PES UNIVERSITY

PES UNIVERSITY
Dept. of Electronics and Communication Engineering 2025

UE25EC141A - Electronic Principles and Devices (4-0-0-4-4)

NOTES

UNIT- 4: Digital Electronics

Syllabus:

- Number Systems – binary and hexadecimal
- Binary Addition and Subtraction, 2's complement subtraction
- Boolean Algebra, Logic gates, Basic Theorems and Properties of Boolean Algebra
- Boolean Functions, Canonical and Standard Form, other Logical Operations.
- Combinational Logic Circuits: Half Adder and Full adder
- Sequential Circuits: RS, D, T, JK Flip-Flops, SISO Register, 3 Bit Asynchronous Up/Down counters.

Digital electronics is the branch of electronics that deals with the digital systems which processes the data/information in the form of binary (0s and 1s) numbers. These binary values correspond to two voltage levels, typically representing "low" (0) and "high" (1).

Number system

Number system a mathematical notation for representing numbers using digits or other symbols in a consistent manner. Each number system is characterized by its **base** or ***radix*****,** which determines the number of symbols used.

Base (or **Radix**) refers to the number of unique digits or symbols used in a number system.

For example: The **Decimal number system** has a base of 10, using digits 0 to 9.

The **Binary number system** has a base of 2, using digits 0 and 1.

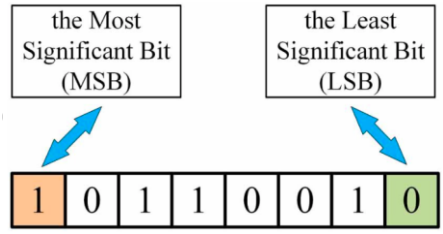
The **Hexadecimal number system** has a base of 16, using digits 0 to 9 and letters A to F.

Binary Number System

Key features

- The **binary number system** is a base-2 system that uses only two digits: **0** and **1**.
- Each digit in a binary number is called a **bit** (short for "binary digit").
- **Place Value:** Each bit in a binary number represents a specific power of 2, starting from 2^0 (rightmost bit) and increasing to 2^1 ,2^2,2^3, etc., as you move leftward.
- **Representation of Numbers:**
  Any numerical value can be represented in binary form using combinations of 0s and 1s.
- The number of **bits** in a binary number determines the total number of unique combinations or values that can be represented.
- If n is the number of bits, the maximum number values that can be represented using n bits equals 2^n
- In a binary number, the rightmost bit is called the **Least Significant Bit (LSB)** and the left most bit is the **Most Significant Bit (MSB).**

- The place value of the LSB is 2^0 and the next digits to the left gets the place value in terms of increasing powers of 2.
- In the below number, the place value of MSB is 2^7.



2 bit binary

Representation	Range
00	0
01	1
10	2
11	3

3 bit binary

Representation	Range
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

4 bit binary

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

Hexadecimal Number System

The hexadecimal (base-16) number system is a positional numeral system that uses 16 distinct symbols to represent values.

Symbols Used:

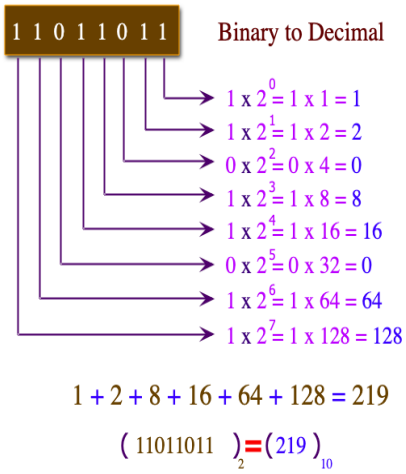
- **Digits:** 0, 1, 2, 3, 4, 5, 6, 7, 8, 9
- **Letters:** A, B, C, D, E, F
  (where A = 10, B = 11, C = 12, D = 13, E = 14, F = 15 in decimal).
- **Place Values:**
  Each digit in a hexadecimal number represents a power of 16, starting from 16^0 at the rightmost position
- Ex: 1A316, 2AC

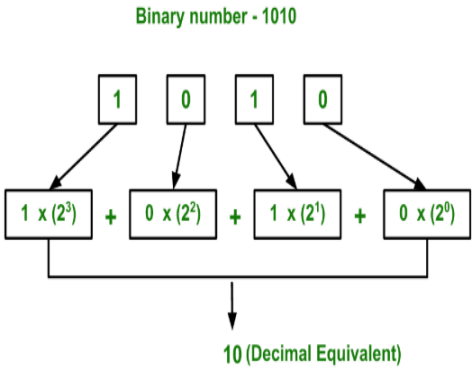
Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

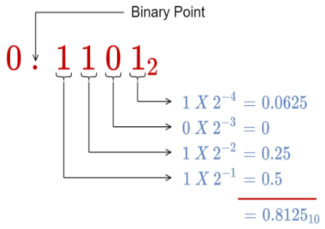
Number System Conversions

Binary to Decimal Conversion

- Multiply each binary digit by the value of its position or place value (power of 2).
- Add all the products together to get the decimal equivalent







- **Convert the following binary numbers to its decimal equivalent**

a) 1011

Starting from LSB,

$1\times 2^0 + 1\times 2^1 + 0\times 2^2 + 1\times 2^3$

$=1+2+0+8 = 11$

$(1011)_2 = (11)_{10}$

b) 11010

$0\times 2^0 + 1\times 2^1 + 0\times 2^2 + 1\times 2^3 + 1\times 2^4$

$=0+2+0+8+16 = 26$

$(11010)_2 = (26)_{10}$

c) 1111101

$(1111101)_2 = (125)_{10}$

d) 101.01011

$1\times 2^0 + 0\times 2^1 + 1\times 2^2 = 1+0+4 = 5$

$(.01011)_2 = 0\times 2^{-1}+1\times 2^{-2} + 0\times 2^{-3} +1\times 2^{-4} +1\times 2^{-5}$

$= 0.34375$

$(101.01011)_2 = (5.34375)_{10}$

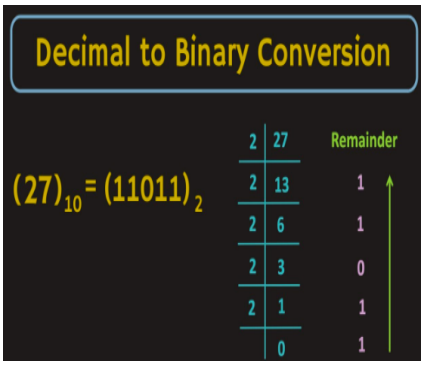
Decimal to Binary Conversion

- Divide the Decimal Number by 2.
- Record the quotient and the remainder.
- Repeat the division by 2 until the quotient is 0.
- The last remainder represents the MSB and the first remainder represents the LSB. The remainders represent the equivalent binary number

- **Convert the following decimal numbers to its binary equivalent**

(i) 4215 (ii) 0.6875

2 | 4215		
2 | 2107	1	← LSB
2 | 1053	1	
2 | 526	1	
2 | 263	0	
2 | 131	1	
2 | 65	1	
2 | 32	1	
2 | 16	0	
2 | 8	0	
2 | 4	0	
2 | 2	0	
2 | 1	0	
0	1	← MSB



	Integer		Fraction
0.6875 × 2 =	1	+	0.3750
0.3750 × 2 =	0	+	0.7500
0.7500 × 2 =	1	+	0.5000
0.5000 × 2 =	1	+	0.0000

$(0.6875)_{10} = (0.1011)_2$

Hexadecimal to Decimal Conversion

- Convert each hexadecimal bit into its decimal equivalent.
- Multiply each bit by the value of its position or place value (power of 16).
- Add all the products together to get the decimal equivalent

Examples:

(i) $(2F36)_{16}$

$= 6x16^0+3x16^1+15x16^2+2x16^3$

$= 8192+3840+48+6$

$=12086$

(ii) $(B65F)_{16}$

$$(B65F)_{16} = 11 \times 16^3 + 6 \times 16^2 + 5 \times 16^1 + 15 \times 16^0 = (46,687)_{10}$$

(iii) $(54.D2)_{16}$

$54.D2_{16}$

$= 5 \cdot 16^1 + 4 \cdot 16^0 + D \cdot 16^{-1} + 2 \cdot 16^{-2}$

$= 5 \cdot 16^1 + 4 \cdot 16^0 + 13 \cdot 16^{-1} + 2 \cdot 16^{-2}$

$= 80 + 4 + 0.8125 + 0.0078125$

$= 84.8203125$

Decimal to Hexadecimal Conversion

- **Divide the Decimal Number by 16:** Divide the decimal number by 16. The quotient will be used for the next division, and the remainder will be one of the hexadecimal digits.
- **Record the Remainder:** The remainder of each division corresponds to a hexadecimal digit. If the remainder is between 10 and 15, use the hexadecimal letters:

Hexadecimal	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
Decimal	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- **Repeat the Division:** Divide the quotient by 16 again, and continue recording the remainders until the quotient is 0.
- **Read the Remainders in Reverse Order**: The remainders, when read from last to first, form the hexadecimal representation of the decimal number.

