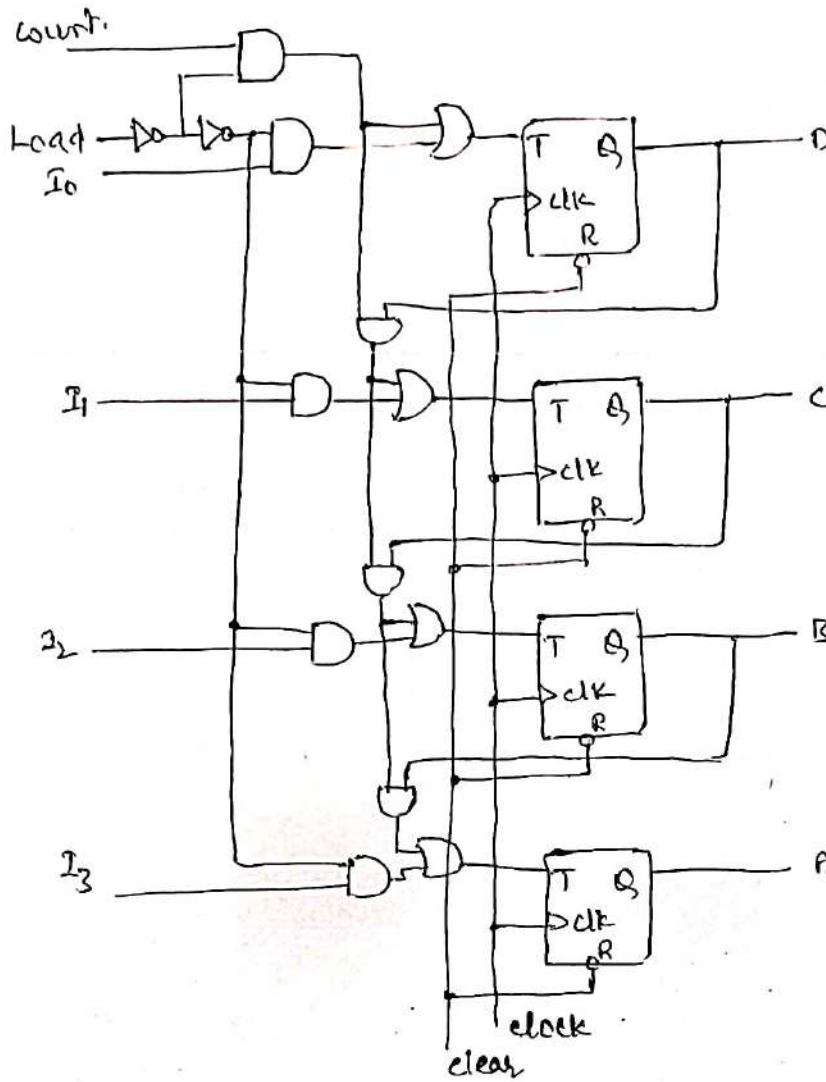


fig:- 4-bit binary counter with parallel load

clear	clk	load	count	Function
0	x	x	x	clear to zero
1	↑	1	x	Load inputs
1	↑	0	1	count next binary state
1	↑	0	0	No change.

- when the input load control is equal to 1 disables the count operation and causes a transfer of data from the 4 data i/p's into the 4 H's.
- If both control inputs (load & count) are "0", the clock pulses do not change the state of registers.
- The operation of the counter is summarized in table. The 4 control i/p's: clear, clock, load & count determine the next state.

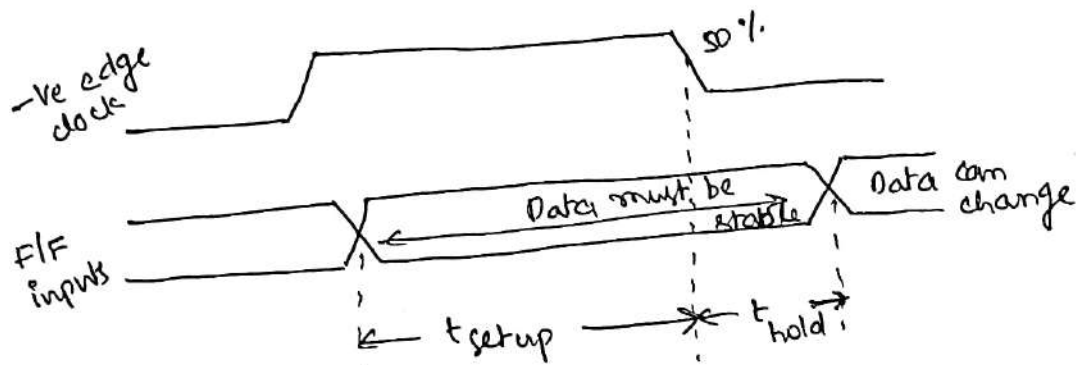
58. Flip-Flop Timing

In designing practical digital systems with ff 's we have to consider 3 important parameters

- ① setup time
- ② hold time
- ③ Propagation delay

Set up time:-

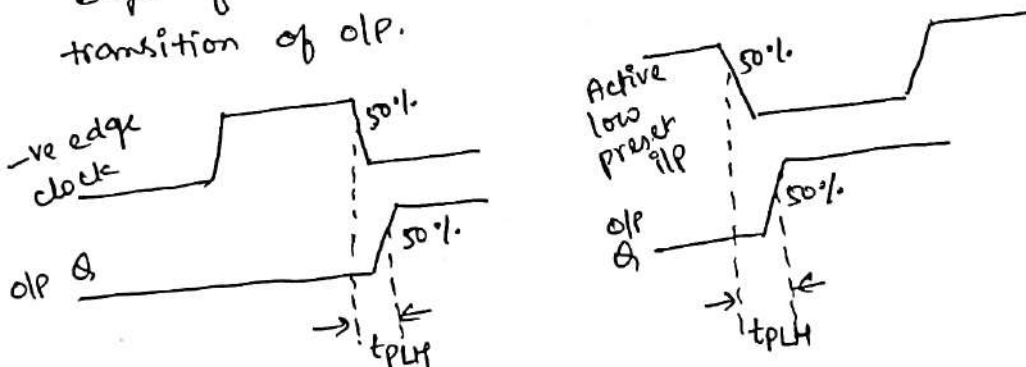
It is the minimum time required to maintain a constant voltage levels (data) at the excitation if 's ^{of ff} prior to the triggering edge of clock pulse in order for the levels to be reliably clocked into ff . It is denoted as t_{setup} .



Hold time:- It is the min. time for which the voltage levels (data) at the excitation inputs must remain constant after the triggering edge of the clock pulse in order for the levels to be reliably clocked into ff . It is denoted by t_{hold} .

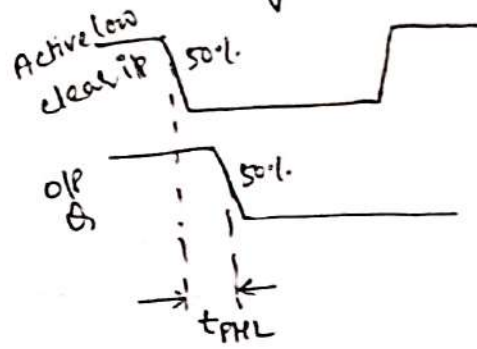
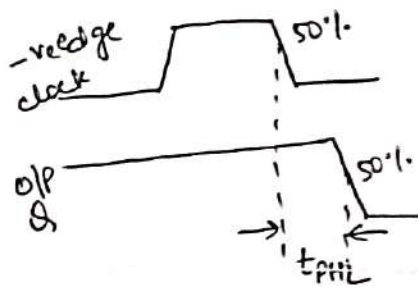
Propagation delay:- It is the time required to change the of after application of the if . Several propagation delays are important in operation of ff .

① propagation delay (t_{PLH}):- It is measured from the triggering edge of clock pulse (or) the preset if to the low to high transition of of .



(3a)

propagation delay (t_{PHL}):- It is measured from triggering edge of the clock pulse (or) clear ip to the high to low transition of the op.



① Differences blw combinational circuits & sequential circuits

combinational circuits

① In combinational circuits, the o/p variables are dependent on the present values of i/p variables.

② memory unit is not present in combinational circuits.

③ combinational circuits are easy to design

④ combinational circuits are faster in speed because the delay b/w input and output is due to propagation delay of gates.

Ex:- parallel adder

sequential circuits

① In sequential circuits, the o/p variables ~~not~~ depend not only on the present values of i/p v'bles but also the past values of i/p variables.

② memory unit is required to store the past history of i/p variables in the sequential circuit.

③ Sequential circuits are comparatively harder to design.

④ Sequential circuits are slower than combinational circuits.

Ex:- Serial adder.

② Differences blw synchronous & asynchronous sequential circuits

synchronous sequential circuits

① In synchronous circuits, memory elements are clocked fl's.

② In synchronous circuits, the change in i/p signals can affect memory element upon the activation of clock signal.

③ Easier to design

④ The max. operating speed of clock depends on time delays involved.

asynchronous sequential circuits

① In Asynchronous circuits memory elements are either latches (or) time delay elements.

