Digital Electronic

References:

1- Digital design , fourth edition by Morris Mano

Basic Number Systems:

The feature which distinguishes one system from another is the number of symbols(digits) which are used , and this is called the base (radix) of the system.
Table 1 shows some of number systems

Base	name of number system	digits used in system
2	Binary	0,1
8	Octal	0,1,2,3,4,5,6,7
10	Decimal	0,1,2,3,4,5,6,7,8,9
16	Hexa decimal	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

In general, quantities are represented as:

$N= a_{-1}\, r^{-1} + a_{-2}\, r^{-2} + \ldots.+ a_0\, r^{0} + a_1\, r^{1} + a_2\, r^{2} + \ldots.+ a_n\, r^{n}$

Where each coefficient a , can take any value of the number system digits

And r is the base of the number system .

Number representation:

- Binary Number: the decimal number can be represent in binary by arranging the 1 and 0 under weight of the binary system to get the decimal number. Each digit in binary number called a ***Bit***. The bit in the right is called Least Significant Bit(LSB) ,and the bit in the left is called Most Significant Bit(MSB)

EX:

		32	16	8	4	2	1
	…..	2^5	2^4	2^3	2^2	2^1	2^0
Decimal							
4		0	0	0	1	0	0
12		0	0	1	1	0	0
22		0	1	0	1	1	0

- Octal Number: the decimal number can be present in Octal by arranging basic digits according to the octal system to get the decimal number.

EX:

	512	64	8	1
	…. 8^3	8^2	8^1	8^0
Decimal				
4	0	0	0	4
54	0	0	6	6
90	0	1	3	2

- Hexadecimal: the decimal number can be present in hexadecimal by arranging basic digits according to the weight of the hexadecimal system to get the decimal number.

EX:

	4096	256	16	1
	…. 16^3	16^2	16^1	16^0
Decimal				
4	0	0	0	4
25	0	0	1	9
45	0	0	2	D
284	0	1	1	C

Number Conversions:

In this section , the conversion between these number systems will be studied.

1- Binary to Decimal conversion:

By weighted sum method

EX:

· $(110)_2$ $(?)_{10}$

$$1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 6$$

The answer is $(6)_{10}$

· $(1011.1100)_2$ $(?)_{10}$

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \ . \ 1 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} = 11.75$$

The answer is $(11.75)_{10}$

2-Decimal to Binary conversions:

For integer part By division on the base (2) and take the remainder in each stage.
And by multiplication with(2) for mantesa.

EX:

·

$(50)_{10}$ $(?)_2$

2	50	0	LSB
2	25	1	
2	12	0	
2	6	0	
2	3	1	
2	1	1	MSB

The answer is $(110010)_2$

· $(0.6875)_{10}$ $(?)_2$

$0.6875 \times 2 = 1.375$ 1

$0.375 \times 2 = 0.75$ 0

$0.75 \times 2 = 1.5$ 1

$0.5 \times 2 = 1.0$ 1

The answer is $(0.1011)_2$

3- Binary to Octal conversions: each three bits from right to left represent a number.

—

- $(111010011)_2 \longrightarrow (723)_8$
- $(110100001)_2 \longrightarrow (641)_8$

4- Octal to binary conversions: each number is representing in binary using three bits.

- $(752)_8 \longrightarrow (111101010)_2$
- $(631)_8 \longrightarrow (110011001)_2$

5- Binary to Hexadecimal conversions: each four bits from right to left represent a number.

- $(1101110001010001)_2 \longrightarrow (DC51)_{16}$

6- Hexadecimal to Binary conversions: number is representing in binary using four bits.

EX:

- $(A35C)_{16} \longrightarrow (1010001101011100)_2$

7- Octal to Decimal conversions:

- $(371)_8 \longrightarrow (?)_{10}$

$1 \times 8^0 + 7 \times 8^1 + 3 \times 8^2 = 249$

The answer is $(249)_{10}$

8- Decimal to Octal conversions:

- $(30.5)_{10} \longrightarrow (?)_8$

8	30	6	LSB	$0.5 \times 8 = 4.0$	
8	3	3	MSB		

The answer is $(36.4)_8$

9-Hexadecimal to decimal conversions:

EX:

- $(1BF)_{16} \longrightarrow (?)_{10}$

$F \times 16^0 + B \times 16^1 + 1 \times 16^3 = 447$

The answer is $(447)_{10}$

10- Decimal to Hexadecimal conversions:

EX:

- $(28.3)_{10} \longrightarrow (?)_{16}$

16	28	12	LSB	$0.3 \times 16 = 4.8$	4
16	1	1	MSB	$0.8 \times 16 = 12.8$	12
				$0.8 \times 16 = 12.8$	12

The answer is $(1C.4CC)_{16}$

Table 2 shows the first 16 decimal number and there conversion in (Binary , Hexadecimal, and Octal) number systems.

Decimal	Binary	Hexadecimal	Octal
0	0	0	0
1	1	1	1
2	10	2	2
3	11	3	3
4	100	4	4
5	101	5	5
6	110	6	6
7	111	7	7
8	1000	8	10
9	1001	9	11
10	1010	A	12
11	1011	B	13
12	1100	C	14
13	1101	D	15
14	1110	E	16
15	1111	F	17

Complement:

Complement is used in digital computer to simplify the subtraction operation and for logical manipulation.

There are two types of complement for each base (r)

1. The r's complement.
2. The (r-1)'s complement.

The Binary numbers Complement:

1- One's (first) Complement:

1's complement= $r^n - N-1$

where n : number of bits

N: binary number

r : system base

Simply the first complement of binary number is the number we get by changing each bit (0 to 1)and (1 to 0).

Ex: the first complement of

a) 101100 is 010011

b) 1000000 is 0111111

2- The Two's (second) Complement:

The equation is:

2's complement = $r^n - N$

Simply the 2's complement is equal to 1's complement added by one.

EX: The 2's complement of :

a) 101101 is 010011

b) 0111101 is 1000011

Binary Arithmetic

1- Addition:-

0 + 0 = 0
0 + 1 = 1
1 + 0 = 1
1 + 1 = 0 carry 1

EX: add the two binary numbers (001) and (100)

```
  0 0 1
+ 1 0 0
-------
  1 0 1
```

EX: add the two binary numbers (111) and (001)

```
  1 1 1
+ 0 0 1
-------
1 0 0 0
```

2- Subtraction:-

0 - 0 = 0
0 - 1 = 1 borrow 1
1 - 0 = 1
1 - 1 = 0

EX: subtract the binary number (100) from (101)

```
  1 0 1
- 1 0 0
-------
  0 0 1
```

EX: subtract the binary number (1110) from (1111)

```
  1 1 1 1
- 1 1 1 0
---------
  0 0 0 1
```

Subtraction Using 2's complement:

Take the 2's complement to the subtracted N and then add it to M, if an end carry occur, discard it. If an end carry does not occur, take 2's complement of the number obtained in step 1.

EX: subtract the number (1010100) from (1000100) using 2's complement.

M 1000100
N - 1010100 ⟶ 1000100
+ 0101100
1110000

No end carry ⟶ □

The answer is - 10000

EX: subtract the number (1011101) from (1110101) using 2's complement.

M 1110101
N - 1011101 ⟶ 1110101
+ 0100011
1 0011000

There is an end carry ⟶
The answer is 11000

Subtraction Using 1's Complement:

Add M to 1's complement of N (subtracted) and check the carry:
If an end carry occur, add 1 to the least significant bit. And if an end carry does not occur, take the 1's complement of the number obtained in step 1 and place a negative sign in front.

EX: subtract the number (1010100) from (1000100) using 1's complement.

M 1000100
N - 1010100 ⟶

1000100
\+ 0101011
1101111

No end carry ⟶ ☐

The answer is - 10000

EX: subtract the number (1011101) from (1110101) using 1's complement.

M 1110101
N - 1011101 ⟶

1110101
\+ 0100010
1 0010111
\+ 0000001
11000

There is an end carry will ⟶ be neglected

The answer is 11000

*3- **Multiplication:***

0 * 0 = 0
0 * 1 = 0
1 * 0 = 0
1 * 1 = 1

EX: Multiply the two numbers (111) and (101).

```
   111
*  101
   111
  0000
 11100
100011
```

*4- **Division***

Binary division is again similar to its decimal counterpart:

EX: divide the number (11011) on (101)

```
          1 0 1
        ---------
1 0 1  ) 1 1 0 1 1
       − 1 0 1
         -----
         0 0 1 1 1
       −     1 0 1
             -----
             1 0
```

The result is (101) and the reminder is (10)

Binary Codes:

Three types of code will be considered:

1. BCD (Binary code decimal).
2. Excess – 3 code.
3. Gray code.

Binary Codes for Decimal Digits

- Internally, digital computers operate on binary numbers.
- When interfacing to humans, digital processors, e.g. pocket calculators, communication is decimal-based.
- Input is done in decimal then converted to binary for internal processing.
- For output, the result has to be converted from its internal binary representation to a decimal form.
- To be handled by digital processors, the decimal input (output) must be coded in binary in a digit by digit manner.
- For example, to input the decimal number **957**, each *digit* of the number is individually *coded* and the number is stored as **1001_0101_0111.**
- Thus, we need a specific code for each of the 10 decimal digits. There is a variety of such decimal binary codes.
- The shown table gives several common such codes.
- One commonly used code is the *Binary Coded Decimal* (**BCD**) code which corresponds to the first 10 binary representations of the decimal digits 0-9.
- The BCD code requires 4 bits to represent the 10 decimal digits.
- Since 4 bits may have up to 16 different binary combinations, a total of 6 combinations will be unused.
- The position weights of the BCD code are 8, 4, 2, 1.