Example 1: Convert $(255)_{10}$ to Hexadecimal

Step 1: Divide 255 by 16:

$255 \div 16 = 15$ (quotient) remainder $= 15$

Step 2: Divide the quotient (15) by 16:

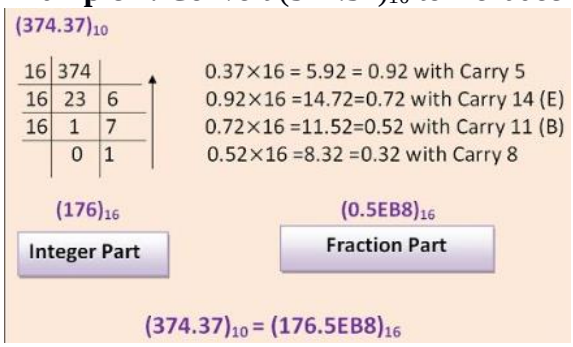
$15 \div 16 = 0$ (quotient) remainder $= 15$

Step 3: The quotient is now 0, so stop dividing.

Step 4: Read the remainders in reverse order: The remainders, from last to first, are F and F.

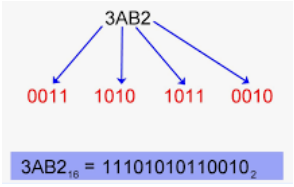
Thus, 255 (decimal) = FF (hexadecimal).

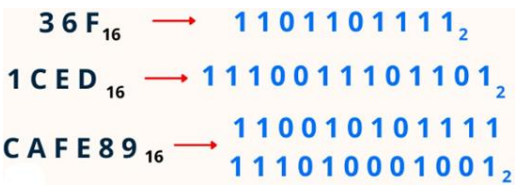
Example 2: Convert $(374.37)_{10}$ to Hexadecimal



Hexadecimal to Binary conversion

- Replace each hexadecimal digit with its equivalent 4-bit binary representation

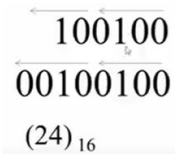




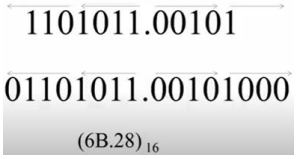
Binary to Hexadecimal conversion

- Break down the binary number into groups with 4 digits in each group, write the hexadecimal equivalent of each of the groups.
- Combine all the numbers together to get the hexadecimal number.

(i) $(100100)_2$



(ii) $(1101011.00101)_2$



1's Complement

- The 1's complement of a binary number is the result of inverting all the bits in the number's binary representation.
- This means swapping all 0s for 1s and vice versa.
- For example, the 1's complement of 1010 is 0101.

In 1's complement representation, the representation of the negative number is different.
For example, if we want to represent -34 in 8-bit 1's complement form, then first write the positive number (+34). And invert all 1s in that number by 0s and 0s by 1s in that number. The corresponding inverted number represents the -34 in 1's complement form. It is also called 1s complement of the number +34.

To represent -34 in 1's complement form

```
+ 34 =  0 0 1 0 0 0 1 0
        ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
- 34 =  1 1 0 1 1 1 0 1   (1's complement of + 34)
```

2's complement

- 2's complement is the most common method used to represent signed numbers

To find the 2's complement
(i) Find the 1's complement
(ii) Add 1 to the 1's complement

To represent -34 in 2's complement form

```
+ 34 =  0 0 1 0 0 0 1 0
        ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
        1 1 0 1 1 1 0 1   (1's complement of + 34)
      +               1
      ---------------------
- 34 =  1 1 0 1 1 1 1 0   (2's complement of + 34)
```

NOTE:

- Range of Values for Unsigned Binary Numbers
  In **unsigned binary**, all values are non-negative, starting from 0.
  Range= 0 to (2^n-1)

For ex: With n=1 bit: no of values is 2^1=2 values (0,1)
With n=2 bits: no of values is 2^2=4 values (00,01,10,11), 0 to 3 in decimal.
With n= 3 bits: 2^3=8 values (000-111), 0 to 7 in decimal

- Range of Values for **Signed Binary Numbers** (2's complement):
  Range= -2^{n-1} to $(2^{n-1}-1)$
  For ex: With 2 bits: 2^2=4 values (−2 to 1), −2,−1,0,1
  With 3 bits: 2^3=8 values (−4 to 3), −4,−3,−2,−1,0,1,2,3

Binary Addition

(1) 8 + 5 = ?
Binary representation of 8 = 00001000
Binary representation of 5 = 00000101

```
  00001000
+ 00000101
----------
  00001101
```

Sum: 0 0 0 0 1 1 0 1

Sign bit is zero, so the result is a positive number

```
  8
+ 5
---
 13
```

www.vlsifacts.com

13 => 00001101

(2) 6 + 13 = ?

```
+ 6    00000110
+13    00001101
+19    00010011
```

Binary Subtraction using 2's Complement method

Steps to find A-B=?

1. At first, find 2's complement of the B (subtrahend).
2. Then add it to the A (minuend).
3. If the final carry-over of the sum is 1, then it is dropped and the result is positive. →(A>B)
4. If there is no carry over, then 2's complement of the sum is the final result and it is negative. → (A<B)

(1) 8 -5= ?

Binary representation of 8 = 00001000
Binary representation of 5 = 00000101
Step-1: 2's complement of $(00000101)_2$ is $(11111011)_2$.
Step-2: Add (00001000)2 to (11111011)2. This is shown below.

```
  00001000
+ 11111011
----------
1 00000011
```

Discard Carry

Sum: 0 0 0 0 0 0 1 1

```
  8
+ -5
----
  3
```

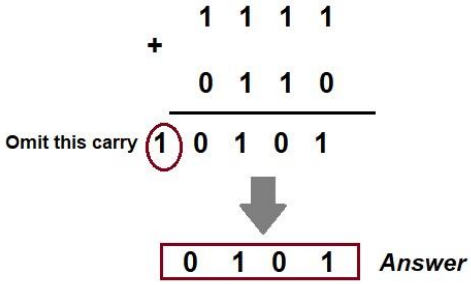
www.vlsifacts.com

3 => 00000011

Step-3: The Final carry is 1 indicating that answer is positive and drop the carry

(2) Subtract $(1010)_2$ from $(1111)_2$ using 2's complement method.

Solution: $(1111)_2$-$(1010)_2$=?
Step-1: 2's complement of $(1010)_2$ is $(0110)_2$.
Step-2: Add $(0110)_2$ to $(1111)_2$. This is shown below.



Step-3: The Final carry is 1 indicating that answer is positive and drop the carry

(3) Subtract $(1010)_2$ from $(1000)_2$ using 2's complement.

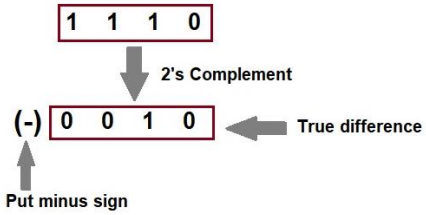
Solution:

Step-1: The 2's complement of $(1010)_2$ is $(0110)_2$.

Step-2: Add $(0110)_2$ to $(1000)_2$.

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

Step-3: No final carry, i.e, answer is negative. So, take 2's complement of the final sum and it is negative.



(4)

Given the two binary numbers $X = 1010100$ and $Y = 1000011$, perform the subtraction **(a)** $X - Y$ and **(b)** $Y - X$ by using 2's complements.

(a)
$$\begin{aligned} X &= \quad 1010100 \\ \text{2's complement of } Y &= + \quad \underline{0111101} \\ \text{Sum} &= \quad 10010001 \\ \text{Discard end carry } 2^7 &= - \underline{10000000} \\ \textit{Answer: } X - Y &= \quad 0010001 \end{aligned}$$

(b)
$$\begin{aligned} Y &= \quad 1000011 \\ \text{2's complement of } X &= + \quad \underline{0101100} \\ \text{Sum} &= \quad 1101111 \end{aligned}$$

There is no end carry. Therefore, the answer is $Y - X = -(\text{2's complement of } 1101111) = -0010001$.

(5) Subtract 100011 from 10101 using 2's complement method

Step 1: Make equal number of bits

Minuend= **0**10101

Subtrahend=100011

Step 2: The 2's complement of $(100011)_2$ is $(011101)_2$.