② In asynchronous circuits, change in input signals can affect memory element at any instant of time.

③ difficult to design

④ Because of absence of clock, asynchronous circuits can operate faster than synchronous circuits.

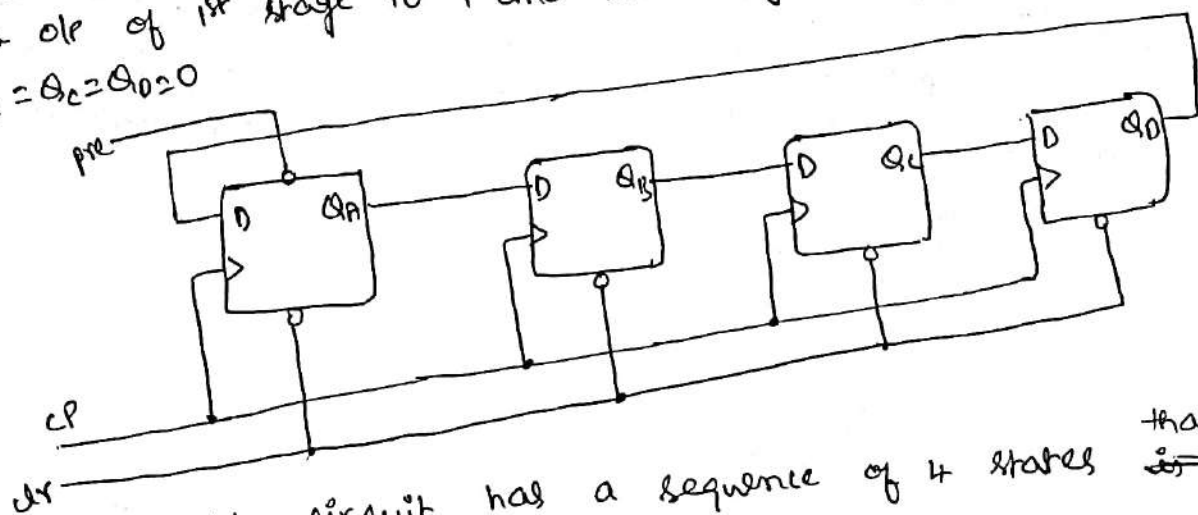
(vi) Other Counters:-

Counters can be designed to generate any desired sequence of states. A divide by N counter (or modulo- N counter) is a counter that goes through a repeated sequence of N states. Counters are used to generate timing signals to control the sequence of operations in a digital system. Counters can also be constructed by means of shift registers. In this section, we see a few examples of nonbinary counters.

Ring Counter:-

Timing signals that control the sequence of operations in a digital system can be generated by a shift register (or by a counter with a decoder).

A ring counter is a circular shift register with only one flip being set at any particular time, all others are cleared. The single bit is shifted from one flip to the next to produce the sequence of timing signals. Fig shows a 4-bit shift register connected as a ring counter. The initial value of the register is 1000 and requires preset/clear flip's. The "clr" followed by "pre" makes the output of 1st stage to '1' and remaining flip's are '0' i.e. $Q_A = 1$ & $Q_B = Q_C = Q_D = 0$



The circuit has a sequence of 4 states that are listed in table.

clock pulse	Q_A	Q_B	Q_C	Q_D
0	1	0	0	0
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	1	0	0	0

(12)

As shown in Table, '1' is retained in the counter and simply shifted around the ring, advancing one stage for each clock pulse. In this case 4 stages of f/f's are used, so a sequence of 4 states is produced and repeated. Fig shows the timing sequence for a 4-bit ring counter.

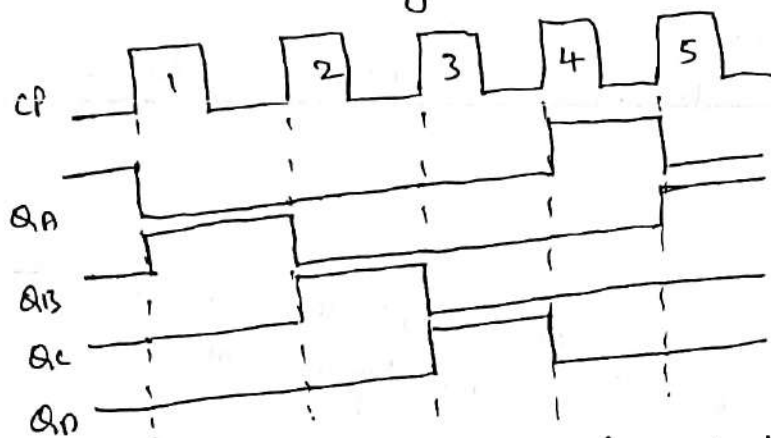
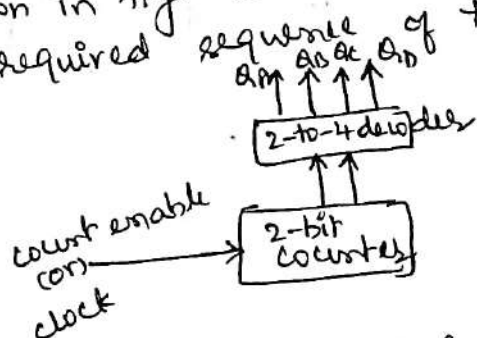


Fig:- Timing sequence for a 4-bit ring counter.

The ring counter can be used for counting the no. of pulses. The no. of pulses counted is read by noting which f/f is in state '1'. No decoding circuitry is required.

→ The ring counters suffer from one major problem if its single '1' output is lost due to a temporary hardware problem, the counter goes to state 0000 and stays there forever.

Alternatively, the timing signals can be generated by a 2-bit counter that ~~just~~ and 2 to 4 decoder. The decoder shown in fig decodes the 4 states of counter and generates the required sequence of timing signals.



Note! - To generate 2^n timing signals, we need either a shift register with 2^n f/f's (or) an n -bit binary counter together with n to 2^n decoder.

It is also possible to generate the timing signals with a combination of a shift register and a decoder. In this the no. of f/f's is less than that in a ring counter and the decoder required only 2-input ^{AND} gates. This combination is called a Johnson counter (or) switch tail counter.

Johnson counter

The no. of states can be doubled if the shift register is connected as a switch tail ring counter. A switch-tail ring counter is a circular shift register with the complemented Q of last FF connected to the 1^{st} FF as shown in fig.

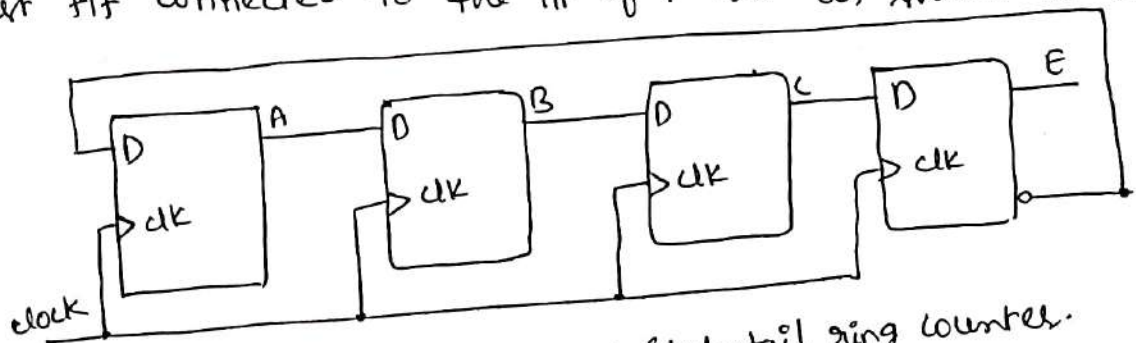


fig:- 4-stage switch tail ring counter.

The register shifts its contents once to the right with every clock pulse and at the same time the complemented value of E FF is transferred into the A FF . Starting from a cleared state, counter goes through 8 states as listed in table.

Sequence number	FF outputs				AND gate required for Q
	A	B	C	E	
1	0	0	0	0	$A'E'$
2	1	0	0	0	AB'
3	1	1	0	0	BC'
4	1	1	1	0	CE'
5	1	1	1	1	AE
6	0	1	1	1	$A'B$
7	0	0	1	1	$B'C$
8	0	0	0	1	$C'E$

Starting from all 0's, each shift operation inserts 1s from the left until the register is filled with all 1's. In the next sequences, 0's are inserted from the left until register is again filled with all 0's.

A Johnson counter is a k -bit switch tail ring counter with $2k$ decoding gates to provide Q 's for $2k$ timing signals. The 8 AND gates listed in the table when connected to the circuit, will complete the construction of Johnson counter.

①

UNIT-II

GATE LEVEL MINIMISATION

* We know that the boolean functions can be realized using logic gates. The total number of logic gates and literals can be reduced, if the boolean function is simplified. The simplification of a boolean function is required for reducing the complexity and cost of the designing of its logic circuit.

* During the process of simplification using boolean algebra one must know the boolean laws, rules, properties and theorems thoroughly. And also it is required to predict the successive steps to get the simplest expression.

* The map method gives us the systematic procedure to simplify the given boolean expression. It is also called as karnaugh map method (or) k-map method.