- Other codes (shown in the table) use position weights of 8, 4, -2, -1 and 2, 4, 2, 1.
- An example of a non-weighted code is the *excess-3 code* where digit codes are obtained from their binary equivalent after adding 3. Thus the code of a decimal 0 is 0011, that of 6 is 1001, etc.

Decimal Digit	BCD 8	4	2	1	8	4	-2	-1	2	4	2	1	Excess-3			
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
1	0	0	0	1	0	1	1	1	0	0	0	1	0	1	0	0
2	0	0	1	0	0	1	1	0	0	0	1	0	0	1	0	1
3	0	0	1	1	0	1	0	1	0	0	1	1	0	1	1	0
4	0	1	0	0	0	1	0	0	0	1	0	0	0	1	1	1
5	0	1	0	1	1	0	1	1	1	0	1	1	1	0	0	0
6	0	1	1	0	1	0	1	0	1	1	0	0	1	0	0	1
7	0	1	1	1	1	0	0	1	1	1	0	1	1	0	1	0
8	1	0	0	0	1	0	0	0	1	1	1	0	1	0	1	1
9	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0

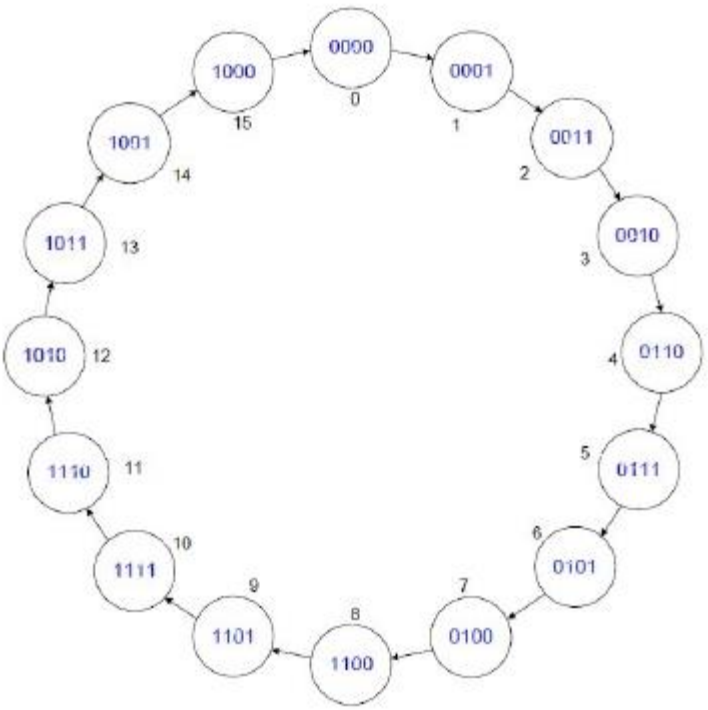
Example Converting **(13)10** into binary, we get **1101**, coding the same number into BCD, we obtain **00010011**.

Exercise: Convert **(95)10** into its binary equivalent value and give its BCD code as well.

Answer {**(1011111)2**, and **and BCD is 10010101**}

Gray Code

- The Gray code consists of 16 4-bit code words to represent the decimal Numbers 0 to 15.
- For Gray code, successive code words differ by only one bit from one to the next as shown in the table and further illustrated in the Figure.



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

Binary Number to Gray Code Conversion:

The procedure of conversion from binary to gray code is :
1- put down the MSB
2- start from the MSB adding with out carry each two adjacent bits

EX: Convert from Binary to Gray code

110101110 Binary

101111001 Gray

EX: Convert from Binary to Gray code

1100 Binary

1010 Gray

Gray Code to Binary Number Conversion:

The procedure of conversion from gray code to binary is :

1- put down the MSB

2- start from the MSB adding with out carry each result binary bit with the lower gray code bit

EX: Convert from Gray code to Binary

101110101 Gray

110100110 Binary

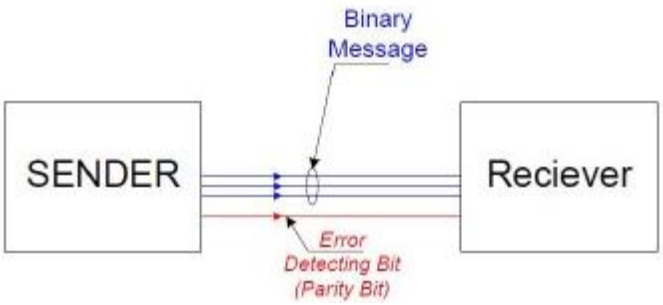
EX: Convert from Gray code to Binary

1110010001 Gray

10111000010 Binary

Error-Detection Codes

- Binary information may be transmitted through some communication medium, e.g. using wires or wireless media.
- A corrupted bit will have its value changed from 0 to 1 or vice versa.
- To be able to detect errors at the receiver end, the sender sends an extra bit (parity bit) with the original binary message.



- A *parity bit* is an extra bit included with the *n-bit binary message* to make the total number of 1's in this message (*including the parity bit*) either odd or even.
- If the parity bit makes the total number of 1's an *odd* (*even*) number, it is called *odd* (*even*) parity.
- The table shows the required *odd* (*even*) parity for a 3-bit message.

Three-Bit Message			Odd Parity Bit	Even Parity Bit
X	Y	Z	P	P
0	0	0	1	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	0	1

- At the receiver end, an error is detected if the message does not match have the proper parity (odd/even).
- Parity bits can detect the occurrence 1, 3, 5 or any odd number of errors in the transmitted message.
- No error is detectable if the transmitted message has 2 bits in error since the total number of 1's will remain even (or odd) as in the original message.
- In general, a transmitted message with even number of errors cannot be detected by the parity bit.

Logic Gates

Introduction:

The logic gate is the basic building block in digital systems. Logic gates operate with binary numbers. Gates are therefore referred to as binary logic gates. All voltages used with logic gates will be either HIGH or LOW. In this lecture, a HIGH voltage will mean a binary 1. A LOW voltage will mean a binary 0. Remember that logic gates are electronic circuits. These circuits will respond only to HIGH voltages (called 1s) or LOW (ground) voltages (called 0s).

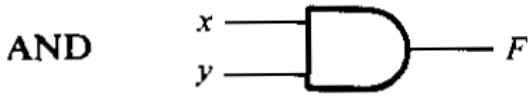
All digital systems are constructed by using only three basic logic gates. These basic gates are called the AND gate, the OR gate, and the NOT gate. This chapter deals with these very important basic logic gates, or functions.

1- The AND gate:

The AND gate is called the "all or nothing" gate. The standard logic symbol for the AND gate is drawn in Fig. (1.a). This symbol shows the inputs as x and y. The output is shown as F. This is the symbol for a 2-input AND gate. The truth table for the 2-input AND gate is shown in Fig. 5.2b. The inputs are shown as binary digits (bits). Note that only when both input x and input B are 1 will the output be 1. Binary 0 is defined as a LOW, or ground, voltage. Binary 1 is defined as a HIGH voltage. In this book, a HIGH voltage will mean about +5 volts (V) if the integrated circuits (ICs) being used are from the TTL family.

Boolean algebra is a form of symbolic logic that shows how logic gates operate. A Boolean expression is a "shorthand" method of showing what is happening in a logic circuit.

The AND gate symbol, Boolean expression and truth table are shown in fig (1).



$F = xy$

x	y	F
0	0	0
0	1	0
1	0	0
1	1	1

a) Symbol b) Boolean expression c) Truth table

Fig(1) The AND gate symbol , Boolean expression and truth table

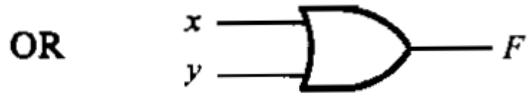
The Boolean expression reads x AND y equals the output F.
The laws of Boolean algebra govern how AND gates operate. The formal laws for the AND functions are:

x . 1 = x
x . 0 = 0
x . x = x
x . x' = 0

2- The OR gate:

The OR gate is called the “any or all” gate the standard logic symbol for an OR gate is drawn in Fig (2). Note the different shape of the OR gate. The OR gate has two inputs labeled x and y. The output is labeled F. The shorthand Boolean expression for this OR function is given as x + y = F. Note that the plus (+) symbol means OR in Boolean algebra. The expression (x + y = F) is read as x OR (+ means OR) y equals output F. You will note that the plus sign does not mean to add as it does in regular algebra.

The OR gate symbol, Boolean expression and truth table are shown in fig(2).



$F = x + y$

x	y	F
0	0	0
0	1	1
1	0	1
1	1	1

a) Symbol b) Boolean expression c) Truth table

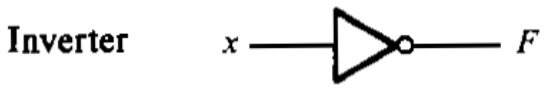
Fig (2) The OR gate symbol, Boolean expression and truth table

The Boolean expression reads x OR y equals the output F. The laws of Boolean algebra govern how OR gates operate. The formal laws for the OR function are:

x + 1 = 1
x + 0 = x
x + x = x
x + x' = 1

3- The NOT gate:

The NOT gate is also called an inverter. NOT gate, or inverter, is an unusual gate. The NOT gate has only one input and one output. Fig (3) illustrates the logic symbol for the NOT gate , Boolean expression and truth table.



$F = x'$

x	F
0	1
1	0

a) Symbol b) Boolean expression c) Truth table

Fig (3) The NOT gate symbol , Boolean expression and truth table.