Step 3: Add $(011101)_2$ with minuend $(010101)_2$

```
  011101
+ 010101
--------
  110010
```

Step-3: No final carry, i.e, answer is negative. So, take 2's complement of the final sum and it is negative

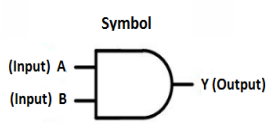
Final answer: -001110

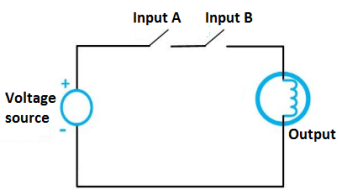
Boolean Algebra and Logic gates

- Digital Electronic circuits process data that contains **binary values 1's and 0's.**
- A **Logic "1**" is also referred as HIGH voltage or TRUE or ON state.
- A **Logic "0"** is also referred as LOW voltage or FALSE or OFF state
- A binary digit "0" or "1" is called a **bit**
- Digital information is stored using a series of binary values 1's and 0's.
- Digital systems such as digital telephones, digital cameras, computers, handheld portable devices and other high technology systems **stores and process** these binary values (1's and 0's)
- Boolean Algebra is the mathematics used to analyse and **simplify logic or digital circuits**.
- Digital circuits are constructed by using **logic gates**.
- Logic gates are the basic building blocks of digital systems, performs logical functions based on Boolean algebra.
- Logic gates have one or more inputs and only one output.
- Number of possible input states is $\mathbf{2^n}$. Where n is Number of inputs
- Logic gates are implemented using diodes or transistors which acts as electronic switches
- Logic gates : (i) **Basic Gates:** AND, OR, NOT,
  (ii) **Derived Gates:** NAND, NOR, EXOR and EXNOR gates

AND Gate

- **AND Gate** is an electronic circuit that gives output as logic "1" (HIGH) only if all inputs are "1". Otherwise output is Logic "0" (LOW)
- Boolean algebra representation for AND gate is **Y = A.B**
- AND gate is represented as **series connection** of two switches



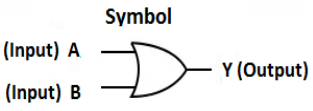


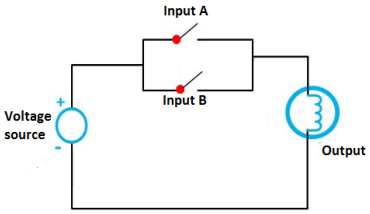
Truth Table

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

OR Gate

- **OR Gate** is an electronic circuit that gives as Logic "1" (HIGH) if any of the inputs are HIGH .Output of OR gate goes Logic "0" (LOW) only if all inputs are Logic "0" (LOW)
- Boolean algebra representation of OR gate is Y = A + B
- OR gate is represented as **parallel connection** of two switches



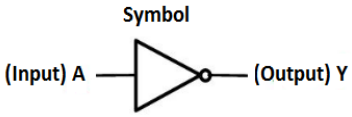


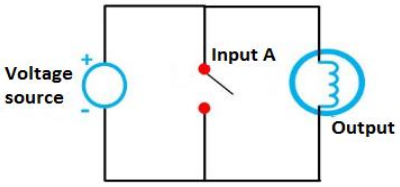
Truth Table

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

NOT Gate

- **NOT Gate** is a single input and single output gate which performs the inversion of the applied binary input signal. Hence it is also called as Inverter Gate.
- Boolean expression for NOT gate is $Y = \overline{A}$
- NOT Gate can be represented by connecting a switch parallel to bulb

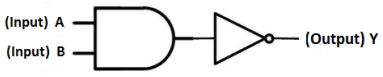


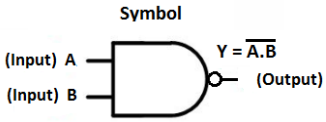


Truth Table

Input	Output
A	Y
0	1
1	0

NAND Gate

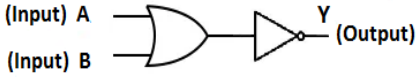


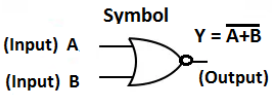


Truth Table

NAND		
Input		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate





Truth Table

NOR		
Input		Output
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

Logical Boolean Laws:

NOT / Inversion Law:

$\overline{1} = 0$

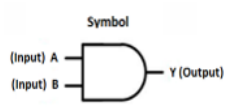
$\overline{0} = 1$

(i) If A = 1 then A' = 0 and if A = 0 then A' = 1

(ii) (A')' = A itself

AND Law

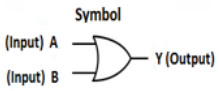
❖AND Law:



0.0 = 0	A.0 = 0
0.1 = 0	A.1 = A
1.0 = 0	A.A = A
1.1 = 1	A.A′ = 0

OR Law

❖ OR Law:



0+0 = 0	A+0 = A
0+1 = 1	A+1 = 1
1+0 = 1	A+A = A
1+1 = 1	A+A′ = 1

Principle of duality in Boolean Algebra

- In Boolean Algebra, One type of expression can be converted into another type of expression by replacing "0 with 1 " , " 1 with 0" , "(+) sign with (.) sign " and vice versa.

Example:

AND (.)	OR (+)
0.0 = 0	1+1 = 1
0.1 = 0	1+0 = 1
1.0 = 0	0+1 = 1
1.1 = 1	0+0 =0

Expression	Dual Expression
$\bar{0} = 1$	$\bar{1} = 0$
0.1 = 0	1 + 0 = 1
A.0 = 0	A + 1 = 1
A.B = B. A	A + B = B + A
$A.\bar{A} = 0$	$A + \bar{A} = 1$

Basic Theorem and Properties of Boolean Algebra

- Boolean Laws and Theorems are used to simplify the Boolean expressions. Hence reduce the number of logic gates.

Commutative Laws:

(i) A+ B = B+ A

(ii) A.B = B.A

A	B	(A+B)	(B+A)	(A.B)	(B.A)
0	0	0	0	0	0
0	1	1	1	0	0
1	0	1	1	0	0
1	1	1	1	1	1

Associative Law:

(A+B)+C = A+(B+C) Boolean addition

A	B	C	A + B	(A + B) + C	B + C	A + (B + C)
0	0	0	0	0	0	0
0	0	1	0	1	1	1
0	1	0	1	1	1	1
0	1	1	1	1	1	1
1	0	0	1	1	0	1
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

- From principle of duality: (A.B).C = A.(B.C) Boolean Multiplication

Distributive Law:

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

A	B	C	BC	A+BC	(A+B)	(A+C)	(A+B)(A+C)
0	0	0	0	0	0	0	0
0	0	1	0	0	0	1	0
0	1	0	0	0	1	0	0
0	1	1	1	1	1	1	1
1	0	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	1	0	0	1	1	1	1
1	1	1	1	1	1	1	1

Dual of distributive law:

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

A	B	C	B+C	A(B+C)	AB	AC	AB+AC
0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	1	1	1	0	0	0	0
1	0	0	0	0	0	0	0
1	0	1	1	1	0	1	1
1	1	0	1	1	1	0	1
1	1	1	1	1	1	1	1

De Morgan's Theorem

(i) The complement of the sum of 2 variables is equal to the product of the complements of individual variables:

$$\overline{A + B} = \overline{A} \cdot \overline{B}$$

(ii) The complement of the product of 2 variables is equal to the sum of the complements of individual variables:

$$\overline{A.B} = \overline{A} + \overline{B}$$

A	B	$\overline{A}$	$\overline{B}$	$A+B$	$A.B$	$\overline{A+B}$	$\overline{A} \cdot \overline{B}$	$\overline{A \cdot B}$	$\overline{A} + \overline{B}$
0	0	1	1	0	0	1	1	1	1
0	1	1	0	1	0	0	0	1	1
1	0	0	1	1	0	0	0	1	1
1	1	0	0	1	1	0	0	0	0

Absorption Theorem:

(i) A+AB = A

LHS: = A + AB
= A.1 + AB → since A.1 = A
= A(1+B) → since 1 + B = 1
= A.1
= A = RHS

(ii) A(A+B) = A

LHS = A (A + B)
= A.A + A.B
= A+AB → since A . A = A
= A (1 + B)
= A.1
= A = RHS

(iii) A+$\bar{A}$B = A+B

LHS = A + $\bar{A}$B
= (A + $\bar{A}$) (A + B) → since A+BC = (A+B)(A+C)
= (1) . (A + B) → since A + $\bar{A}$ = 1
= A + B = RHS

(iv) A.($\bar{A}$+B) = AB

LHS = A.($\bar{A}$ + B)
= A. $\bar{A}$ + A.B → (A $\bar{A}$ = 0)
= AB = RHS

Consensus Theorem:

LHS = AB+$\bar{A}$C+BC
= AB + $\bar{A}$C + BC.1
= AB + $\bar{A}$C + BC (A + $\bar{A}$) → since A + $\bar{A}$ = 1
= AB + $\bar{A}$C + ABC + $\bar{A}$BC
= AB (1 + C) + $\bar{A}$C (1 + B)
:1 + B = 1 + C = 1
= AB + $\bar{A}$C = RHS

Dual of consensus theorem:

(A+B)($\bar{A}$+C)(B+C) = (A+B)($\bar{A}$+C)

LHS= (A+B)($\bar{A}$+C)(B+C)

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

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

LHS= (B+ AC) ($\bar{A}$+C)
= $\bar{A}$B +$\bar{A}$ AC+ BC+ ACC
= $\bar{A}$B + BC+ AC +A$\bar{A}$
= B($\bar{A}$ +C) + A($\bar{A}$ +C)
= (A+B)($\bar{A}$+C) =RHS