* The map method was first proposed by Veitch and modified by karnaugh. And hence map method is also called as Veitch diagram (or) karnaugh-map.

The map method (or) karnaugh-map (or) K-map method:

→ The K-map is a diagram made of square boxes. Each square box is called as a cell that represents either a minterm or a max term.

→ The simplified function produced using K-map is present in any one of the standard forms i.e either in product of sums (POS) form or in sum of products form.

→ The simplified function should have less number of terms and each term should have minimum number of literals.

PLDs

1. INTRODUCTION:

An IC that contains large numbers of gates, flip-flops, etc. that can be configured by the user to perform different functions is called a Programmable Logic Device (PLD). It permits elaborate digital logic designs to be implemented by the user on a single device. The internal logic gates and/or connections of PLDs can be changed/configured by a programming process.

Comparison: programmable logic Vs fixed logic

The fixed logic system has circuits whose configurations are permanent. Their instructions perform only a fixed set of operations repeatedly. Once manufactured and programmed, the logic cannot be changed. This system is a fantastic asset for repeated tasks.

But one tiny mistake in the manufacturing process like uploading the wrong code in the device, and the entire system is discarded, and a new design is developed. That's quite some risk that companies aren't willing to take unless necessary. Additionally, fixed logic does not allow the users to expand or build on their existing functionalities.

Thus, we need something more flexible, easy to work, and more cost-efficient. Thus, programmable logic comes to the rescue. It is easy-to-program, affordable and equipped with better features. Inexpensive software is used to develop, code and test the required design. This design is then programmed into a device and tested in a live electronic circuit.

The corresponding performance then decides if the logic needs to be altered, or if the prototype is fit to be determined as the final design itself. The fixed logic system thus offers limited usability; a programmable logic seems more feasible and beneficial.

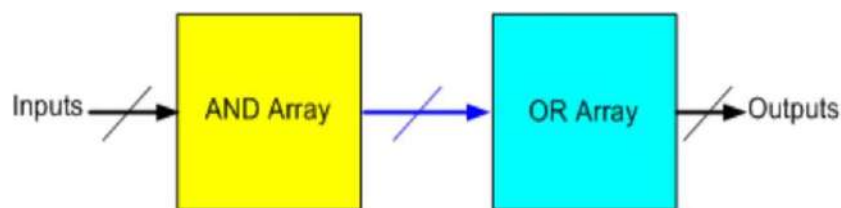
Implementing Boolean functions:

Every Boolean logic can be decomposed into product-of-sum (POS) or sum-of-product by Karnaugh map(k-map),

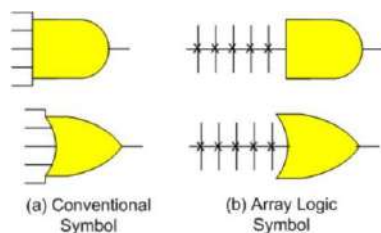
$$S = A \oplus B \oplus C = \overline{A}\overline{B}C + \overline{A}B\overline{C} + A\overline{B}\overline{C} + ABC$$

$$= (A + B + \overline{C})(A + \overline{B} + C)(\overline{A} + B + C)(\overline{A} + \overline{B} + \overline{C})$$

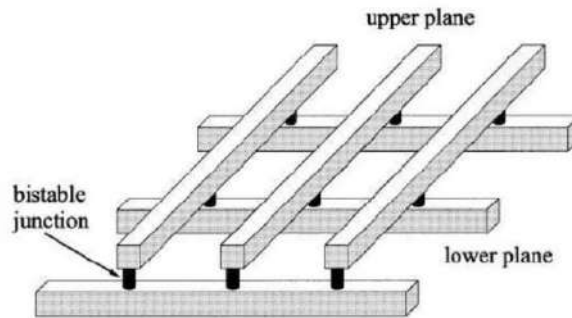
PLDs are typically built with an array of AND gates (AND-array) and an array of OR gates (OR-array) to implement the sum-of-products as shown in figure.



In order to show the internal logic diagram for such technologies in a concise form, it is necessary to have special symbols for array logic. Figure shows the conventional and array logic symbols for a multiple input AND and a multiple input OR gate.



One of the simplest programming technologies is to use fuses. In the original state of the device, all the fuses are intact. Programming the device involves blowing those fuses along the paths that must be removed in order to obtain the particular configuration of the desired logic function. Anti-fuse employs a thin barrier of non-conducting amorphous silicon between two metal conductors. Usually in mesh structure. When a sufficiently high voltage is applied across the amorphous silicon it is turned into a polycrystalline silicon-metal alloy with a low resistance, which is conductive



Problems of using standard ICs: Problems of using standard ICs in logic design are that they require hundreds or thousands of these ICs, considerable amount of circuit board space, a great deal of time and cost in inserting, soldering, and testing. Also require keeping a significant inventory of ICs.

Advantages of using PLDs: Advantages of using PLDs are less board space, faster, lower power requirements (i.e., smaller power supplies), less costly assembly processes, higher reliability (fewer ICs and circuit connections means easier troubleshooting), and availability of design software.

Types of PLDs:

PLDs are broadly classified into simple and complex programmable logic devices

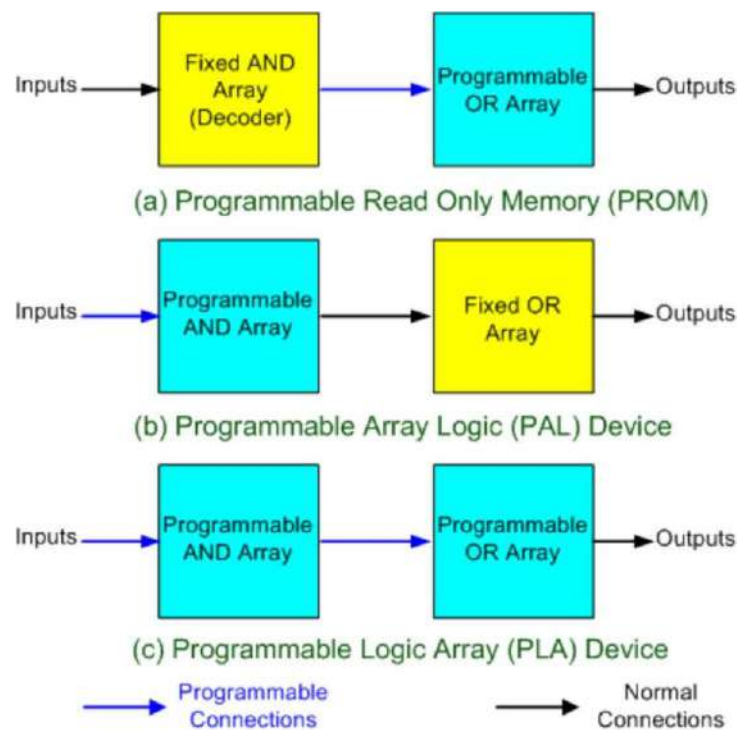
Further, this is grouped as,

- SPLDs (Simple Programmable Logic Devices)
 - ROM (Read-Only Memory)
 - PLA (Programmable Logic Array)
 - PAL (Programmable Array Logic)
 - GAL (Generic Array Logic)

- HCPLD (High Capacity Programmable Logic Device)
 - CPLD (Complex Programmable Logic Device)
 - FPGA (Field-Programmable Gate Array)

Programmable Connections in PLDs:

The programmable connections of AND-OR arrays for different types of PLDs are described here. Figure shows the locations of the programmable connections for the three types.



The PROM (Programmable Read Only Memory) has a fixed AND array (constructed as a decoder) and programmable connections for the output OR gates array. The PROM implements Boolean functions in sum-of-minterms form. The PAL (Programmable Array Logic) device has a programmable AND array and fixed connections for the OR array. The PLA (Programmable Logic Array) has programmable connections for both AND and OR arrays. So it is the most flexible type of PLD.

Applications of Programmable Logic Devices:

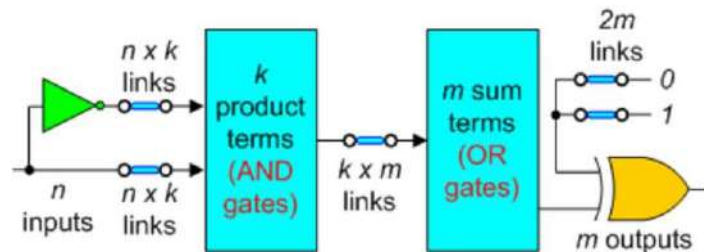
- Glue Logic
- State Machines
- Counters

- Synchronization
- Decoders
- Bus Interfaces
- Parallel-to-Serial
- Serial-to-Parallel

2. PROGRAMMABLE LOGIC ARRAY (PLA):

In PLAs, instead of using a decoder as in PROMs, a number (k) of AND gates is used where $k < 2^n$, (n is the number of inputs). Each of the AND gates can be programmed to generate a product term of the input variables and does not generate all the minterms as in the ROM. The AND and OR gates inside the PLA are initially fabricated with the links (fuses) among them. The specific Boolean functions are implemented in sum of products form by opening appropriate links and leaving the desired connections.