The process of inverting is simple. Figure 5.5b is the truth table for the NOT gate. The input is always changed to its opposite. If the input is 0, the NOT gate will give its complement, or opposite, which is 1. If the input to the NOT gate is a 1, the circuit will complement it to give a 0. This inverting is also called complementing or negating. The terms negating, complementing, and inverting all mean the same thing.

If the input x is inverted to x' (not x). The x' is then inverted again to form x (not not x). The double inverted x is equal to the original x.

The laws of Boolean algebra govern how NOT gates operate. The formal laws for the NOT function are:

0' = 1 1' = 0
If x = 0 then x' = 1
If x = 1 then x' = 0
x'' = x

The four other logic gates which implemented from the above basic gates are:-

a) The NAND gate :

This is implemented from the AND gate with NOT gate, so that it is the **complement of the AND** gate.

The NAND gate symbol, Boolean expression and truth table are shown in fig (4).

x	y	F
0	0	1
0	1	1
1	0	1
1	1	0

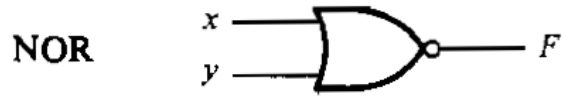
a) Symbol b) Boolean expression c) Truth table

Fig (4) The NAND gate symbol, Boolean expression and truth table.

b) The NOR gate :

This is implemented from the OR gate with NOT gate, so that it is the complement of the OR gate.

The NOR gate symbol, Boolean expression and truth table are shown in fig (5).



$F = (x + y)'$

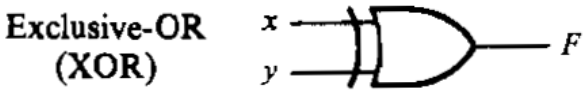
x	y	F
0	0	1
0	1	0
1	0	0
1	1	0

a) Symbol b) Boolean expression c) Truth table

Fig (5) The NOR gate symbol , Boolean expression and truth table

c) The Exclusive OR (XOR) gate :

This is implemented from the (OR, NOT, AND) gates, as you can see its Boolean expression. The XOR gate symbol, Boolean expression and truth table are shown in fig (6).



$F = xy' + x'y$
$= x \oplus y$

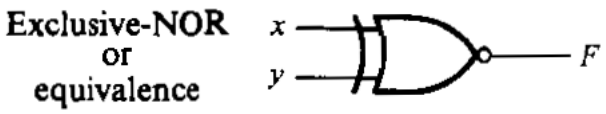
x	y	F
0	0	0
0	1	1
1	0	1
1	1	0

a) Symbol b) Boolean expression c) Truth table

Fig (6) The XOR gate symbol, Boolean expression and truth table.

d) The Exclusive NOR (XNOR) gate:

This is the **complement of the XOR** gate. The XNOR gate symbol, Boolean expression and truth table are shown in fig (7).



$F = xy + x'y'$
$= x \odot y$

x	y	F
0	0	1
0	1	0
1	0	0
1	1	1

a) Symbol b) Boolean expression c) Truth table

Fig(7) The XNOR gate symbol , Boolean expression and truth table>

The summery of all logic gates is shown in fig(8)

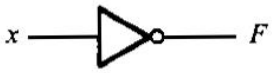
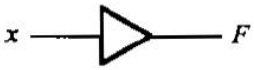
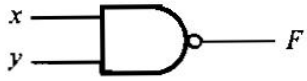
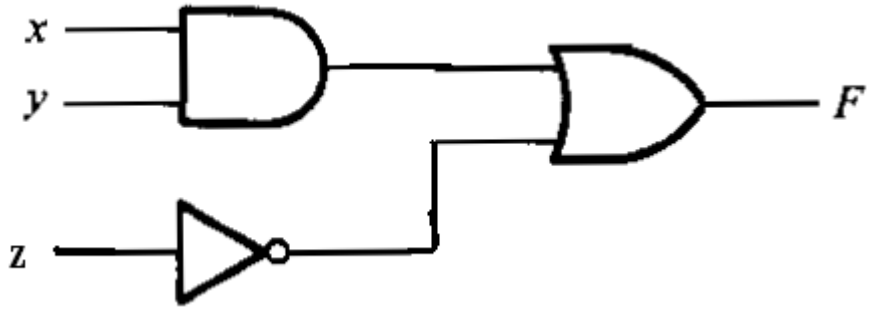
Name	Graphic symbol	Algebraic function	Truth table
AND		$F = xy$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$
OR		$F = x + y$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 1$
Inverter		$F = x'$	$x \mid F$ $0 \mid 1$ $1 \mid 0$
Buffer		$F = x$	$x \mid F$ $0 \mid 0$ $1 \mid 1$
NAND		$F = (xy)'$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
NOR		$F = (x + y)'$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 0$
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	$x\ y \mid F$ $0\ 0 \mid 0$ $0\ 1 \mid 1$ $1\ 0 \mid 1$ $1\ 1 \mid 0$
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= x \odot y$	$x\ y \mid F$ $0\ 0 \mid 1$ $0\ 1 \mid 0$ $1\ 0 \mid 0$ $1\ 1 \mid 1$

Fig (8) . The logic gates summery

Boolean algebra:

Boolean algebra is a form of symbolic logic that shows how logic gates operate. A Boolean expression is a “shorthand” method of showing what is happening in a logic circuit.

Example: Write the Boolean expression for the logic circuit shown.



The Boolean expression of this circuit is:

F = xy + z'

The basic rules of Boolean algebra are:-

Postulates and Theorems of Boolean Algebra

Postulate 2	(a) $x + 0 = x$	(b) $x \cdot 1 = x$
Postulate 5	(a) $x + x' = 1$	(b) $x \cdot x' = 0$
Theorem 1	(a) $x + x = x$	(b) $x \cdot x = x$
Theorem 2	(a) $x + 1 = 1$	(b) $x \cdot 0 = 0$
Theorem 3, involution	$(x')' = x$	
Postulate 3, commutative	(a) $x + y = y + x$	(b) $xy = yx$
Theorem 4, associative	(a) $x + (y + z) = (x + y) + z$	(b) $x(yz) = (xy)z$
Postulate 4, distributive	(a) $x(y + z) = xy + xz$	(b) $x + yz = (x + y)(x + z)$
Theorem 5, DeMorgan	(a) $(x + y)' = x'y'$	(b) $(xy)' = x' + y'$
Theorem 6, absorption	(a) $x + xy = x$	(b) $x(x + y) = x$

Simplification using Boolean algebra:

The above theorems of Boolean algebra can be used to simplify the logic expressions as in the following examples:

Example : Simplify the following Boolean expression.

F = xyz + x'y + xyz'

Sol:

F = xy(z + z') + x'y

F = xy + x'y

F = y(x + x')

F = y

Example : Simplify the following Boolean expression.

F = xy + x'z + yz

Sol :

F = xy + x'z + yz (x + x')

F = xy + x'z + yzx + yzx'

F = xy(1 + z) + x'z(1 + y)

F = xy + x'z

Example: Simplify the following Boolean expression.

F = ((x' + y). z')'

Sol: using De Morgan theorem

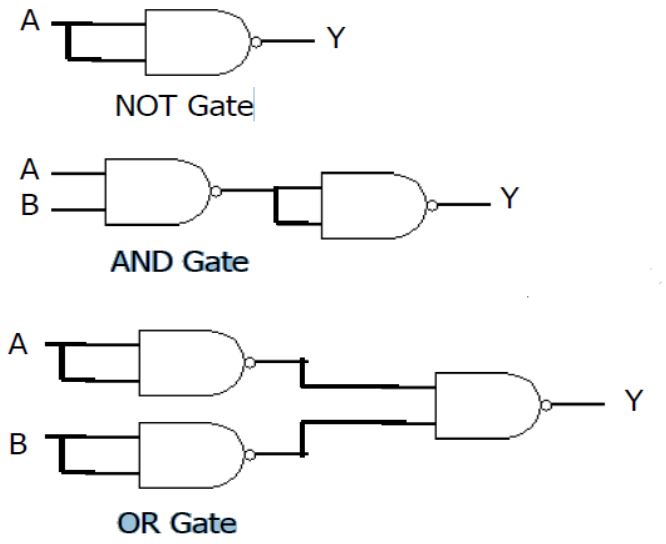
F = (x' + y)' + z''

F = x'' . y' + z

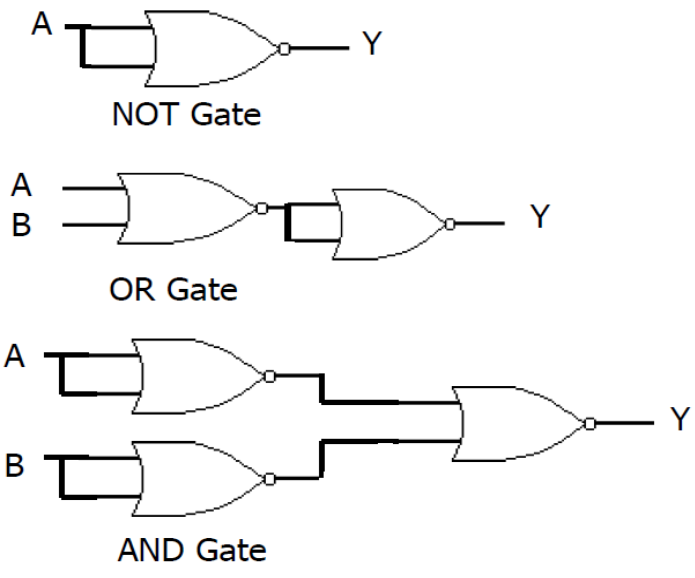
F = x. y' + z

Universality of NAND & NOR Gates

It is possible to implement any logic expression using only NAND gates and no other type of gate. This is because NAND gates, in the proper combination, can be used to perform each of the Boolean operations OR, AND, and NOT.