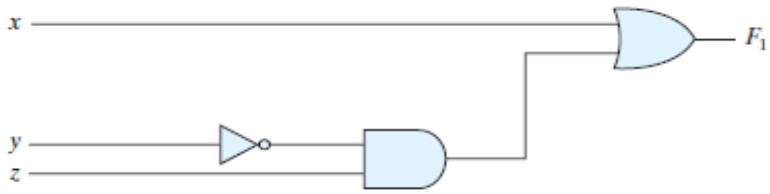
Boolean Functions, Canonical and Standard Form

- Boolean function described by an algebraic expression consists of **binary variables, the constants 0 and 1, and the logic operation symbols**.
- Boolean function is **evaluated to logic-1 or logic-0** for a given value of the binary variables.
- Boolean function can be represented in a **truth table**
- A Boolean function can be implemented as **digital circuit**,
- Which is constructed by using logic gates.
- Example: F = X + Y'Z
  F = 1 if X = 1 or if Y = 0 and Z = 1
  F = 0 Otherwise
- Example: F1 = x + y'z
- F1 contains either 0 or 1 for each of these combinations. The table shows that the function is equal to 1 when x = 1 or when yz = 01 and is equal to 0 otherwise

Truth Table:

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

Gate Level Implementation:

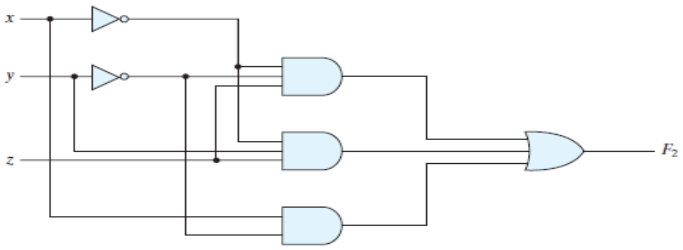


- Simplify and realize the given Boolean function using basic gates
  F2 = x'y'z + x'yz + xy'…….(1)
  F2 = x'z (y' + y) + xy'
  F2 = x'z + xy'……..(2)

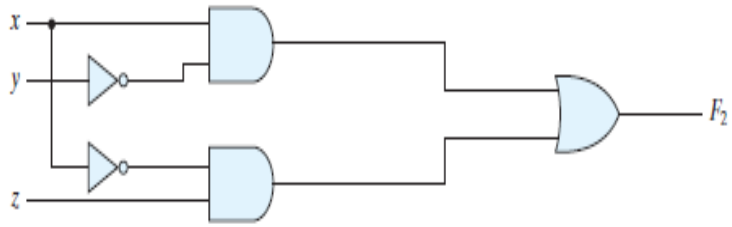
Truth Table

x	y	z	F_2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Logic Diagram for eq.1



Logic Diagram for eq.2

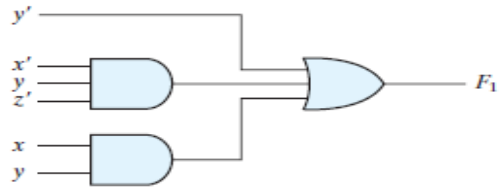


Sum of Products: (SOP)

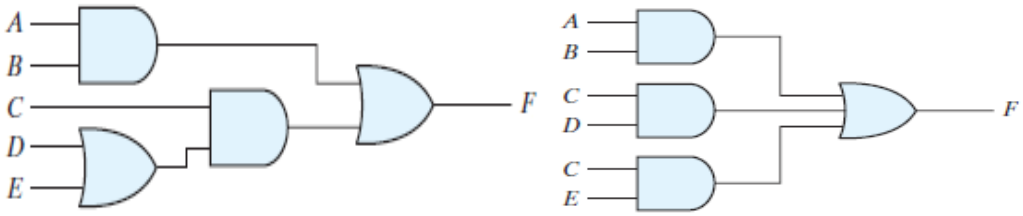
- The sum of products is a Boolean expression containing **AND** terms, called **product terms,** with one or more literals each. The **sum** denotes the **OR**ing of these terms.

Two- Level Logic implementation of SOP

Example 1: F1 = y' + xy + x'yz'



Example 2: F = AB + C (D+E)



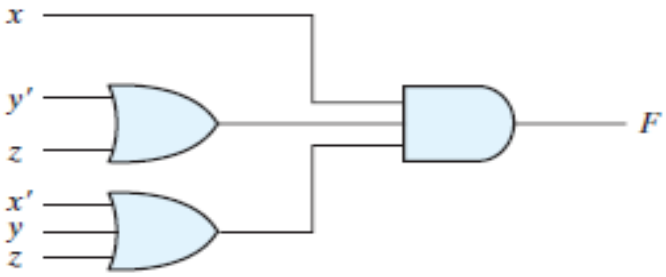
F = AB + CD + CE

Product Of Sum: POS

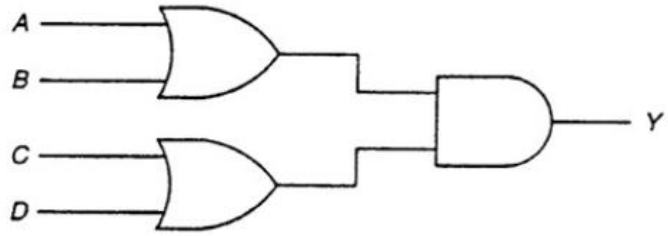
- **A product of sum (POS) is a Boolean expression containing OR terms, called sum terms.**

Each term may have any number of literals. The product denotes the ANDing of these terms

- **Example 1 : F = x.(y' + z).(x' + y + z)**



- **Example 2: Y = (A+B). (C+D)**



- **Canonical SOP Form**
- **Each product term contains all the literals of that function either in true or complement form**
- **Example: F(x,y,z) = xyz + x'y'z + x'yz' +x'y'z'**
- **Each product term is called as minterm**
- **For three variable function: Truth Table**

x	y	z	Function f_1	Function 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

			Minterms	
x	y	z	Term	Designation
0	0	0	$x'y'z'$	m_0
0	0	1	$x'y'z$	m_1
0	1	0	$x'yz'$	m_2
0	1	1	$x'yz$	m_3
1	0	0	$xy'z'$	m_4
1	0	1	$xy'z$	m_5
1	1	0	xyz'	m_6
1	1	1	xyz	m_7

- **Consider the function: f1 and f2 in the truth table**
- **Canonical SOP Form for f1**
- **f1 = x'y'z + xy'z' + xyz = m1 + m4 + m7**
- **Canonical SOP Form for f2**
- **f2 = x'yz + xy'z + xyz' + xyz = m3 + m5 + m6 + m7**

Canonical POS Form

- **Each sum term contains all the literals of that function either in true or complement form**
- **Example: F(x, y, z) = (x +y+z) (x'+y'+z) (x'+y+z')**

➢ **Each sum term is called as Maxterm**

			Maxterms	
x	y	z	Term	Designation
0	0	0	$x + y + z$	M_0
0	0	1	$x + y + z'$	M_1
0	1	0	$x + y' + z$	M_2
0	1	1	$x + y' + z'$	M_3
1	0	0	$x' + y + z$	M_4
1	0	1	$x' + y + z'$	M_5
1	1	0	$x' + y' + z$	M_6
1	1	1	$x' + y' + z'$	M_7

x	y	z	Function f_1	Function 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

❖Consider the POS Form for f1 and f2 in the truth table
f1 = (x + y + z) (x + y' + z)(x + y' + z')(x' + y + z')
(x' + y' + z)
= M0. M2 .M3 . M5 . M6
f2 = (x + y + z)(x + y + z')(x + y' + z)(x' + y + z)
= M0. M1. M2. M4

- Find the Minterms for the given expression: (Convert into canonical SOP Form) F = A + BC

F = A.1.1 + BC.1
F = A. (B+B'). (C+C'). + BC.(A+A')
F = A. (BC + BC' +B'C +B'C') + ABC +A'BC
F = ABC + ABC' + AB'C + AB'C' + ABC + A'BC F = ABC + ABC' + AB'C + AB'C' +A'BC
F = m7 + m6 +m5 + m4 + m3
F = Σ (3,4,5,6,7)

Convert the given expression F(A, B, C) = A + B'C into canonical SOP form

We need to ensure that each term includes all three variables (A, B, and C). Expanding A
A:$A=A(B+B')=AB+AB'$A=A(B+B ′)=AB+AB ′
Now, expand each term to include C:$AB=AB(C+C')=ABC+ABC'$AB=AB(C+C′)=ABC+ABC′
$AB'=AB'(C+C')=AB'C+AB'C'$AB ′
=AB′(C+C′)=AB ′ C+AB ′ C ′
Expanding $B'C=B'C(A+A')=AB'C+A'B'C$

Final Canonical SOP Form

Combining all unique min terms: F(A,B,C)=ABC+ABC′+AB′C+AB′C′+A′B′C
F(A,B,C)=∑m(1,4,5,6,7)

Note:

An alternative procedure for deriving the minterms of a Boolean function is to obtain the truth table of the function directly from the algebraic expression and then read the minterms from the truth table.