A block diagram of the PLA is shown in the figure. It consists of n inputs, m outputs, and k product terms. The product terms constitute a group of k AND gates each of $2n$ inputs. Links are inserted between all n inputs and their complement values to each of the AND gates. Links are also provided between the outputs of the AND gates and the inputs of the OR gates.



Since PLA has m-outputs, the number of OR gates is m. The output of each OR gate goes to an XOR gate, where the other input has two sets of links, one connected to logic 0 and other to logic 1. It allows the output function to be generated either in the true form or in the complement form. The output is inverted when the XOR input is connected to 1 (since $X \oplus 1 = \bar{X}$). The output does not change when the XOR input is connected to 0 (since $X \oplus 0 = X$). Thus, the total number of programmable links is $2n \times k + k \times m + 2m$.

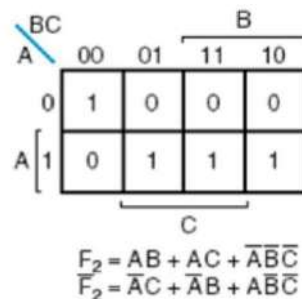
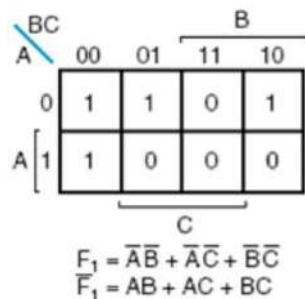
The size of the PLA is specified by the number of inputs (n), the number of product terms (k), and the number of outputs (m), (the number of sum terms is equal to the number of outputs).

Example 1:

Implement the combinational circuit having the shown truth table, using PLA.

A	B	C	F ₁	F ₂
0	0	0	1	1
0	0	1	1	0
0	1	0	1	0
0	1	1	0	0
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	0	1

Each product term in the expression requires an AND gate. To minimize the cost, it is necessary to simplify the function to a minimum number of product terms.



Designing using a PLA, a careful investigation must be taken in order to reduce the distinct product terms. Both the true and complement forms of each function should be simplified to see which one can be expressed with fewer product terms and which one provides product terms that are common to other functions.

The combination that gives a minimum number of product terms is,

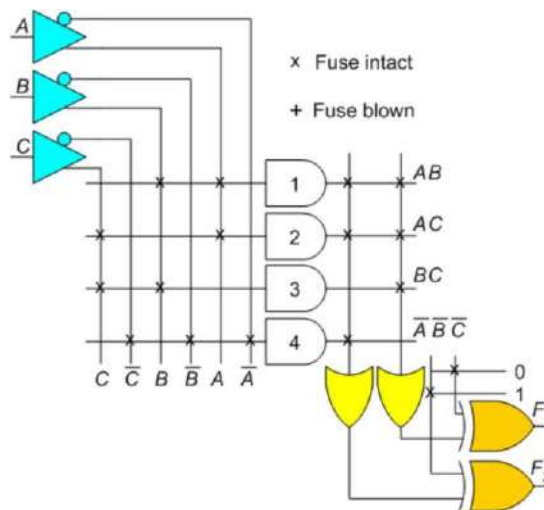
$$F_1 = AB + AC + BC \text{ or } F_1 = (AB + AC + BC)'$$

$$F_2 = AB + AC + A'B'C'$$

This gives only 4 distinct product terms: AB, AC, BC, and $A'B'C'$. So the PLA table will be as follows,

PLA programming table					
	Product term	Inputs A B C	Outputs		
			(C) F ₁	(T) F ₂	
AB	1	1 1 –	1	1	
AC	2	1 – 1	1	1	
BC	3	– 1 1	1	–	
$\overline{A}\overline{B}\overline{C}$	4	0 0 0	–	1	

For each product term, the inputs are marked with 1, 0, or – (dash). If a variable in the product term appears in its normal form (unprimed), the corresponding input variable is marked with a 1. A 1 in the Inputs column specifies a path from the corresponding input to the input of the AND gate that forms the product term. A 0 in the Inputs column specifies a path from the corresponding complemented input to the input of the AND gate. A dash specifies no connection.



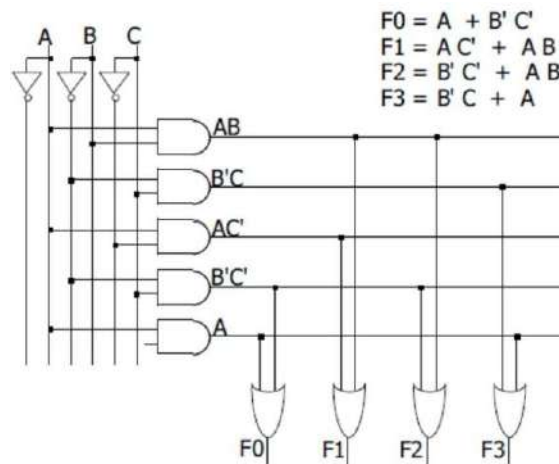
The appropriate fuses are blown and the ones left intact form the desired paths. It is assumed that the open terminals in the AND gate behave like a 1 input.

In the Outputs column, a T (true) specifies that the other input of the corresponding XOR gate can be connected to 0, and a C (complement) specifies a connection to 1.

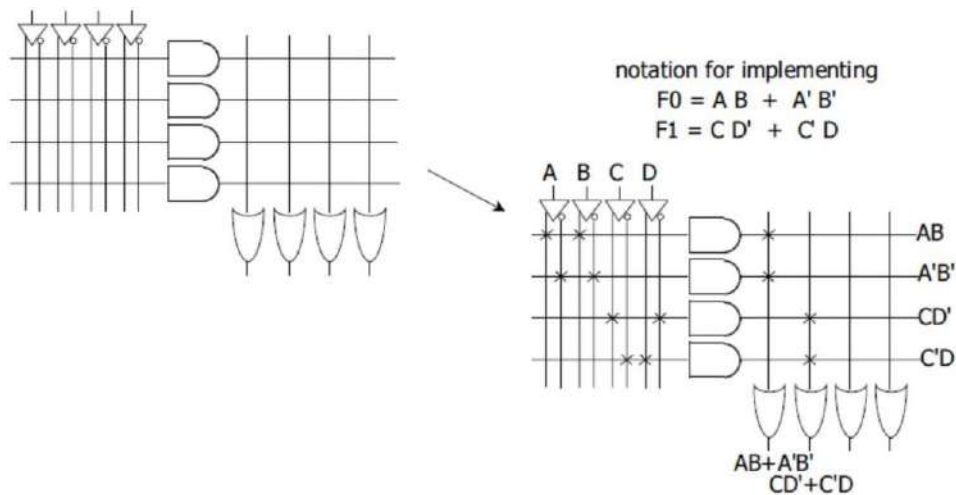
Note that output F_1 is the normal (or true) output even though a C (for complement) is marked over it. This is because F_1' is generated with AND-OR circuit prior to the output XOR. The output XOR complements the function F_1' to produce the true F_1 output as its second input is connected to logic 1.

Example 2:

All possible connections are available *before programming* as follows,



Unwanted connections are blown in the fuse (normally connected, break the unwanted ones) and in the anti-fuse (normally disconnected, make the wanted ones) *after programming for the given example* as follows,



Limitations of PLAs

PLAs come in various sizes. Typical size is 16 inputs, 32 product terms, 8 outputs

- Each AND gate has large fan-in. This limits the number of inputs that can be provided in a PLA
- 16 inputs forms 2^{16} , possible input combinations; only 32 permitted (since 32 AND gates) in a typical PLA
- 32 AND terms permitted large fan-in for OR gates as well
 - This makes PLAs slower and slightly more expensive than some alternatives to be discussed shortly
- 8 outputs could have shared min-terms, but not required

Applications of PLA:

- PLA is used to provide control over datapath.
- PLA is used as a counter.
- PLA is used as a decoders.
- PLA is used as a BUS interface in programmed I/O.

3. PROGRAMMABLE READ ONLY MEMORY (PROM):

Read Only Memory (ROM) is a memory device, which stores the binary information permanently. 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. The input lines to the AND array are hard-wired and the output lines to the OR array are programmable. Thus, we generate 2^n product terms using 2^n AND gates having n inputs each, using $n \times 2^n$ decoder. This decoder generates ' n ' min-terms. Each AND gate generates one of the possible AND products (i.e., min-terms).