In a similar manner, it can be shown that NOR gates can be arranged to implement any of the Boolean operations.



Example : Implement the Boolean expression

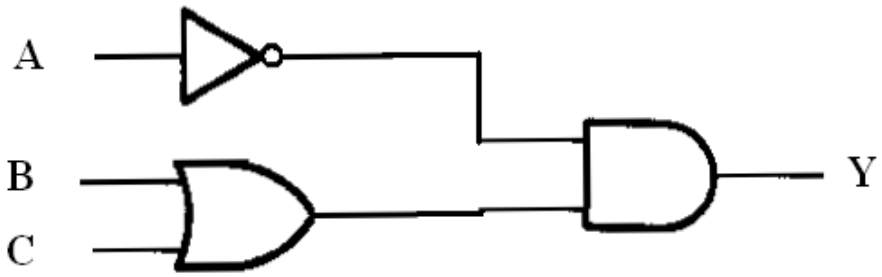
Y = A' . (B + C)

a) using basic logic gates.

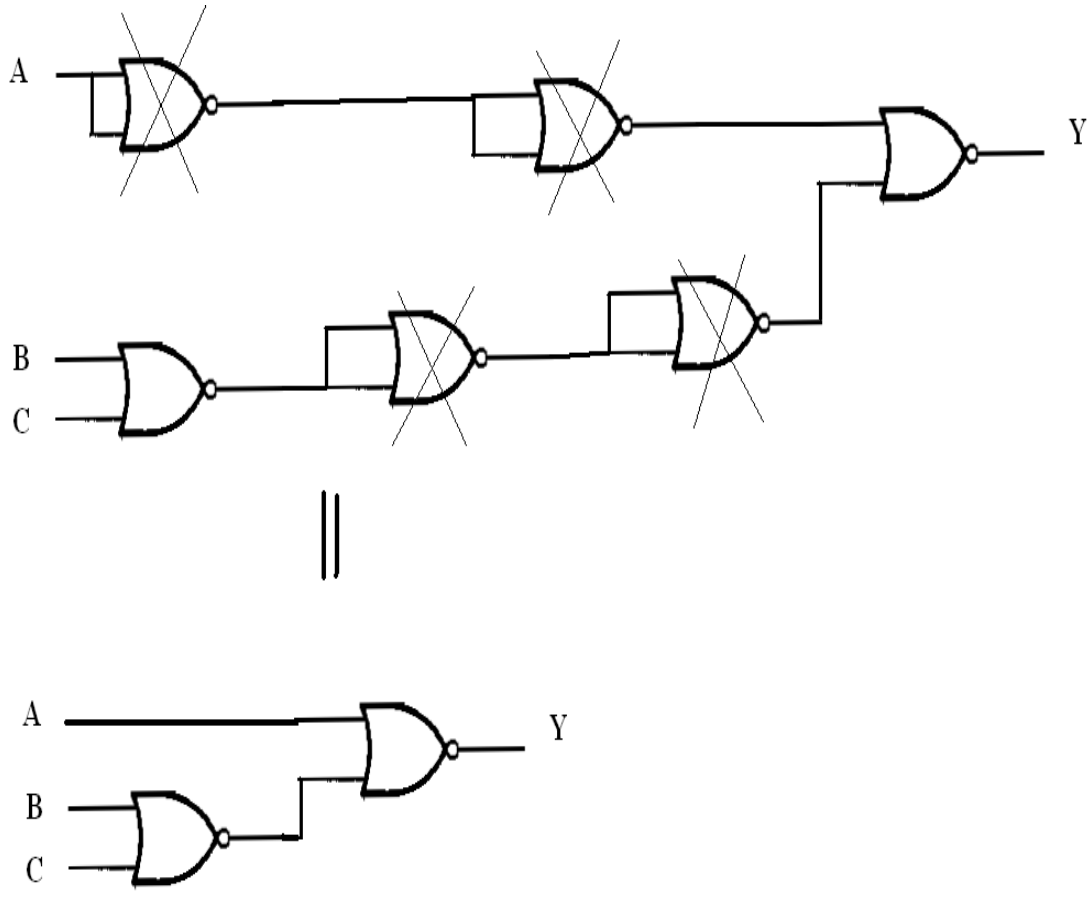
b) using NOR gates only.

Sol :

a)



b)



Logic Circuits

The logic circuits can be divided into two categories :

1- Combinational Logic Circuits.
2- Sequential Logic Circuits.

1- Combinational Logic Circuits:

It is the logic circuit implemented with logic gates only with out needing to synchronous signal.
The design of Combinational logic circuit depending on the derivation of the Boolean expression from the truth table. There are two methods for this derivation :-

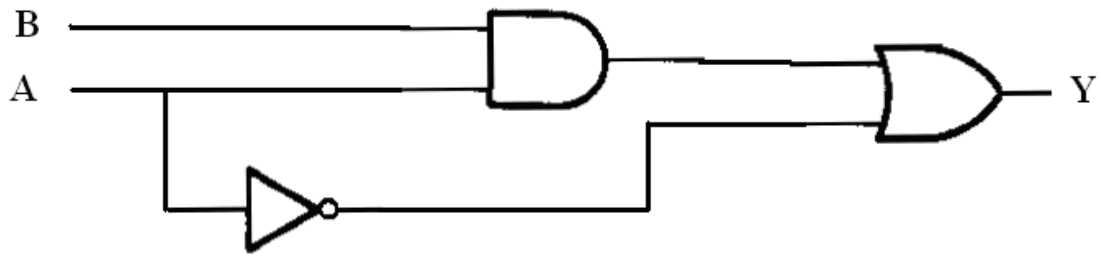
a) Sum of Products (SOP) method:- it is called Minterms expansion , given that the truth table , The minterm Boolean expression is developed from the output 1s in the truth table. Each (logic 1) in the output column becomes a term to be ORed in the minterm expression.

Example : Derive the Boolean expression from the following truth table and implement it.

B	**A**	**Y**		
0	0	1	-------	A'B'
0	1	0		
1	0	1	-------	A'B
1	1	1	-------	AB

Sol:

Y = A'B' + A'B + AB
= A'(B' + B) +AB
= A' + AB

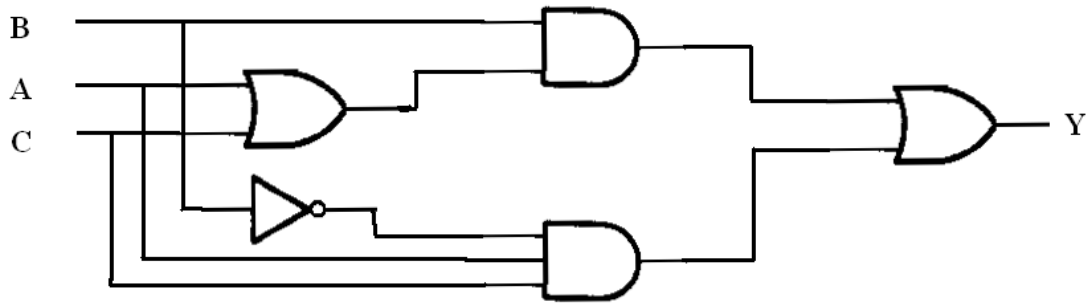


Example : Derive the Boolean expression from the following truth table and implement it.

C	B	A	Y	
0	0	0	0	
0	0	1	0	
0	1	0	0	
0	1	1	1	------- = ABC'
1	0	0	0	
1	0	1	1	------- = AB'C
1	1	0	1	------- = CBA'
1	1	1	1	------- = ABC

Sol:

Y = ABC' + AB'C + A'BC + ABC
= AB(C' + C) + AB'C + A'BC
= AB + AB'C + A'BC
= AB (C + 1) + AB'C + A'BC
= BCA + BA + AB'C + A'BC
= BC (A + A') + AB + AB'C
= B(A + C) + AB'C



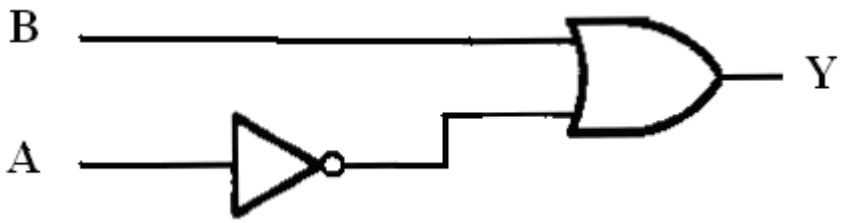
a) **Product of Sums (POS) method**:- it is called Maxterms expansion , given that the truth table , you can find the Boolean expression for the output by ANDing the fundamental sums that produce a (logic 0) output .

Example : Derive the Boolean expression from the following truth table and implement it.

B	**A**	**Y**	
0	0	1	
0	1	0	------- A' + B
1	0	1	
1	1	1	

Sol:

Y = A' + B



Example : Design a logic circuit that gives a (logic 1) output when the two inputs are identical using SOP and POS.