❖ **Express the Boolean function F = x y + x ′ z as a product of maxterms. (Convert the given expression** F = x y + x ′ z **into canonical POS form).**

First, convert the function into OR terms by using the distributive law, x +y z = (x + y) (x + z)

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

= M0 M2 M4 M5
F (x , y , z) = Π (0 , 2 , 4 , 5)

❖ XOR Gate is a digital logic Gate which has two or more inputs and only one output that performs Exclusive OR operation. Hence it is also called as Ex-OR or XOR

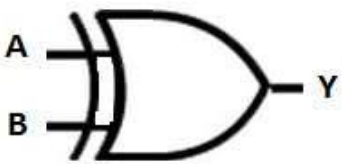
❖ For two input XOR gate output is logic-1 only when one of its input is logic-1 (unequal input i.e..A = 0 and B = 1 or A = 1 and B = 0).

❖ Output of XOR gate is logic-0 if both inputs are same (i.e.., A = 0 and B = 0 or A = 1 and B = 1)

❖ Truth Table of XOR

Input A	Input B	Output Y
0	0	0
0	1	1
1	0	1
1	1	0

Symbol



$A.B' + A'.B = A \oplus B$

❖ XOR gate performs modulo sum operation without including carry. i.e.., 0+0 = 0 , 0+1 = 1, 1+0 = 1, 1+1 = 0 (carry 1)

❖ Truth T

Y = A'.B +
AY.B='A XOR B Y = A ⊕ B

A.B' + A'.B = A ⊕
B
LHS = A.B' +

A'.B
Case1: If A = 0 and B = 0
0.0' + 0'.0 = 0.1 + 1.0 = 0

Case2: If A = 0 and B = 1 0.1' + 0'.1 = 0.0 + 1.1 = 1

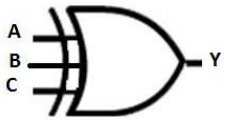
Case3: If A = 1 and B = 0
1.0' + 1'.0 = 1.1 + 0.0 = 1
Case4: If A = 1 and B = 1
1.1' + 1'.1 = 1.0 + 0.1 = 0

Hence Y = A ⊕ B = A.B' +
A'.B

- **3 - Input XOR Gate:** **Y = A⊕B ⊕ C**

Truth

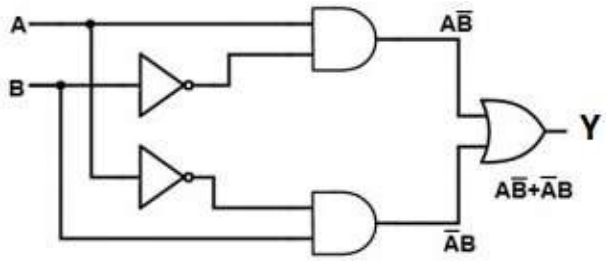
Symbol



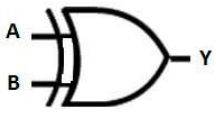
- **XNOR Gate**: Logical Complement of XOR Gate
- For two input XNOR Gate if both inputs are same i.e.., A = 0 and B = 0 or A = 1 and B = 1. Then output of logic gate is logic-1 (High). Output of XNOR is logic-0 if inputs are unequal.
- Equality detector
- Implementation of XOR Gate using basic gates:

Y = A ⊕ B = A.B' + A'.B

- Logic Diagram



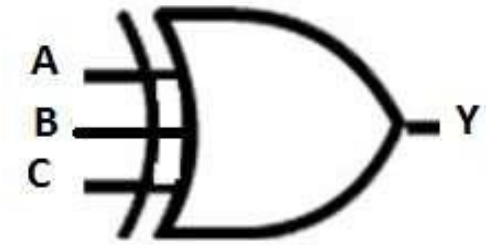
- Symbol



XNOR Gate:
Y = (A⊕B)'
Y = (A.B' + A'. B)'
De Morgan's Theorem

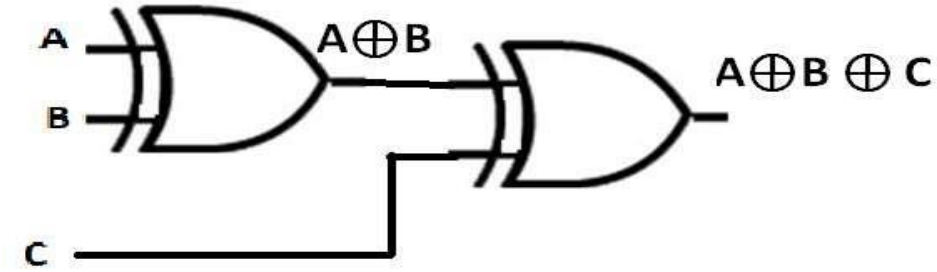
Y = (A.B')' .(A'. B)'
Y = (A'+B) . (A + B')
Y = A'. A + A'.B' + A.B + B.B'
Y = 0 + A'.B' + A.B + 0
Y = A'.B' + A.B

- **3 - Input XOR Gate:**
- **Y = A⊕B ⊕ C**



A	B	C	Y	A⊕B ⊕ C
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	0
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	0	1

Associative Law:
(A⊕B)⊕ C = A⊕(B ⊕ C) ❖ **Even or odd Parity Bits**



❖ **XNOR Gate**: Logical Complement of XOR Gate
For two input XNOR Gate if both inputs are same i.e.., A = 0 and B = 0 or A = 1 and B = 1.

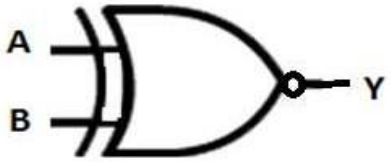
❖ Then output of logic gate is logic-1 (High). Output of XNOR is logic-0 if inputs are unequal.

❖ Equality detector

Truth

Input A Ta	bInleput B	Output Y
0	0	1
0	1	0
1	0	0
1	1	1

Boolean Expression
Y = A.B + A'.B'
Y = A ⊙ B



- **XNOR Gate:**

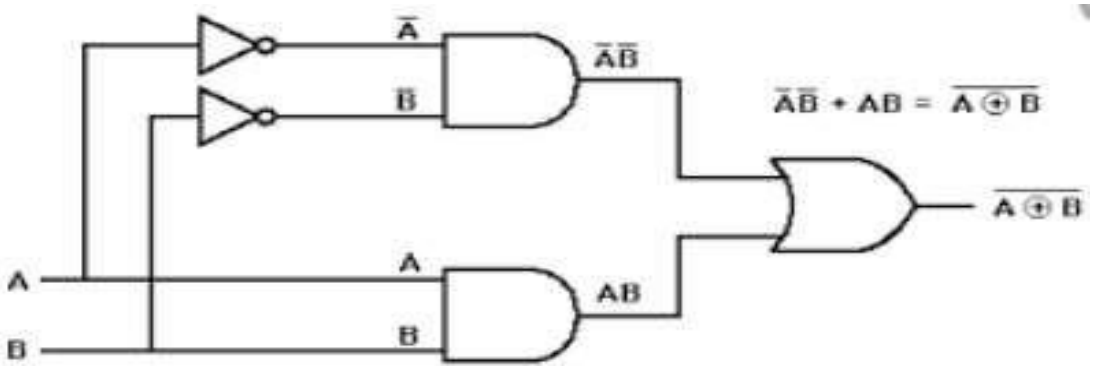
Y = (A⊕B)'
Y = (A.B' + A'. B)'
De Morgan's Theorem
Y = (A.B')' . (A'. B)'
Y = (A'+B) . (A + B')
Y = A'. A + A'.B' + A.B + B.B'
Y = 0 + A'.B' + A.B + 0
Y = A'.B' + A.B
= (A⊕B)' = A⊙B
Y = (A XOR B)' = (A XNOR B)
Logic Diagram

Symbol:

- **Implement the XNOR Gate using basic gates:**

Y = A'.B' + A. B

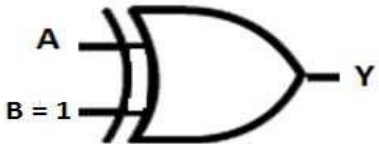
Logic Diagram:



Y = (A⊕B)' = A⊙B

Properties of XOR Gate:

- **Identity element: A ⊕ 0 = A**
- **A ⊕ 1 = A'**
- **A ⊕ A = 0**
- **Commutative Law: A ⊕ B = B ⊕ A**
- **Associative Law : A ⊕ (B ⊕ C) = (A ⊕ B) ⊕ C**



Input A	Input B	Output Y
0	0	0
0	1	1
1	0	1
1	1	0

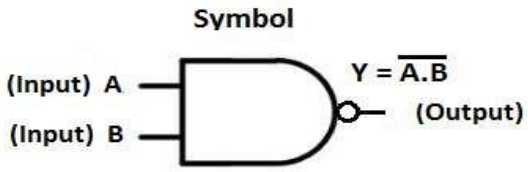
Applications of XOR and XNOR

- XOR gate is used in processor's Arithmetic Logic Unit (ALU) for binary addition.
- XOR logic gate is used to generate pseudorandom numbers in hardware.
- To generate parity bits and error detection
- Equality detector

Universal Gates : (i) NAND Gate
(ii) NOR Gate

Any digital logic circuit can be implemented by using NAND or NOR logic gates.
NAND and NOR gates are easier to fabricate with electronic components and are used in all Integrated Circuit (IC's) digital logic families.