Given a $2^k \times n$ ROM, we can implement ANY combinational circuit with at most k inputs and at most n outputs. Because,

- k -to- 2^k decoder will generate all 2^k possible min-terms
- Each of the OR gates must implement a $\Sigma m()$
- Each $\Sigma m()$ can be programmed

The procedure for implementing a ROM-based circuit is as follows for the given example,

$$f(a,b,c) = a'b' + abc$$

$$g(a,b,c) = a'b'c' + ab + bc$$

$$h(a,b,c) = a'b' + c$$

and its solution can be obtained as,

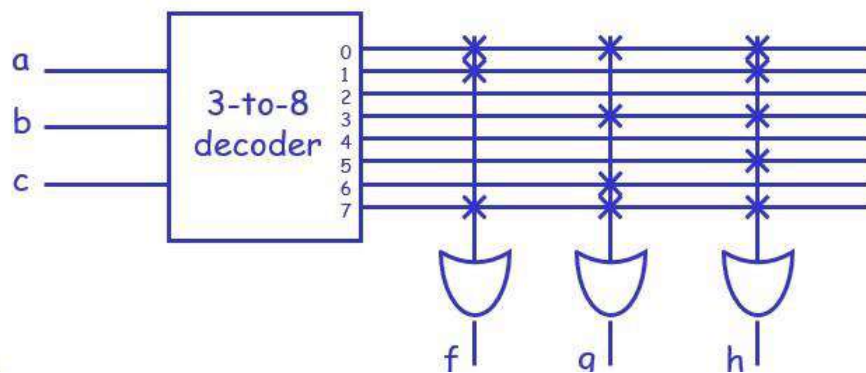
Express $f()$, $g()$, and $h()$ in $\Sigma m()$ format (use truth tables)

Program the ROM based on the 3 $\Sigma m()$'s

Example:

There are 3 inputs and 3 outputs, thus we need a 8x3 ROM block.

- $f = \Sigma m(0, 1, 7)$
- $g = \Sigma m(0, 3, 6, 7)$
- $h = \Sigma m(0, 1, 3, 5, 7)$



Another practical application of PROM device is BCD to 7 Segment Display Controller and the corresponding input and output relationship are shown in the following table.

A B C D	C0	C1	C2	C3	C4	C5	C6
0 0 0 0	1	1	1	1	1	1	0
0 0 0 1	0	1	1	0	0	0	0
0 0 1 0	1	1	0	1	1	0	1
0 0 1 1	1	1	1	1	0	0	1
0 1 0 0	0	1	1	0	0	1	1
0 1 0 1	1	0	1	1	0	1	1
0 1 1 0	1	0	1	1	1	1	1
0 1 1 1	1	1	1	0	0	0	0
1 0 0 0	1	1	1	1	1	1	1
1 0 0 1	1	1	1	0	0	1	1
1 0 1 0	X	X	X	X	X	X	X
1 0 1 1	X	X	X	X	X	X	X
1 1 0 0	X	X	X	X	X	X	X
1 1 0 1	X	X	X	X	X	X	X
1 1 1 0	X	X	X	X	X	X	X
0 1 1 1	X	X	X	X	X	X	X

Comparison: ROM Vs PLA

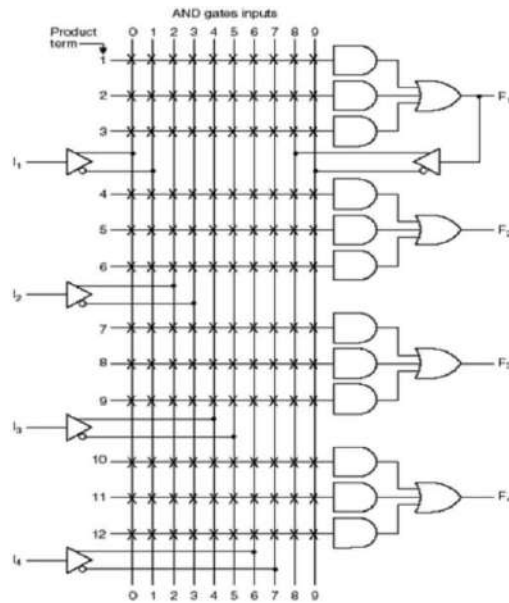
- ❑ ROM approach advantageous when
 - design time is short (no need to minimize output functions)
 - most input combinations are needed (e.g., code converters)
 - little sharing of product terms among output functions
- ❑ ROM problems
 - size doubles for each additional input (32x4 for Calendar example)
 - can't exploit don't cares
- ❑ PLA approach advantageous when
 - design tools are available for multi-output minimization
 - there are relatively few unique minterm combinations
 - many minterms are shared among the output functions
 - Supports multilevel implementation using feedback
- ❑ PAL problems
 - constrained fan-ins on OR plane
 - Difficulty of common term re-use??

4. 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. As only AND gates are programmable, the PAL device is easier to program but it is not as flexible as the PLA. 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.

The device shown in the below figure has 4 inputs and 4 outputs. Each input has a buffer-inverter gate, and each output is generated by a fixed OR gate. The device has 4 sections, each composed of a 3-wide AND-OR array, meaning that there are 3 programmable AND gates in each section.

Each AND gate has 10 programmable input connections indicating by 10 vertical lines intersecting each horizontal line. The horizontal line symbolizes the multiple input configuration of an AND gate. One of the outputs F_1 is connected to a buffer-inverter gate and is fed back into the inputs of the AND gates through programmed connections.



Designing using a PAL device, the Boolean functions must be simplified to fit into each section. The number of product terms in each section is fixed and if the number of terms in the function is too large, it may be necessary to use two or more sections to implement one Boolean function.

Example:

Implement the following Boolean functions using the PAL device as shown above,

$$W(A, B, C, D) = \Sigma m(2, 12, 13)$$

$$X(A, B, C, D) = \Sigma m(7, 8, 9, 10, 11, 12, 13, 14, 15)$$

$$Y(A, B, C, D) = \Sigma m(0, 2, 3, 4, 5, 6, 7, 8, 10, 11, 15)$$

$$Z(A, B, C, D) = \Sigma m(1, 2, 8, 12, 13)$$

Simplifying the 4 functions to a minimum number of terms results in the following Boolean functions:

$$W = ABC' + A'B'CD'$$

$$X = A + BCD$$

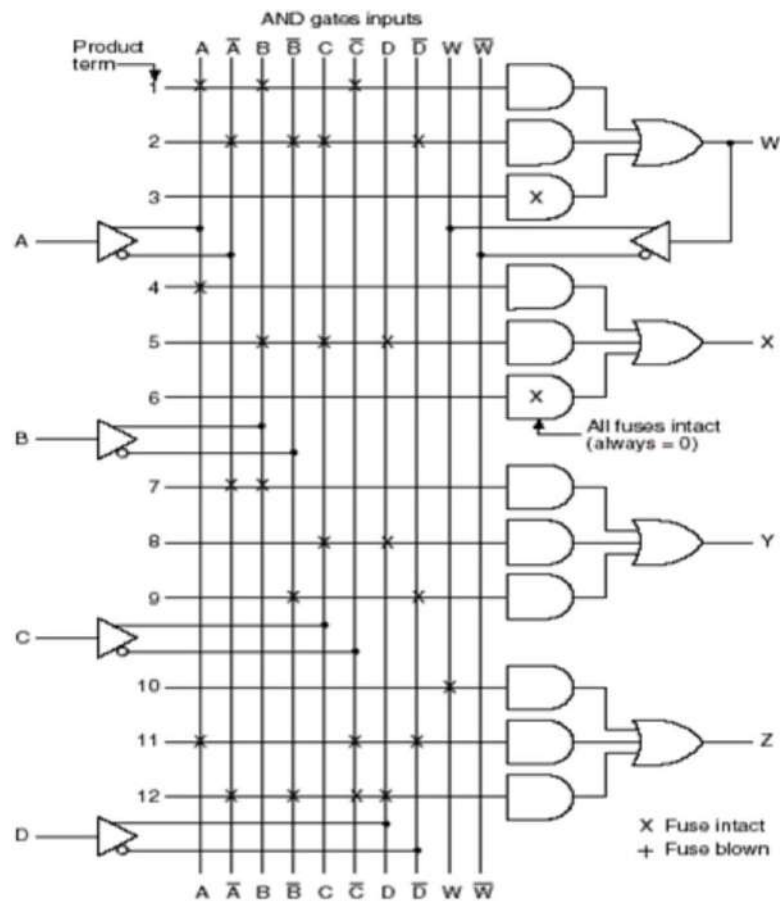
$$Y = A'B + CD + B'D'$$

$$Z = ABC' + A'B'CD + AC'D' + A'B'C'D = W + AC'D' + A'B'C'D$$

Note that the function for **Z** has four product terms. The logical sum of two of these terms is equal to **W**. Thus, by using **W**, it is possible to reduce the number of terms for **Z** from four to three, so that the function can fit into the given PAL device.

Product term	AND Inputs					Outputs
	A	B	C	D	W	
1	1	1	0	—	—	$W = \overline{A}\overline{B}\overline{C}\overline{D}$ $+ \overline{A}\overline{B}CD$
2	0	0	1	0	—	
3	—	—	—	—	—	
4	1	—	—	—	—	$X = A$ $+ BCD$
5	—	1	1	1	—	
6	—	—	—	—	—	
7	0	1	—	—	—	$Y = \overline{A}B$ $+ \overline{C}D$ $+ \overline{B}\overline{D}$
8	—	—	1	1	—	
9	—	0	—	0	—	
10	—	—	—	—	1	$Z = W$ $+ \overline{A}\overline{C}\overline{D}$ $+ \overline{A}\overline{B}\overline{C}\overline{D}$
11	1	—	0	0	—	
12	0	0	0	1	—	

The PAL programming table is similar to the table used for the PLA, except that only the inputs of the AND gates need to be programmed. The following figure shows the connection map for the PAL device, as specified in the programming table.