Sol:

B	**A**	**Y**	**SOP**	**POS**
0	0	1	------- A'B'	
0	1	0	----------	A' + B
1	0	0	----------	A + B'
1	1	1	------ AB	

a) design with SOP: Y = A'B' + AB
b) design with POS : Y = (A + B'). (A' + B)

Design Procedures using SOP are :-

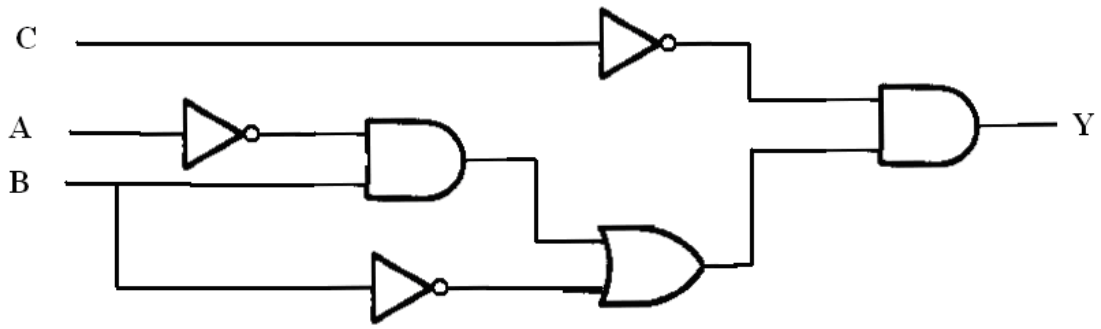
1- Write the truth table for all input states which equal to (2^n) where n is the number of inputs.
2- Write the expression for each (Logic 1) output with AND gate
3- Write the overall output expression by OR ing the terms in step 2 and if it is possible Simplify this expression.
4- Implement this expression using logic gates.

Example: Design a logic circuit that has 3 inputs and gives a (logic 1) output when the binary input value less than or equal 2.

Sol: number of inputs =3 ; number of states = $2^3 = 8$

C	B	A	Y		
0	0	0	1	--------	A'B'C'
0	0	1	1	--------	A B'C'
0	1	0	1	--------	A'BC'
0	1	1	0		
1	0	0	0		
1	0	1	0		
1	1	0	0		
1	1	1	0		

Y = A'B'C' + AB'C' + A'BC'
= B'C'(A' + A) + A'BC'
= B'C' + A'BC'
= C' (B' + BA')



Simplification using Karnuaph Map:

When constructing digital circuits, in addition to obtaining a functionally correct circuit, we like to optimize them in terms of circuit size, speed, and power consumption. In this section, we will focus on the reduction of circuit size. Usually, by reducing the circuit size, we will also improve on speed and power consumption. We saw how we can transform a Boolean function to another equivalent function by using the Boolean algebra theorems. If the resulting function is simpler than the original, then we want to implement the circuit based on the simpler function, since that will give us a smaller circuit size. Using Boolean algebra to transform a function to one that is simpler is not an easy task, especially for the computer. There is no formula that says which is the next theorem to use. Luckily, there are easier methods for reducing Boolean functions.

The *Karnaugh map* method.

The ***Karnaugh map*** method is an easy way for reducing an equation manually and is discussed in the following section.

This graphic method is based on **Boolean** theorems. It is only one of several methods used by logic designers to simplify logic circuits. Karnaugh maps are sometimes referred **to** as ***K- maps.***

❖ The first step in the Karnaugh mapping procedure is to develop a minterm Boolean expression from a truth table. Each 1 in the Y column of the truth table produces input variables ANDed together. These ANDed groups are then ORed to form a sum-of-products (minterm) type of Boolean expression . This expression will be referred to as the unsimplified Boolean expression.

The number of squares in the Map = 2^n

Where n is the number of inputs (variables)

❖ The second step in the mapping procedure is to plot 1s in the Karnaugh map . Each ANDed set of variables from the minterm expression is placed in the appropriate square of the map. The map is just a very special output column of the truth table.

❖ The third step is to loop adjacent groups of two, four, or eight 1s together.

❖ The fourth step is to eliminate variables.

❖ The fifth step is to OR the remaining variables.

a) Two inputs(variables) Karnauph map

The number of squares = 2^2 = 4

B	A	Y
0	0	m_0
0	1	m_1
1	0	m_2
1	1	m_3

A'	A
m_0	m_1
m_2	m_3

b) Three variables (inputs) K- Map:

The number of squares = 2^3 = 8

C	B	A	Y
0	0	0	m_0
0	0	1	m_1
0	1	0	m_2
0	1	1	m_3
1	0	0	m_4
1	0	1	m_5
1	1	0	m_6
1	1	1	m_7

	B'A'	B'A	BA	BA'
C'	m_0	m_1	m_3	m_2
C	m_4	m_5	m_7	m_6

c) Four variables (inputs) K-Map:

The number of squares = 2^4 = 16

D	C	B	A	Y
0	0	0	0	m_0
0	0	0	1	m_1
0	0	1	0	m_2
0	0	1	1	m_3
0	1	0	0	m_4
0	1	0	1	m_5
0	1	1	0	m_6
0	1	1	1	m_7
1	0	0	0	m_8
1	0	0	1	m_9
1	0	1	0	m_{10}
1	0	1	1	m_{11}
1	1	0	0	m_{12}
1	1	0	1	m_{13}
1	1	1	0	m_{14}
1	1	1	1	m_{15}

	B'A'	**B'A**	**BA**	**BA'**
D'C'	m_0	m_1	m_3	m_2
D'C	m_4	m_5	m_7	m_6
DC	m_{12}	m_{13}	m_{15}	m_{14}
DC'	m_8	m_9	m_{11}	m_{10}

Example : Derive the logic expression of this truth table and simplify it using K-map.

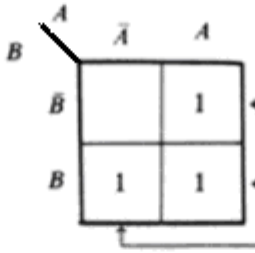
(*a*) **Truth Table**

Inputs		Output
B	*A*	*Y*
0	0	0
1	0	1
0	1	1
1	1	1

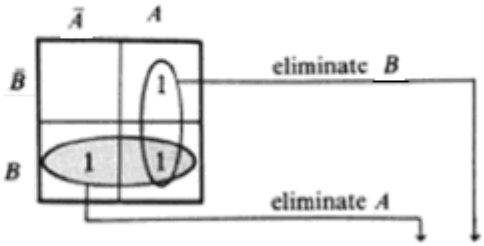
$\bar{A} \cdot B$
$A \cdot \bar{B}$
$A \cdot B$

(*b*) Minterm Boolean expression: $A \cdot B + A \cdot \bar{B} + \bar{A} \cdot B = Y$

(*c*) Plotting 1s on map



(*d*) Looping 1s



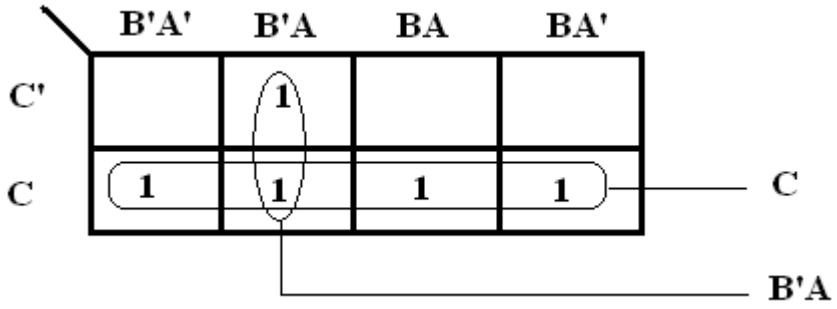
(*e*) Eliminating variables to form simplified Boolean expression: $B + A = Y$

Example : Simplify the logic expression using K- map.

Y = C'B'A + CB'A' + CB'A + CBA' + CBA

SOL:

C	B	A	Y
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



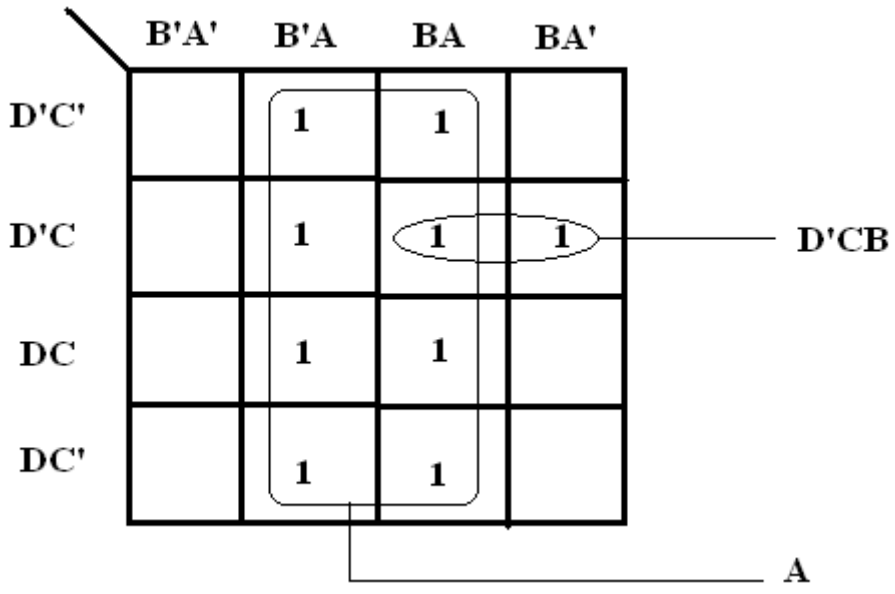
The simplified expression is:

Y = C + B'A

Example : Simplify the logic expression using K- map.
Y = D'C'B'A + D'C'BA + D'CB'A + D'CBA' + D'CBA + DC'B'A + DC'BA + DCB'A + DCBA

SOL:

D	C	B	A	Y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1



The simplified expression is:
Y = A + D'CB

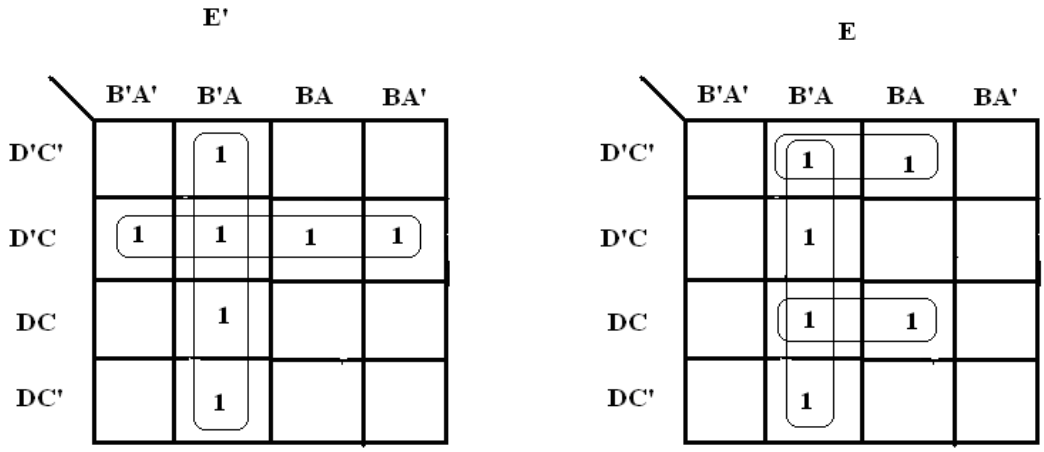
d) Five variables (inputs) K-Map:

The number of squares = 2^5 = 32

Because of the K- Map can not implement greater than four variable , the fifth variable get out and draw K-Map of four variable for the two states of the fifth variable as shown in the example below:

Example : simplify the following logic expression using K-Map

Y = E'D'C'B'A + E'D'CB'A +E'D'CB'A' + E' D'CBA + E'D'CBA' + E'DCB'A + E'DC'B'A + ED'C'B'A + ED'C'BA + ED'CB'A + EDCB'A + EDC'B'A + EDCBA



Y = E'D'C + B'A + ED'C'A + EDCA

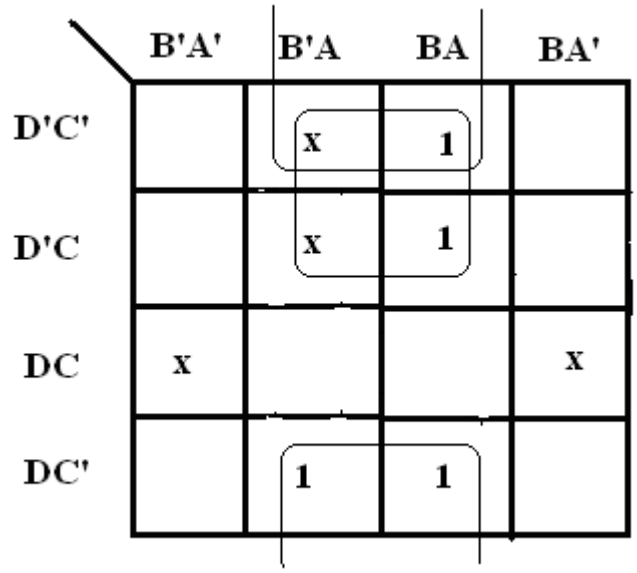
Note : by the same way we can simplify the logic expression with 6 variables or more .

K- Map with don't care:

In certain cases some of the minterms may never occur or it may not matter what happens if they do
– In such cases we fill in the Karnaugh map with an x
• meaning don't care
– When minimizing an x is like a "joker"
• x can be 0 or 1 - whatever helps best with the minimization
• "Don't care" conditions should be changed to either 0 or 1 to produce K-map looping that yields the simplest expression

Example : simplify the logic expression using K-map

Y = D'C'BA + D'CBA + DC'B'A +DC'BA
And
Y = D'C'B'A + D'CB'A + DCB'A' + DCBA' are don't care



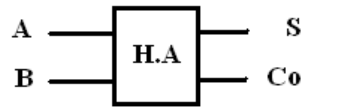
Y = D'A + C'A

Functions of Combinational Logic:

Adders :

In many computer logic applications it's necessary to add binary numbers the two types of adders are :-

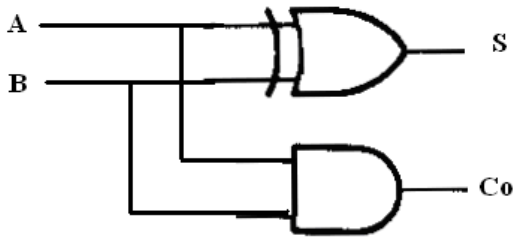
a) Half adder (H. A) :
A half adder is a multiple outputs combinational logic circuit which add two bits of binary data **with out carry**, producing a sum (S) and a carry out (Co).



Symbol of half adder (H.A)

B	A	S	Co
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Truth Table of half adder (H.A)



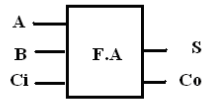
Circuit diagram of half adder (H.A)

S = B'.A + B.A'
Co = A.B

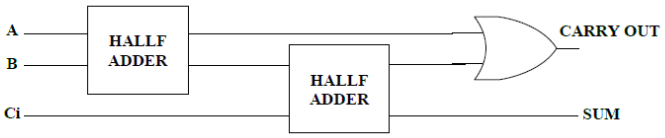
$$S = A \oplus B$$
$$Co = A\,B$$

b) Full adder (F.A) :

A full adder is a multiple outputs combinational logic circuit which add two bits of binary data **with carry input (Ci)**, producing a sum (S) and a carry out (Co).

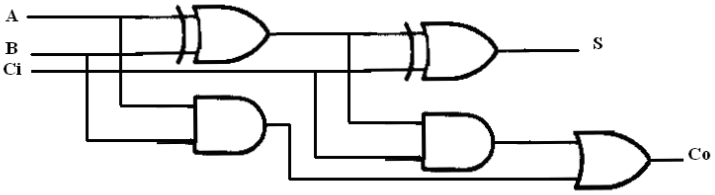


Symbol of full adder (F.A)



Ci	B	A	S	Co
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

Truth Table of full adder (F.A)



Circuit diagram of full adder (F.A)

S = A'B'Ci + A'BCi' + AB'Ci' + ABCi

Co = A'BCi + ABCi' + AB'Ci + ABCi

$$S = A \oplus B \oplus Ci$$

$$Co = AB + (A \oplus B)\,Ci$$

Parallel adder :

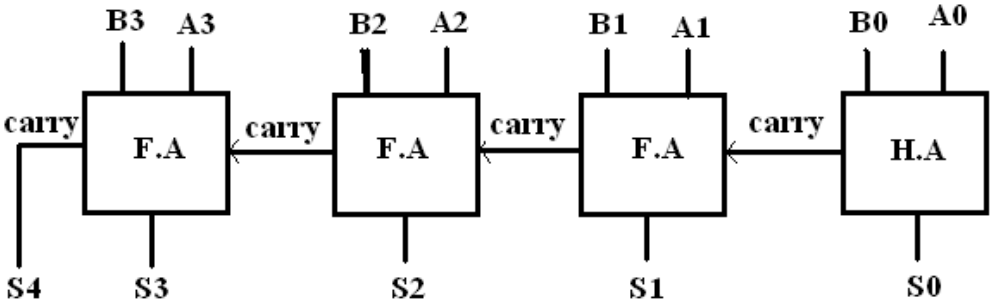
How can we add two binary numbers of 4 bits

N A3 A2 A1 A0

M B3 B2 B1 B0 +

S S4 S3 S2 S1 S0

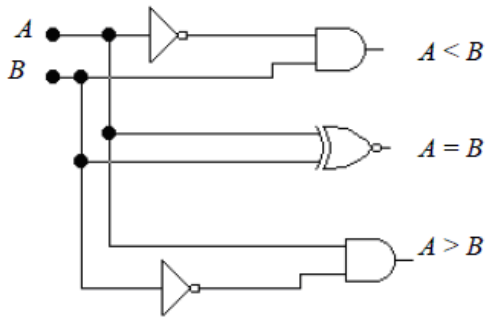
We need 3 full adder and 1 half adder



Digital comparator

Digital Comparator "also called Magnitude Comparator" is a combinational circuit that compares two inputs binary quantities (A and B) and generates outputs to indicate whether the inputs are equal or which input is greater than the other, therefore, the circuit has three outputs to indicate whether A=B, A>B or A<B. At any given input quantities, only one output should be equal to logic '1'. Figure shows a circuit diagram of the magnitude comparator.