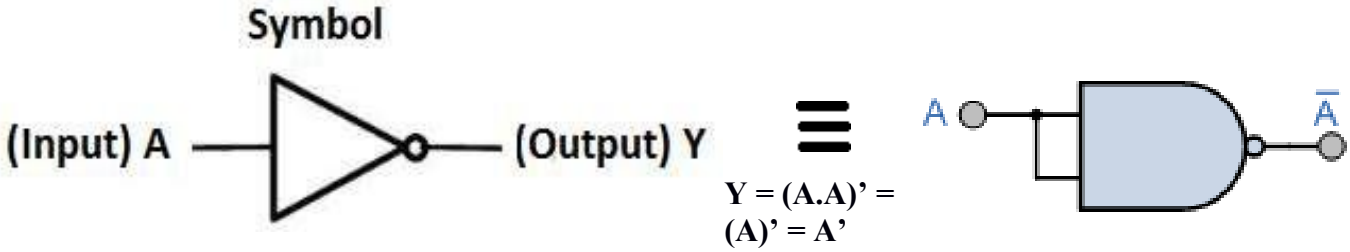
Realization of logic gates using NAND Gates:



Truth Table

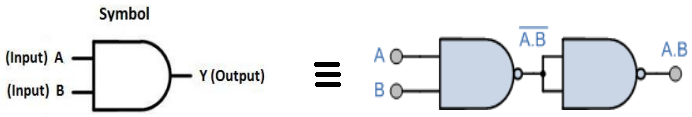
NAND		
Input		Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOT Gate



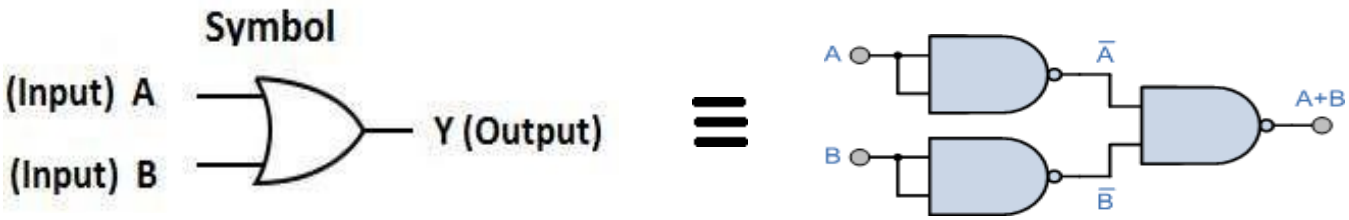
Y = (A.A)' = (A)' = A'

❖ **AND Gate:**



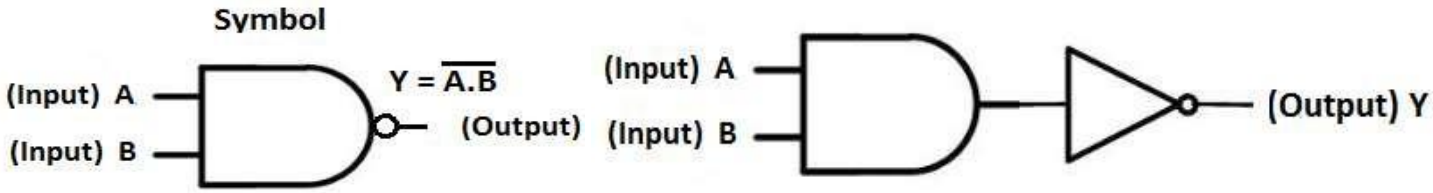
Y = ((A.B)')' = A.B

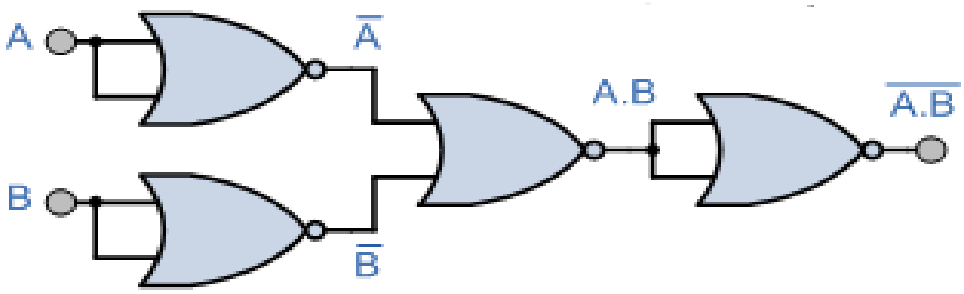
❖ **OR Gate:**



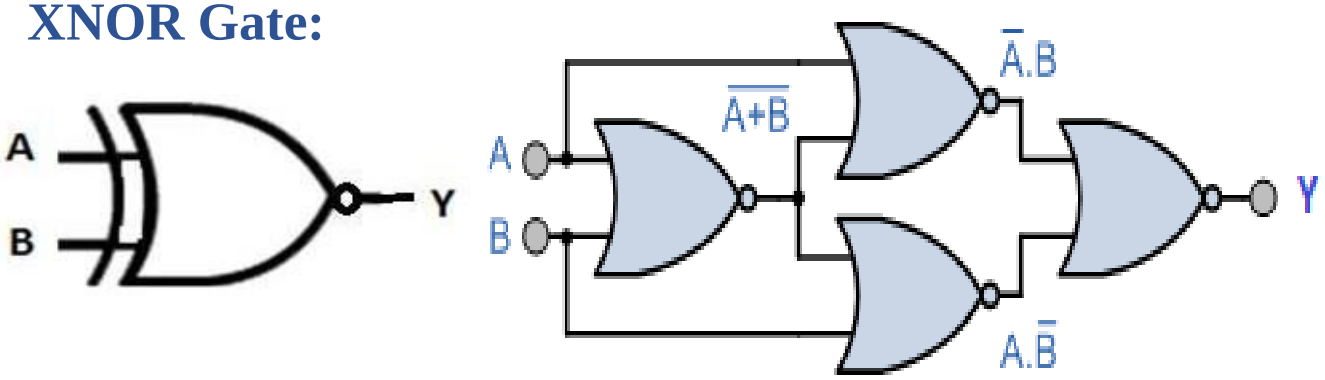
Y = ((A+B)')' = (A' . B')' = A+B

NAND Gate:





XNOR Gate:



Y1 = ((A+B)′ + A)′ = (A+B). A′ = A′.B

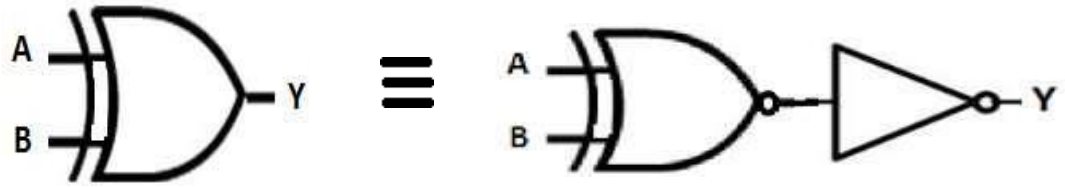
Y2 = ((A+B)′ + B)′ = (A+B). B′ = A.B′

Y = (A′.B + A.B′)′ = (A′.B)′ . (A.B′)′ = (A + B′) . (A′ + B)

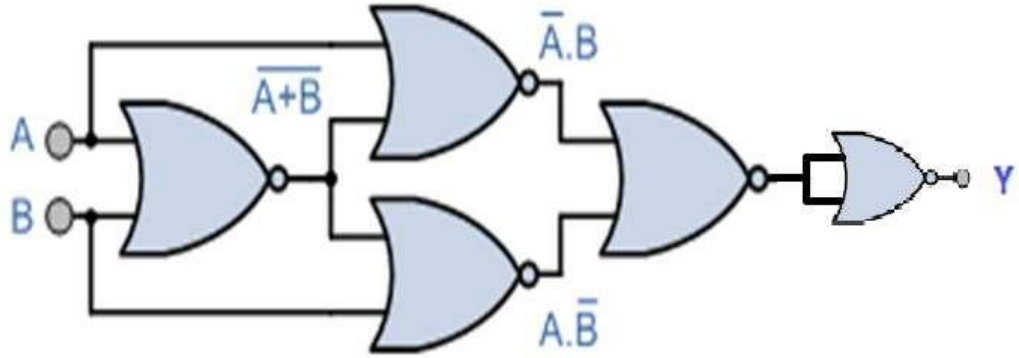
Y = A.B + A′.B′

Y = A⊙B

XOR Gate:



≡

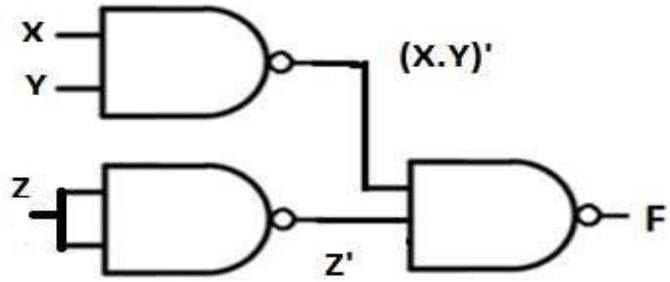


Implement the given function using NAND Gates only.