Since both W and X have two product terms, third AND gate is not used. If all the inputs to this AND gate left intact, then its output will always be 0, because it receives both the true and complement of each input variable i.e., $AA' = 0$

Inferences:

If an I/O pin's output-control gate produces a constant 1, the output is always enabled, but the pin may still be used as an input too.

Outputs can be used to generate first-pass “*helper terms*” for logic functions that cannot be performed in a single pass with the limited number of AND terms available for a single output.

Comparison: PAL Vs PLA

- ❑ PALs have the same limitations as PLAs (small number of allowed AND terms) plus they have a fixed OR plane i.e., less flexibility than PLAs
- ❑ PALs are simpler to manufacture, cheaper, and faster (better performance)
- ❑ PALs also often have extra circuitry connected to the output of each OR gate
 - The OR gate plus this circuitry is called a macro-cell

Comparison of ROM, PAL and PLA

- ❑ ROM – full AND plane, general OR plane
 - cheap (high-volume component)
 - can implement any function of n inputs
 - medium speed
- ❑ PAL – programmable AND plane, fixed OR plane
 - intermediate cost
 - can implement functions limited by number of terms
 - high speed (only one programmable plane that is much smaller than ROM's decoder)
- ❑ PLA – programmable AND and OR planes
 - most expensive (most complex in design, need more sophisticated tools)
 - can implement any function up to a product term limit
 - slow (two programmable planes)

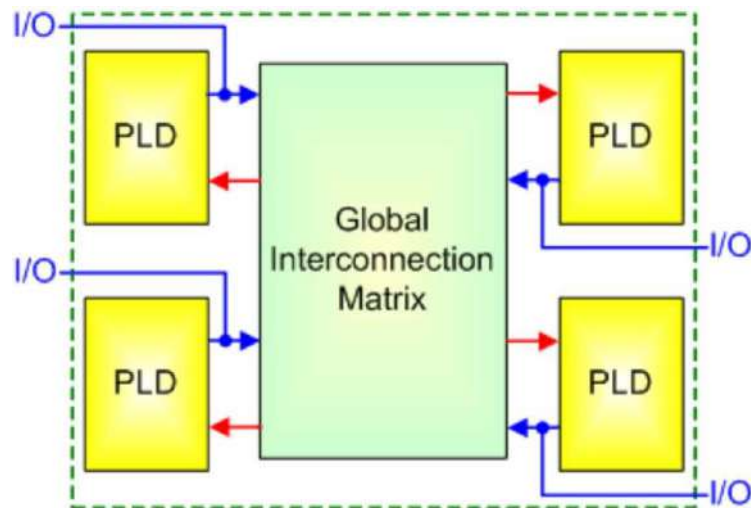
3. COMPLEX PROGRAMMABLE LOGIC DEVICES (CPLDs):

A CPLD contains a bunch of PLD blocks whose inputs and outputs are connected together by a global interconnection matrix. A CPLD is an arrangement of many SPLD-like blocks on a single chip. These circuit blocks might be either PAL-like or PLA-like blocks. Thus a CPLD has two levels of programmability: each PLD block can be programmed, and then the interconnections between the PLDs can be programmed.

Characteristics:

- They have a higher input to logic gate ratio.
- These devices are denser than SPLDs but have better functional abilities.
- CPLDs are based on EPROM or EEPROM technology.
- If you require a larger number of macrocells for a given application, ranging anywhere between 32 to 1000 macrocells, then a Complex Programmable Logic Device is the solution.

- Thus, we use CPLD in applications involving larger I/Os, but data processing is relatively low.



CASE STUDY: Xilinx XC9500 CPLD Family:

The Xilinx XC9500 series is a family of CPLDs with a similar architecture but varying numbers of external input/output (I/O) pins and internal PLDs which is called as function blocks (FBs). Each internal PLD has 36 inputs and 18 macrocells according to the number of chip family. Macro-cells whose outputs are usable only internally are sometimes called buried macrocells.

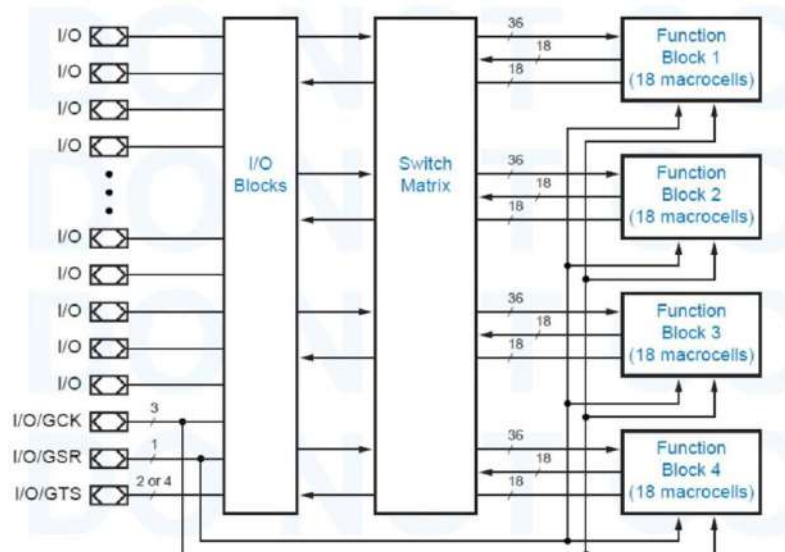


Figure shows the block diagram of the internal architecture of a typical XC9500-family CPLD. The external I/O pins can be used as input, output or bi-directional pins according to device programming. The pins marked I/O/GCK, I/O/GSR and I/O/GTS are special purpose

pins. Any of these pins can be used as global clocks (GCK). The same pin can be used as an asynchronous preset or clear. Two or four pins can be used as global three state controls (GTS).

Function Block Architecture

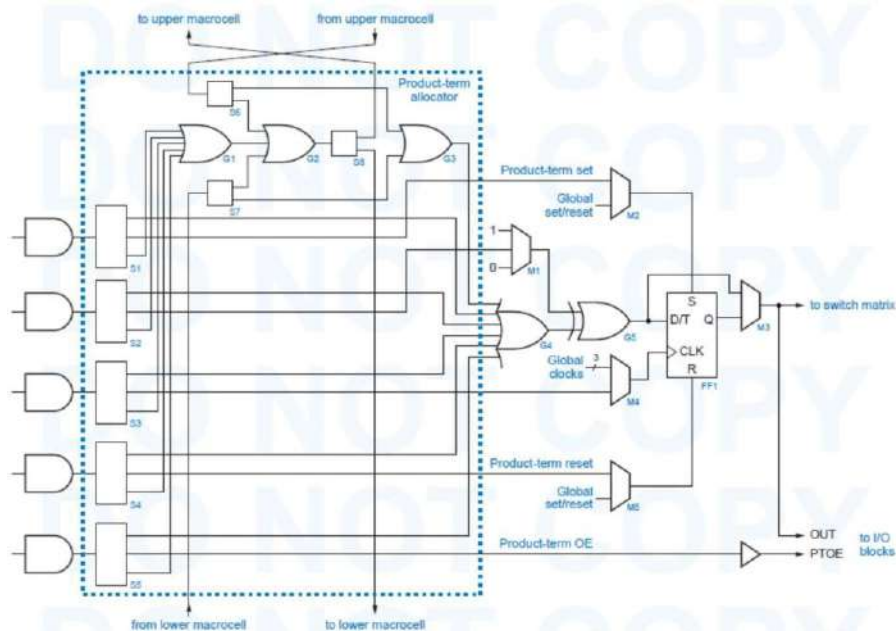
The basic structure of an XC9500 FB is shown in following figure. The programmable AND array has just 90 product terms. It has fewer AND terms per microcell. Each FB will receive 36 signals from the switch matrix, the macrocell outputs from each of the FB and the external inputs from the I/O pins are applied to the switching matrix. Each FB has 18 outputs which run “under” the switch matrix and connect to the I/O blocks. These signals are only the output enable signals for the I/O block output drivers



Macrocell:

The XC9500 and other CPLDs have product-term allocators that allow a macrocell’s unused product terms to be used by other nearby macrocells in the same FB. Figure shows the logic diagram of the XC9500 product-term allocator and macrocell. In this figure, the rectangular boxes labelled S1–S8 are programmable signal-steering elements that connect their input to one of their two or three outputs. The trapezoidal boxes labeled M1–M5 are programmable multiplexers that connect one of their two to four inputs to their output. The five AND gates associated with the macrocell appear on the left-hand side of the figure. Each one is connected to a signal-steering box whose top output connects the product term to the macrocell’s main OR gate G4. Considering just this, only five product terms are available per macrocell. However, the top, sixth input of G4 connects to another OR gate G3 that receives product terms from the macrocells above and below the current one. Any of the macrocell’s product terms that are not otherwise used can be steered through S1–S5 to be combined in an OR gate G1 whose output can eventually be steered to the macrocell above or below by S8. Before steering, product-term allocator these product terms may be combined with product terms from below or above through S6, S7, and G2. Thus, product terms can be “daisy-chained” through successive macrocells to create larger sums of products. In principle, all 90 product

terms in the FB could be combined and steered to one macrocell, although that would leave 17 out of the FB's 18 macrocells with no product terms at all.