We will begin by discussing how to compare two single bit numbers. The truth table for the single bit comparator is as shown :



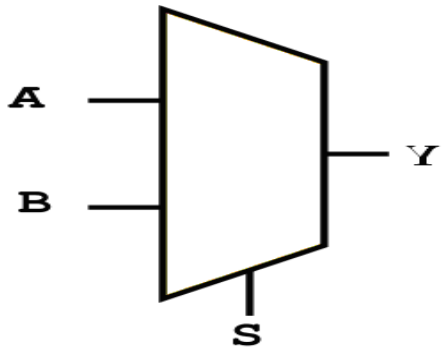
Truth Table

Input (I/P)		Output (O/P)		
A	B	$A > B$	$A = B$	$A < B$
0	0	0	1	0
1	0	1	0	0
0	1	0	0	1
1	1	0	1	0

(A < B) = A' B
(A = B) = A'B' + AB
(A > B) = AB'

Multiplexer

A multiplexer (mux) is a digital system that selects one out of possible 2n inputs depending on n select bit(s). For instance, the truth table and schematic symbol for a 2-to-1 mux are shown below.



symbol of a 2-to-1 mux

And the truth table of (2-to-1) mux is :

S	B	A	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

OR

S	Y
0	A
1	B

Examining the truth table closely we see that a logic value of 0 on the select bit (S) would connect A to the output while a logic value of 1 would connect B to the output. The Boolean expression for the output (Y) in terms of inputs A, B and S is:

Y = A.S' + B.S

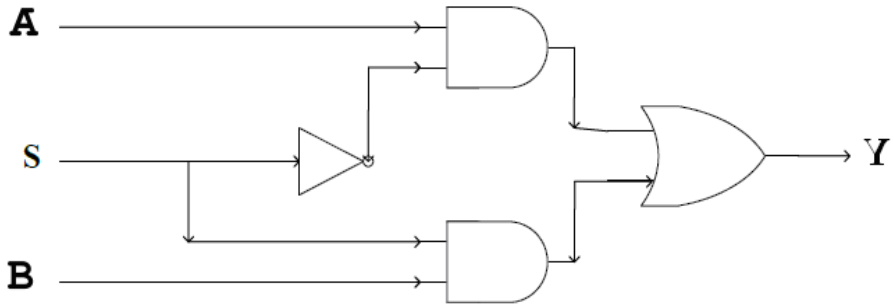


Figure . 2 - to – 1 multiplexer

Larger multiplexers are also common, if you have 4 inputs then you need 2 select bits. This is the reason for the n-select bits mapping 2n inputs to one output.

x	z	Y
0	0	A
0	1	B
1	0	C
1	1	D

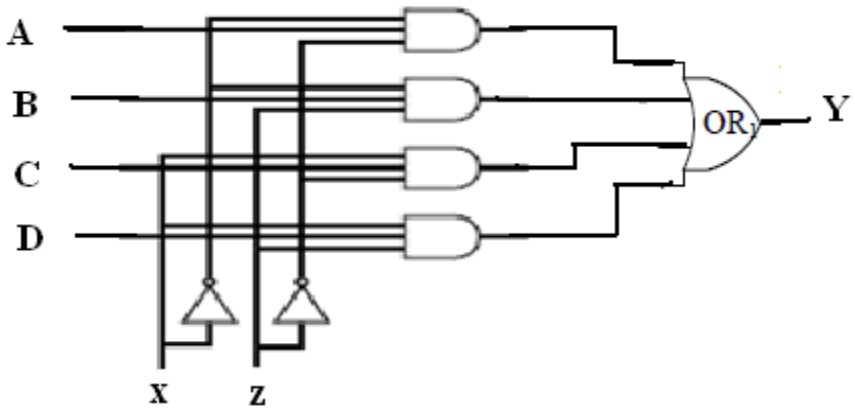


Figure . 4 - to – 1 multiplexer

Y = AX'Z' + BX'Z + CXZ' + DXZ

Demultiplexer

A demultiplexer basically reverses the multiplexing function. It is take data from one line and distribute them to given number of output lines. Figure below shown a one to four line demultiplexer circuit. The input data line goes to all of the AND gates. The two select lines enable only one gate at a time and the data appearing on the input line will pass through the selected gate to the associated output line.
The simplest type of demultiplexer is the 1- to- 2 lines DMUX. as shown in Figure below.

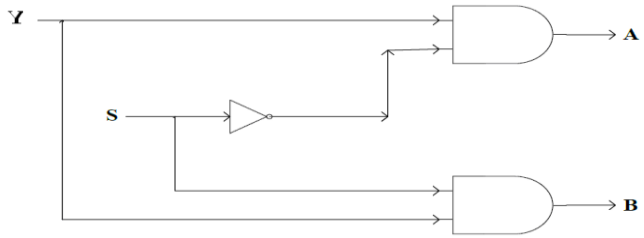


Figure 1- to – 2 Demultiplexer

S	A	B
0	Y	0
1	0	Y

Truth table of 1- to – 2 demultiplexer

A = YS'
B = YS

x	z	A	B	C	D
0	0	Y	0	0	0
0	1	0	Y	0	0
1	0	0	0	Y	0
1	1	0	0	0	Y

Truth table of 1- to – 4 demultiplexer

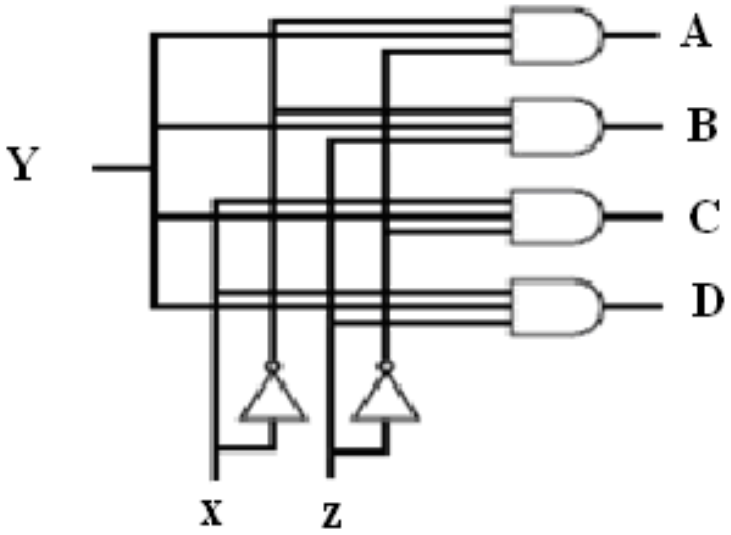
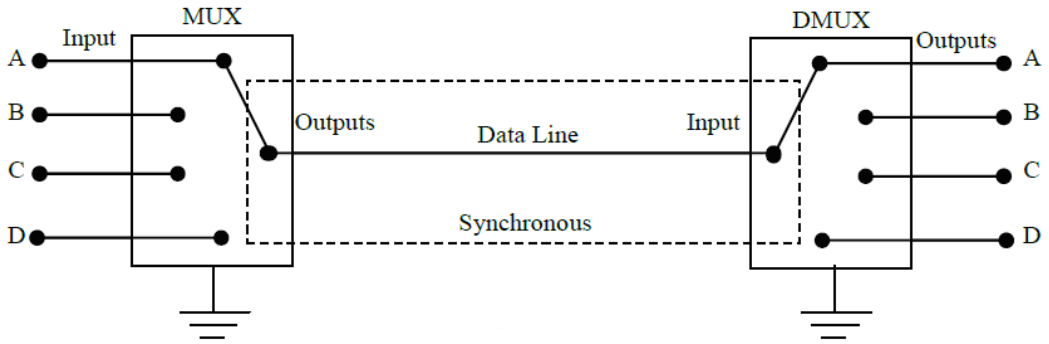


Figure : 1- to – 4 Demultiplexer

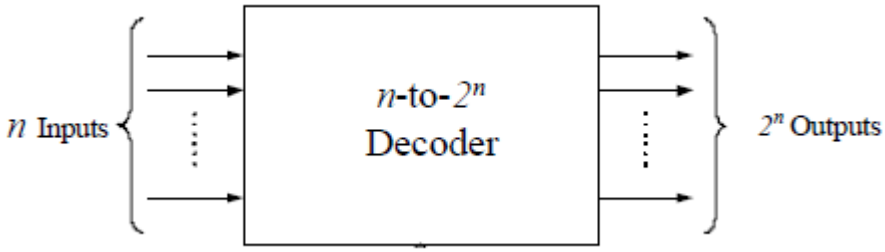
A = YX'Z'
B = YX'Z
C = YXZ'
D = YXZ

The commonly using of multiplexer and demultiplexer is the communication system where a data must be transmitted on a single line as shown in figure below



Decoders

As its name indicates, a decoder is a circuit component that decodes an input code. Given a binary code of n-bits, a decoder will tell which code is this out of the 2n possible codes (See Figure).
Thus, a decoder has n- inputs and 2^n outputs. Each of the 2^n outputs corresponds to one of the possible 2^n input combinations.



In general, output i equals 1 if and only if the input binary code has a value of i.

Example: 2-to-4 decoders

Let us discuss the operation and combinational circuit design of a decoder by taking the specific example of a 2-to-4 decoder. It contains two inputs denoted by A and B and four outputs denoted by D0, D1, D2, and D3 as shown in figure 2. Also note that A1 is the MSB while A0 is the LSB.