F = X.Y + Z

F = ((X.Y + Z)'

)F' = ((X.Y)' . Z'))'



Problems on Boolean algebra

(1) Simplify the given Boolean expression and implement using basic gates

F = A.B.C + A' + A. B'.C

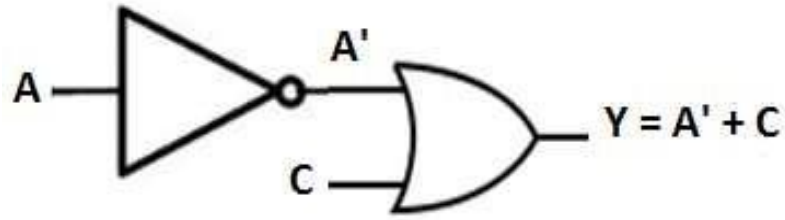
F = A.C (B + B')+ A' where B + B' = 1

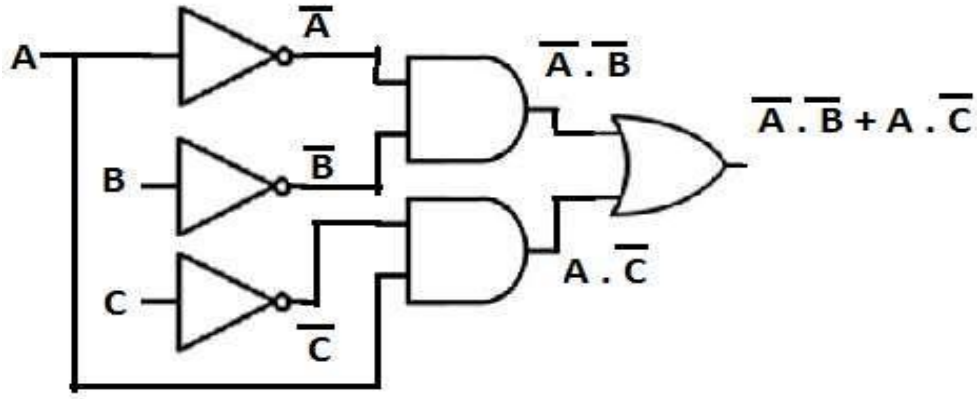
F = A.C + A'

F = (A'+A) . (A'+ C) F = A' + C

Distributive Law

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





(2) Simplify and realize the given function using basic gates:

Y = A'B'C' + AB'C' + A'B' + AC'

Y = B'C' (A' + A) + A'B' + AC'

Y = B'C' . (1) + A'B' + AC'

By Consensus Theorem: Y = A'B' + AC' AB + ĀC+BC = AB+ĀC

(3)Simplify and Implement the given function using NAND Gates only.

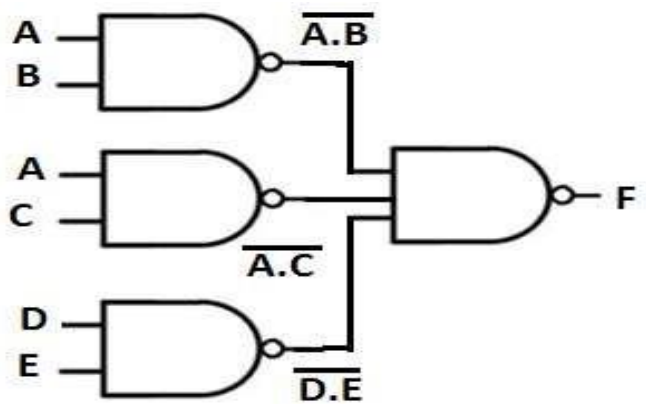
F = A(B + C) + DE

- **SOP Form: F = AB+ AC + DE**

❖ **F = (AB+ AC + DE)**

$$F = \overline{(AB+ AC + DE)}$$

$$F = \overline{((A.B) . (A.C) . (D.E))}$$



DeMorgan's Law

(4)Simplify the given Boolean Expression

Y = A + A'.B + A'.B'.C + A'.B'.C'.D + A'.B'.C'.D'.E

Y = A + A' (B + B'.C + B'.C'.D +

B'.C'.D'.E)

Y = A + B + B'.C + B'.C'.D +

B'.C'.D'.E

Y = A + B + B' (C + C'.D +

C'.D'.E)

Y = A + B + C + C'.D + C'.D'E

Y=A+B+C+D+E

Absorption Law: A + A'.B = A + B

5) Evaluate the Boolean function Y = C + C'B + BA' when A = 1, B = 0 and C = 1

Y = 1 + 1'.0 + 0.1' = 1 + 0 + 0

Y = 1

(6) Simplify the given Boolean expression and implement using Basic and

NAND gates

F = AB + A(B+C) + B(B+C)

F = AB + AB + AC + BB + BC Distributive Law

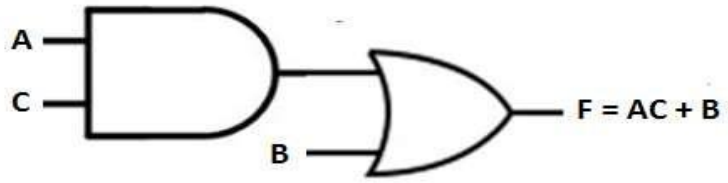
F = AB + AC + B + BC

F = AB + AC + B (1 + C) F = AB + AC+ B

F = AC + B(A + 1) F = AC + B

F = A.C + B

Using Basic Gates:

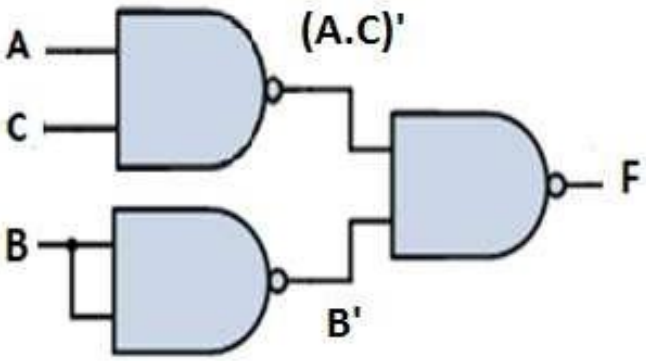


Using NAND Gates:

F = AC + B

F = ((AC + B)')'

F = ((AC)' . B')'



(7)Simplify and realize the given function using:

(i) Basic Gates:

Y = (A'.B + A' +

A.B)'

Y = (A'.B)' . (A')' .

(A.B)'

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

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

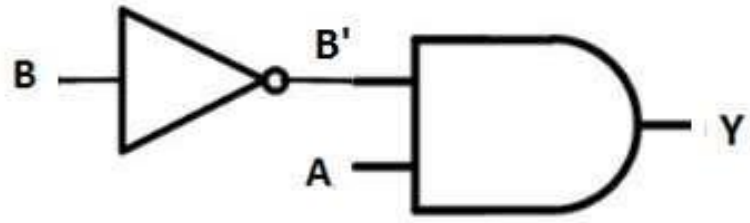
Y = A.A.B'+ A.B'.B'

Y = A.B'+ A.B'

Y = A.B'

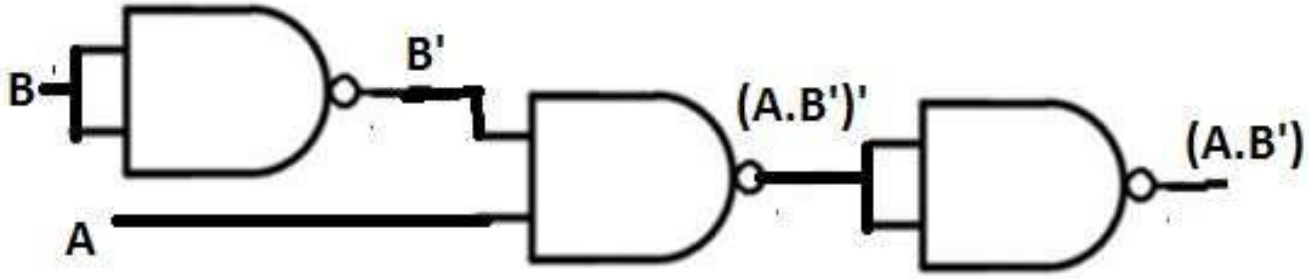
Y = A.B'

Using Basic Gates:



Using NAND Gates:

(8) Simplify the given expression



Y = AB’ + (A’ + B’+ C.C’)’

Y = AB’ + (A’ + B’)’

Y = AB’ + (A’’ B’’) Y = AB’ + (AB)