Switch Matrix:

The Fast CONNECT switch matrix connects signals to the FB inputs. All IOB outputs (corresponding to user pin inputs) and all FB outputs drive the Fast CONNECT matrix. Any of these (up to a FB fan-in limit of 36) may be selected, through user programming, to drive each FB with a uniform delay. The switch matrix is capable of combining multiple internal connections into a single wired-AND output before driving the destination FB. This provides additional logic capability and increases the effective logic fan-in of the destination FB without any additional timing delay.

I/O Block:

The I/O Block (IOB) interfaces between the internal logic and the device user I/O pins. Each IOB includes an input buffer, output driver, output enable selection multiplexer, and user programmable ground control. The input buffer is compatible with standard 5V CMOS, 5V TTL, and 3.3V signal levels. The input buffer uses the internal 5V voltage supply (VCCINT) to ensure that the input thresholds are constant and do not vary with the VCCIO voltage.

The output enable may be generated from one of four options: a product term signal from the macrocell, any of the global OE signals, always [1], or always [0]. There are two global output enables for devices with up to 144 macrocells, and four global output enables for

the rest of the devices. Both polarities of any of the global 3-state control (GTS) pins may be used within the device.

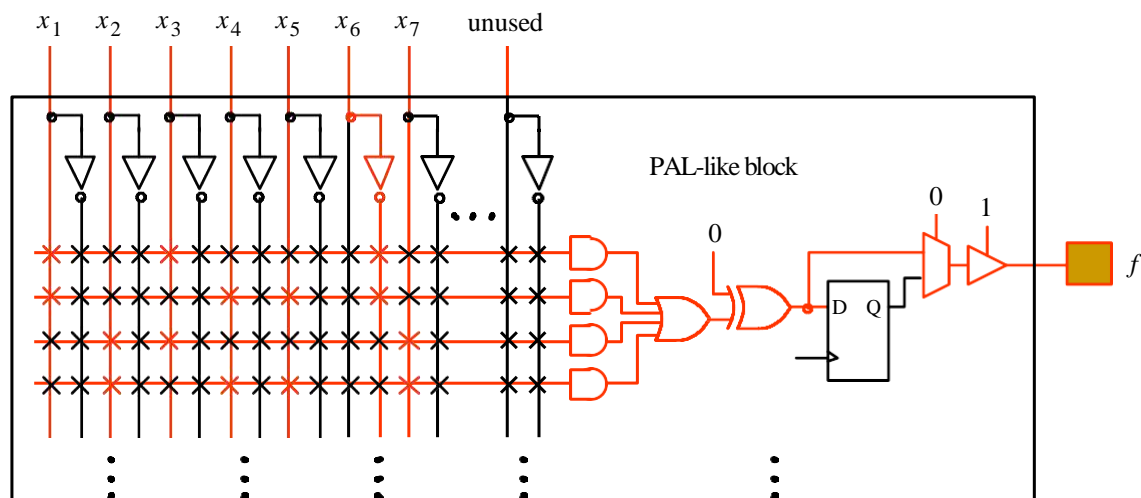
Features:

- High-performance
- Large density range: 36 to 288 macrocells with 800 to 6,400 usable gates
- 5V in-system programmable
- Endurance of 10,000 program/erase cycles
- Program/erase over full commercial voltage and temperature range
- Enhanced pin-locking architecture
- Flexible 36V18 Function Block
- 90 product terms drive any or all of 18 macrocells within Function Block
- Global and product term clocks, output enables, set and reset signals
- Extensive IEEE Std 1149.1 boundary-scan (JTAG) support
- Programmable power reduction mode in each macrocell
- Slew rate control on individual outputs - User programmable ground pin capability
- Extended pattern security features for design protection
- High-drive 24 mA outputs
- 3.3V or 5V I/O capability

Example:

Use a CPLD to implement the function, $f = x_1x_3x_6' + x_1x_4x_5x_6' + x_2x_3x_7 + x_2x_4x_5x_7$

(from interconnection wires)



Applications of CPLD:

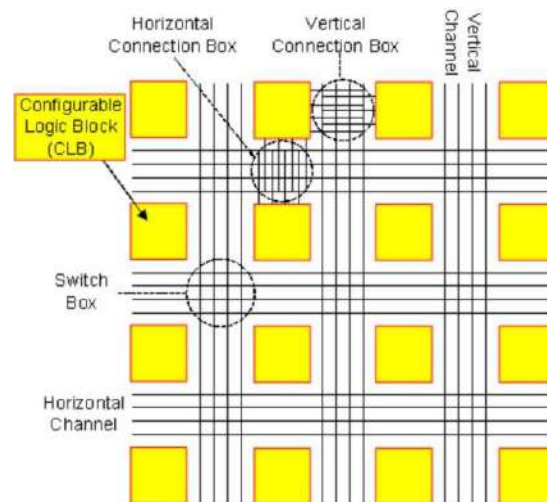
- Complex programmable logic devices are ideal for high performance, critical control applications.
- CPLD can be used in digital designs to perform the functions of boot loader
- CPLD is used for loading the configuration data of a field programmable gate array from non-volatile memory.
- Generally, these are used in small design applications like address decoding
- CPLDs are frequently used many applications like in cost sensitive, battery operated portable devices due to its low size and usage of low power.

4. FIELD PROGRAMMABLE GATE ARRAYS (FPGAs):

A Field Programmable Gate Array has an entire logic system integrated on a single chip. It offers excellent flexibility for reprogramming to the system designers. Logic circuitry involving more than a thousand gates use FPGAs. Compared to a normal custom system chip, the FPGA has ten times better integration density.

The FPGA consists of 3 main structures:

1. Programmable logic structure,
2. Programmable routing structure, and
3. Programmable Input/Output (I/O).



Programmable Logic Structure:

The programmable logic structure FPGA consists of a 2-dimensional array of configurable logic blocks (CLBs). These logic blocks have a lookup table in which the sequential circuitry is implemented. Each CLB can be configured (programmed) to implement

any Boolean function of its input variables. Typically CLBs have between 4-6 input variables.

Functions of larger number of variables are implemented using more than one CLB. In addition, each CLB typically contains 1 or 2 FFs to allow implementation of sequential logic.

Large designs are partitioned and mapped to a number of CLBs with each CLB configured (programmed) to perform a particular function. These CLBs are then connected together to fully implement the target design. Connecting the CLBs is done using the FPGA programmable routing structure.

Configurable Logic Blocks (CLBs):

Look-up Table (LUT)-Based CLB :

The basic unit of look-up table based FPGAs is the configurable logic block. The configurable logic block implements the logic functions. The look-up table based FPGA is the Xilinx 4000-series FPGA. Further, configurable logic block implements functions.

PLA-Based CLB :

PLA-based FPGA devices are based on conventional PLDs. The important advantage of this structure is the logic circuits are implemented using only a few level logic. To improve integration density logic expander is used.

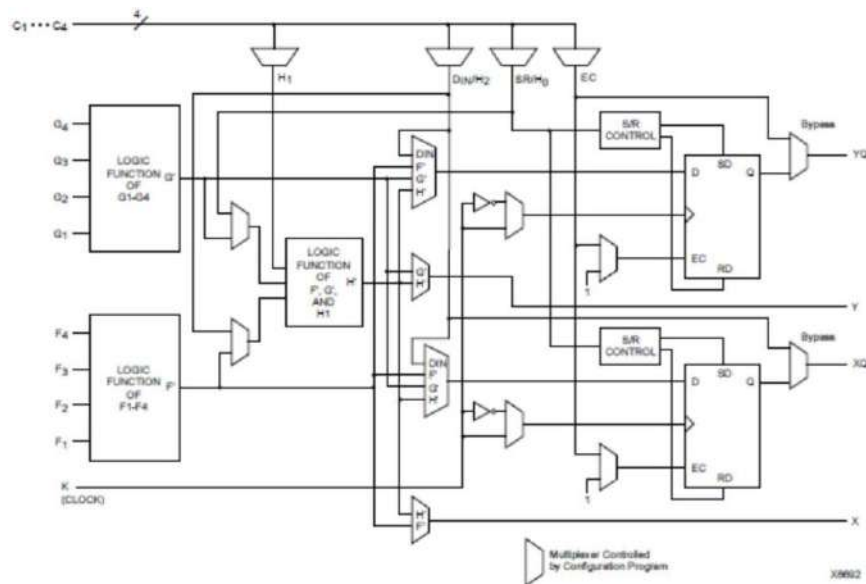
Multiplexer-Based CLB :

In Multiplexer-based FPGAs to implement the logic circuits the multiplexers are used. The main advantage of multiplexer-based FPGA is to provide more functionality by using minimum transistors. Due to large number of inputs, multiplexer-based FPGAs place high demands on routing.