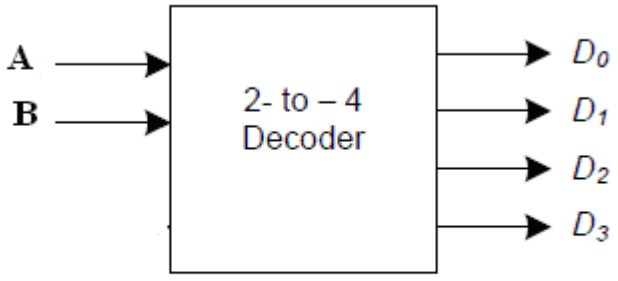


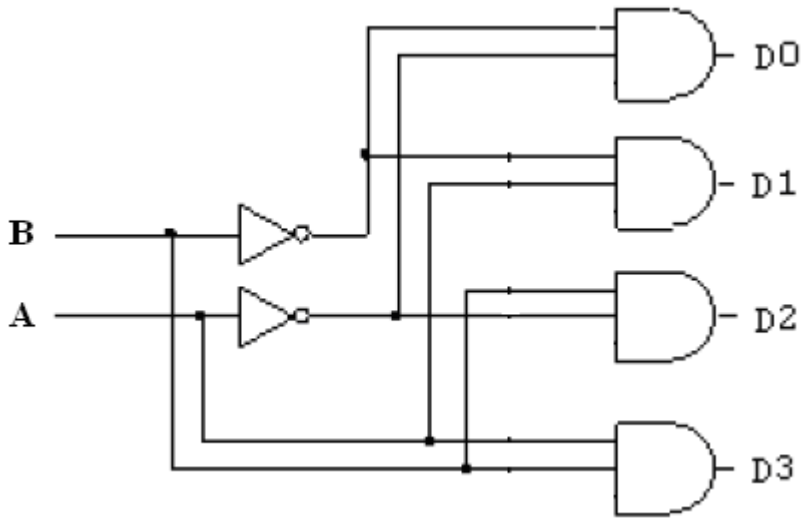
Figure Block Diagram of 2-to-4 Decoder

B	A	D_0	D_1	D_2	D_3
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Truth table of 2-to-4 Decoder

As we see in the truth table , for each input combination, one output line is activated, that is, the output line corresponding to the input combination becomes 1, while other lines remain inactive. For example, an input of 00 at the input will activate line D_0. and 01 at the input will activate line D_1, and so on.

D_0 = A'B'
D_1 = B'A
D_2 = BA'
D_3 = BA



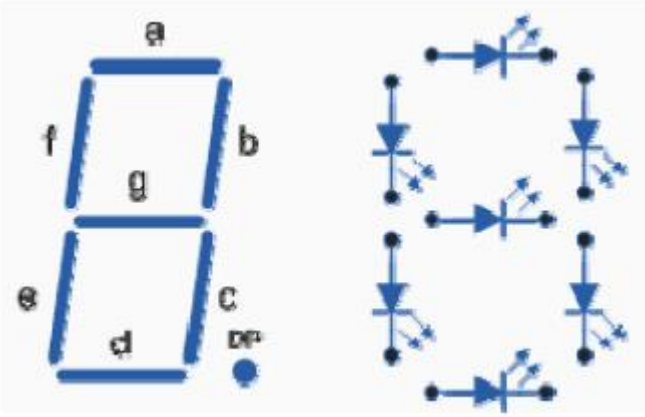
The logic diagram of 2-to- 4 Decoder

BCD to seven segments display decoder

There are two important types of 7-segment LED digital display.

• The Common Cathode Display (CCD) - In the common cathode display, all the cathode connections of the LEDs are joined together to logic "0" and the individual segments are illuminated by application of a "HIGH", logic "1" signal to the individual Anode terminals.

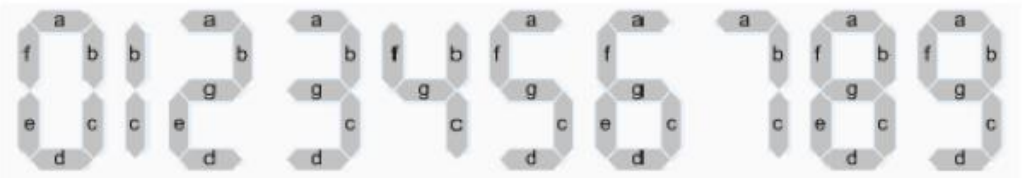
• The Common Anode Display (CAD) - In the common anode display, all the anode connections of the LEDs are joined together to logic "1" and the individual segments are illuminated by connecting the individual Cathode terminals to a "LOW", logic "0" signal.



Truth Table for a 7-segment display

Individual Segments							Display
a	b	c	d	e	f	g	
×	×	×	×	×	×		0
	×	×					1
×	×		×	×		×	2
×	×	×	×			×	3
	×	×			×	×	4
×		×	×		×	×	5
×		×	×	×	×	×	6
×	×	×					7

Individual Segments							Display
a	b	c	d	e	f	g	
×	×	×	×	×	×	×	8
×	×	×			×	×	9
×	×	×		×	×	×	A
		×	×	×	×	×	b
×			×	×	×		C
	×	×	×	×		×	d
×			×	×	×	×	E
×				×	×	×	F

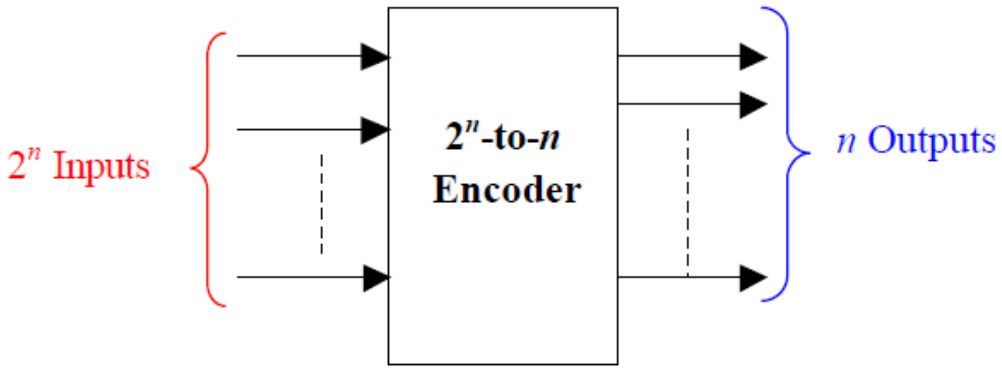


7-Segment Display Elements for all Numbers.

Different display decoders or drivers are available for the different types of display available, e.g. 74LS48 for common-cathode LED types, 74LS47 for common-anode LED types, or the CMOS CD4543 for liquid crystal display (LCD) types.

Encoders :

The encoder is a combinational circuit that performs the reverse operation of the decoder. The encoder has a maximum of 2^n inputs and n outputs. Only one input can be logic 1 at any given time (active input). All other inputs must be 0's.and the Output lines generate the binary code corresponding to the active input. The block diagram of 2^n –to-n encoder as shown below .



Example: 4-to-2 Encoders

The inputs are 4 and the outputs are 2

The block diagram and the truth table of a 4-to-2 encoder are shown below .

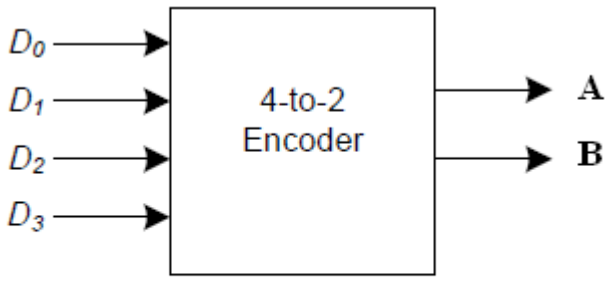


Figure Block Diagram of 4-to-2 Encoder

D_0	D_1	D_2	D_3	B	A
1	0	0	0	0	0
0	1	0	0	0	1
0	0	1	0	1	0
0	0	0	1	1	1

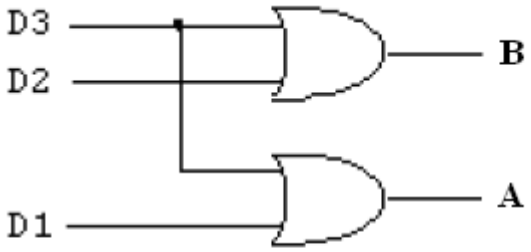
truth table For the 4-to-2 encoder

So that the logic expression of outputs are:

A = D1 + D3

B = D2 + D3

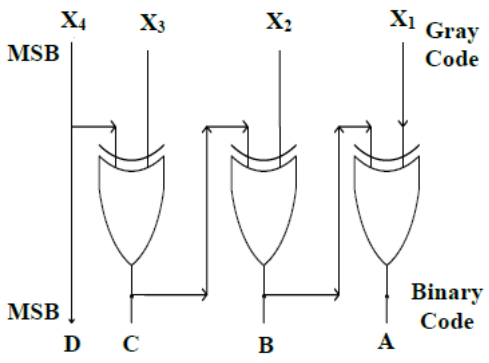
And the logic diagram of 4-to- 2 encoder as shown below:



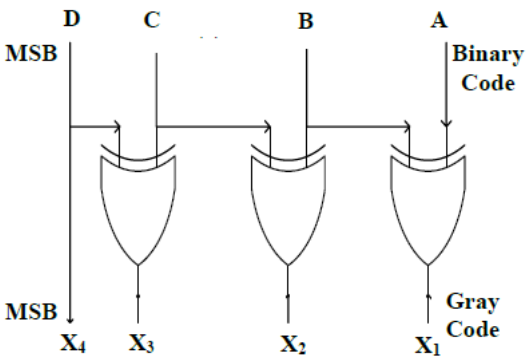
Logic Diagram of the 2-to-4 Encoder

Binary to Gray / Gray to Binary Conversion:

The gray code is widely used in many digital systems, specially in shaft encoders and analog to digital conversion, but it is difficult to use the gray-code in arithmetic operations, since there are only one bit change between two consecutive gray code number, and it is unweighted code, and the XOR gate is the most suitable gate for this purpose as shown in Figure below:



(a). Gary to Binary



(b). Binary to Gray