CASE STUDY: Xilinx 4000 FPGA Family:

The principle CLB elements are shown in following figure. Each CLB contains a pair of flip-flops and two independent 4-input function generators. These function generators have a good deal of flexibility as most combinatorial logic functions need less than four inputs. Thirteen CLB inputs and four CLB outputs provide access to the functional flip-flops. Configurable Logic Blocks implement most of the logic in an FPGA. Two 4-input function generators (F and G) offer unrestricted versatility. Most combinatorial logic functions need four or fewer inputs. However, a third function generator (H) is provided. The H function generator has three inputs. One or both of these inputs can be the outputs of F and G; the other

input(s) are from outside the CLB. The CLB can therefore implement certain functions of up to nine variables, like parity check or expandable-identity comparison of two sets of four inputs.



Each CLB contains two flip-flops that can be used to store the function generator outputs. However, the flip-flops and function generators can also be used independently. DIN can be used as a direct input to either of the two flip-flops. H1 can drive the other flip-flop through the H function generator. Function generator outputs can also be accessed from outside the CLB, using two outputs independent of the flip-flop outputs. This versatility increases logic density and simplifies routing. Thirteen CLB inputs and four CLB outputs provide access to the function generators and flip-flops. These inputs and outputs connect to the programmable interconnect resources outside the block.

Four independent inputs are provided to each of two function generators (F1 - F4 and G1 - G4). These function generators, whose outputs are labelled F' and G', are each capable of implementing any arbitrarily defined Boolean function of four inputs. The function generators are implemented as memory look-up tables. The propagation delay is therefore independent of the function implemented. A third function generator, labelled H', can implement any Boolean function of its three inputs. Two of these inputs can optionally be the F' and G' functional generator out-puts. Alternatively, one or both of these inputs can come from outside the CLB (H2, H0). The third input must come from outside the block (H1).

Signals from the function generators can exit the CLB on two outputs. F' or H' can be connected to the X output. G' or H' can be connected to the Y output. A CLB can be used to implement any of the following functions:

- any function of up to four variables, plus any second function of up to four unrelated variables, plus any third function of up to three unrelated variables
- any single function of five variables
- any function of four variables together with some functions of six variables
- some functions of up to nine variables

Implementing wide functions in a single block reduces both the number of blocks required and the delay in the signal path, achieving both increased density and speed. The versatility of the CLB function generators significantly improves system speed. In addition, the design-software tools can deal with each function generator independently. This flexibility improves cell usage.

The flexibility and symmetry of the CLB architecture facilitates the placement and routing of a given application. Since the function generators and flip-flops have independent inputs and outputs, each can be treated as a separate entity during placement to achieve high packing density. Inputs, outputs and the functions themselves can freely swap positions within the CLB to avoid routing congestion during the placement and routing operation.

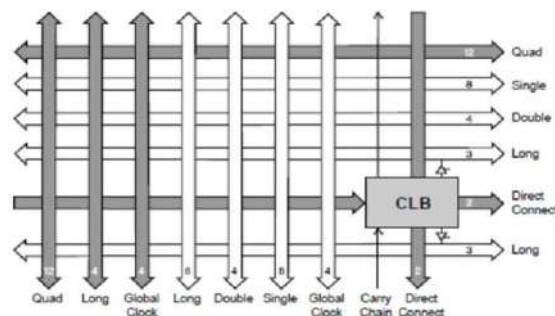
Programmable Routing Structure:

To allow for flexible interconnection of CLBs, FPGAs have 3 programmable routing resources:

1. Vertical and horizontal routing channels which consist of different length wires that can be connected together if needed. These channels run vertically and horizontally between columns and rows of CLBs as shown in the Figure.

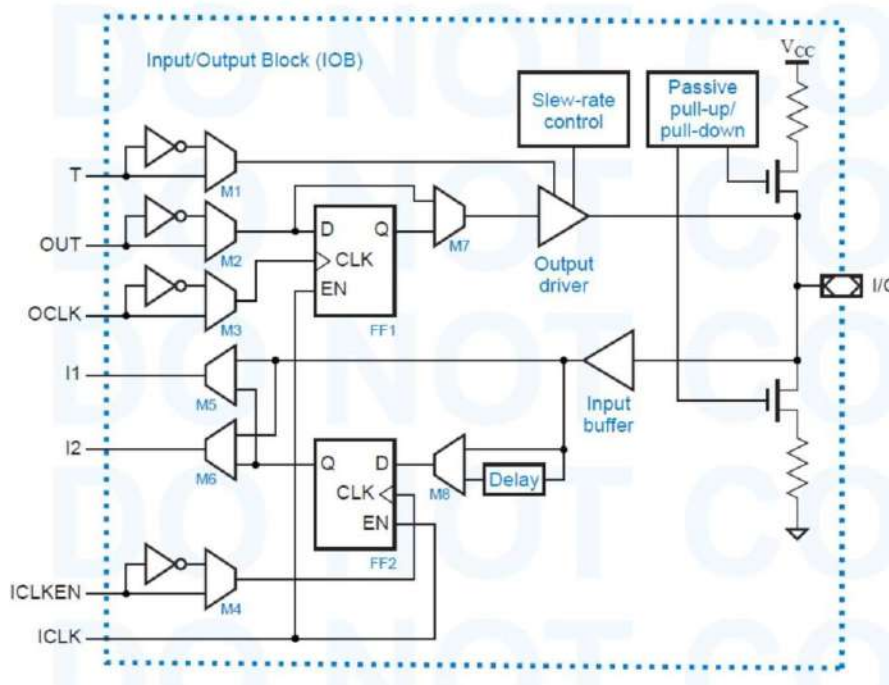
2. Connection boxes, which are a set of programmable links that can connect input and output pins of the CLBs to wires of the vertical or the horizontal routing channels.

3. Switch boxes, located at the intersection of the vertical and horizontal channels. These are a set of programmable links that can connect wire segments in the horizontal and vertical channels.



Programmable I/O:

These are mainly buffers that can be configured either as input buffers, output buffers or input/output buffers. They allow the pins of the FPGA chip to function either as input pins, output pins or input/output pins. The IOBs provide a simple interface between the internal user logic and the package pins.



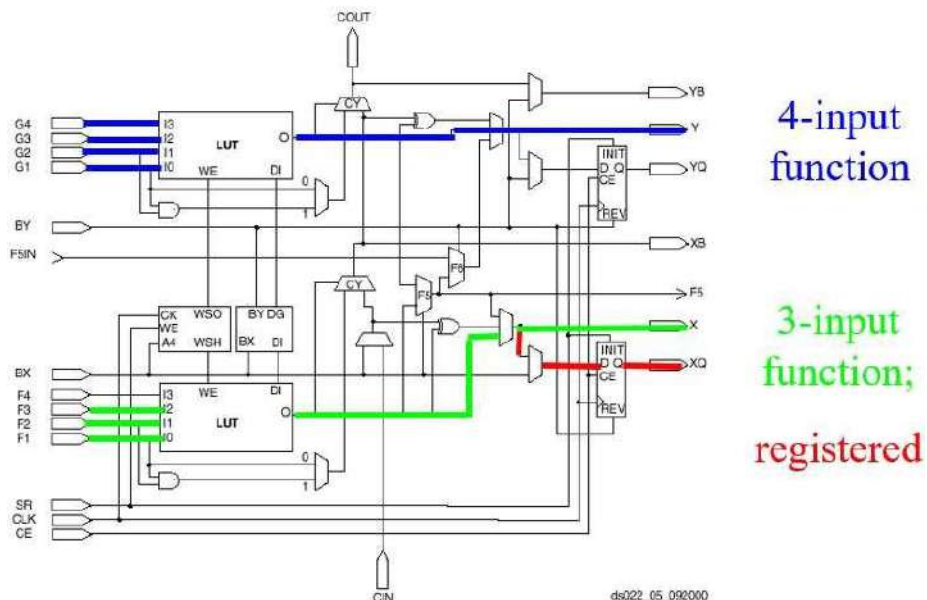
Input Signals:

Two paths, labelled I1 and I2, bring input signals into the array. Inputs also connect to an input register that can be programmed as either an edge-triggered flip-flop or a level-sensitive transparent-Low latch. The choice is made by placing the appropriate primitive from the symbol library. The inputs can be globally configured for either TTL (1.2V) or CMOS (2.5V) thresholds.

The two global adjustments of input threshold and output level are independent of each other. There is a slight hysteresis of about 300mV. Separate clock signals are provided for the input and output registers; these clocks can be inverted, generating either falling-edge or rising-edge triggered flip-flops. As is the case with the CLB registers, a global set/reset signal can be used to set or clear the input and output registers whenever the RESET net is alive.

Registered Inputs:

The I1 and I2 signals that exit the block can each carry either the direct or registered input signal. The input and output storage elements in each IOB have a common clock enable input, which through configuration can be activated individually for the input or output flip-



Applications of FPGAs:

- Implementation of random logic
 - easier changes at system-level (one device is modified)
 - can eliminate need for full-custom chips
- Prototyping
 - ensemble of gate arrays used to emulate a circuit to be manufactured
 - get more/better/faster debugging done than possible with simulation
- Reconfigurable hardware
 - one hardware block used to implement more than one function
 - functions must be mutually-exclusive in time
 - can greatly reduce cost while enhancing flexibility
 - RAM-based only option
- Special-purpose computation engines
 - hardware dedicated to solving one problem (or class of problems)
 - accelerators attached to general-purpose computers