Y = A(B’+B)

Y = A

(9) Simplify the given expression

F = X’Y’Z+XYZ+X’YZ+XY’Z F = Y’Z(X+X’)+YZ(X’+X)

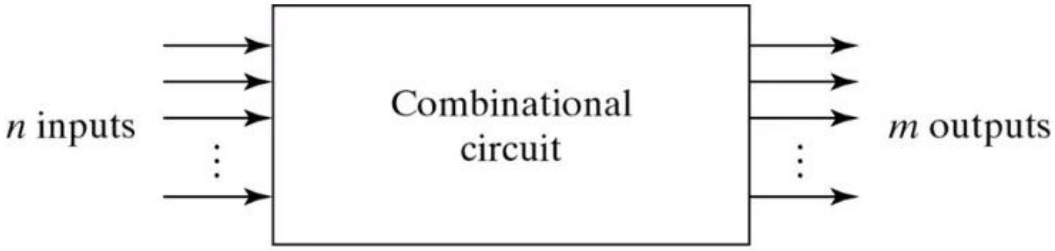
F = Y’Z+YZ

F = Z(Y’+Y)

Y = Z

Combinational Logic Circuits: Half Adder and Full adder

Combinational circuits are constructed by interconnection of logic gates.
whose outputs at any time are determined from only the present combination of inputs



- A combinational circuit performs an operation that can be specified logically by a set of **Boolean functions**
- Examples: Binary Adders, Multiplexers, etc.

Half Adder

A half adder is a combinational logic circuit that performs binary addition of two single-bit inputs, A and B, producing two outputs: **SUM** and **CARRY**. The SUM output which is the least significant bit (LSB) is obtained using an XOR gate while the CARRY output which is the most significant bit (MSB) is generated using an AND gate.
The half adder is a fundamental building block for more complex adder circuits, such as full adders and multi-bit adders. It allows the addition of two binary digits but does not account for carry-in from a previous stage. The circuit requires **o**ne XOR gate and one AND gate for implementation.

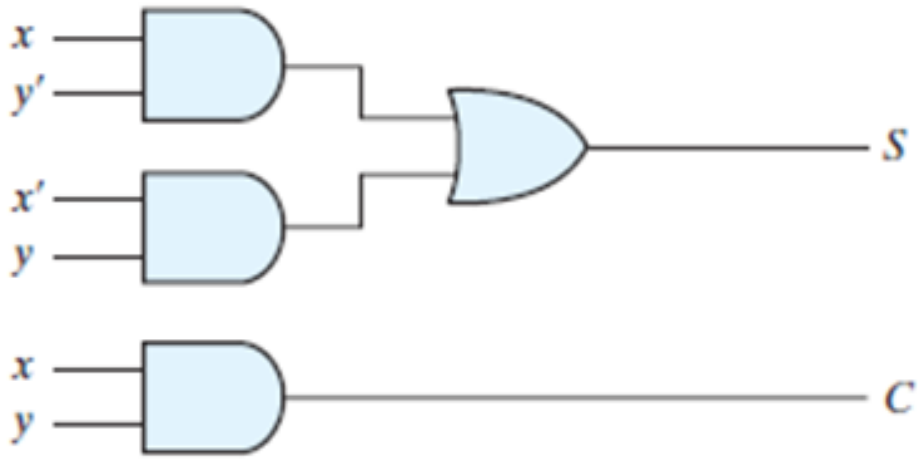
Truth Table of Half Adder

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Here from truth table we see that the sum is same as XOR and carry is same as AND truth table

sum = f(x, y) = ∑(1, 2) = x'.y + x.y'
carry = f(x, y) = ∑(3) = x.y

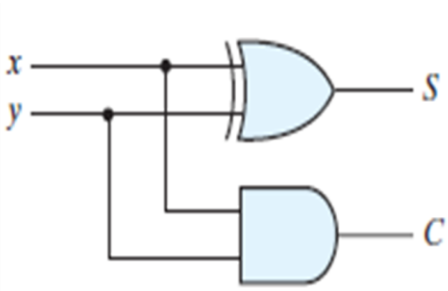
Logic Diagram



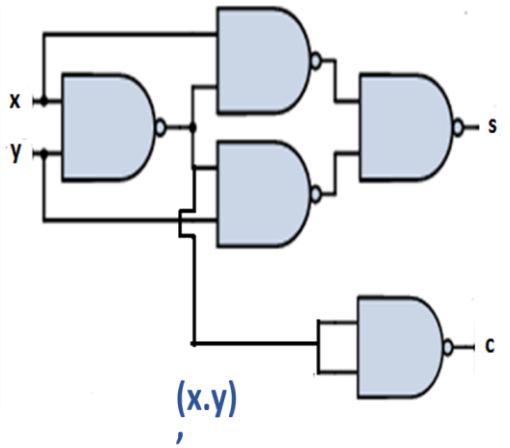
❖ **Half Adder:** $s = x \oplus y$ Sum expression: $s = x'.y + x.y'$

$c = x.y$

❖ **Half Adder using Logic Gates**



❖ **Half Adder using NAND Gates only**



(x.y)'

Full Adder:

- x, y and z are three binary inputs.
- S is sum and C is carry outputs

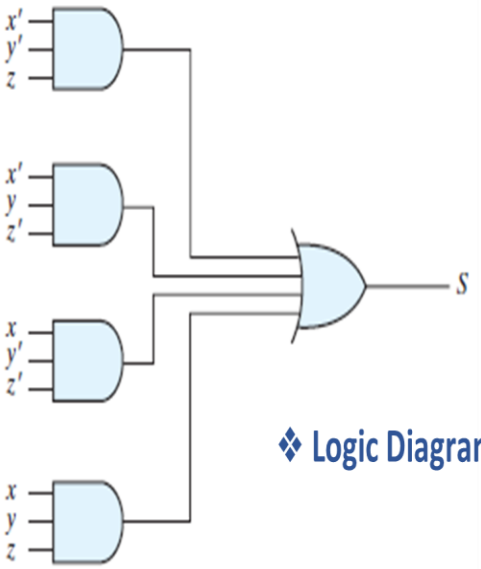
Truth Table:

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Boolean Expression:

$S = f(x, y, z) = \sum(1, 2, 4, 7) = x'y'z + x'yz' + xy'z' + xyz$

$C = f(x, y, z) = \sum(3, 5, 6, 7) = x'yz + xy'z + xyz' + xyz$



Logic Diagram

Full Adder:

- **Carry Expression:**

$C = x'yz + xy'z + xyz' + xyz$

$C = x'yz + xy'z + xy\,(z'+z)$

$C = x'yz + xy'z + xy$

$C = x'yz + x(y'z + y)$ Absorption Law

$C = x'yz + x(z + y)$

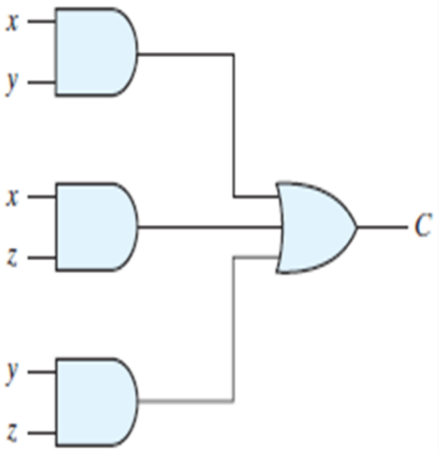
$C = x'yz + xz + xy$

$C = z(x'y + x) + xy$

$C = z(y + x) + xy$

$\mathbf{C = yz + xz + xy}$

Logic Diagram



❖ Full Adder:

➢ **Carry Expression:**

$C = x'yz + xy'z + xyz' + xyz$

$C = x'yz + xy'z + xy\,(z'+z)$

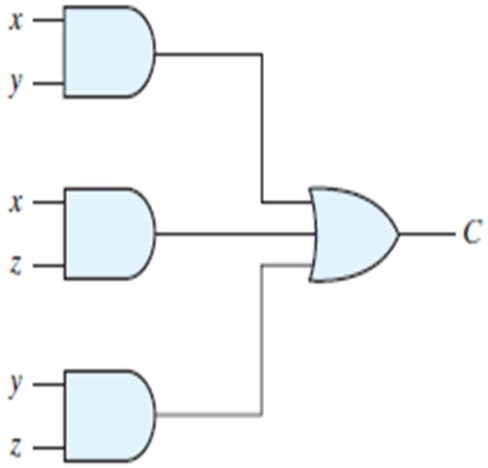
$C = x'yz + x(y'z + y)$ Absorption Law

$C = x'yz + xz + xy$

$C = z(x'y + x) + xy$

$C = z(y + x) + xy$

❖ Logic Diagram



Implement Full adder using two Half Adder

❖ **Boolean Expression for Sum:**

$S = x'y'z + x'yz' + xy'z' + xyz$

$S = x'(y'z + yz') + x(y'z' + yz)$

$S = x'(y \oplus z) + x((y \oplus z)')$

$S = x \oplus (y \oplus z)$

❖ **Boolean Expression for Carry:**

$C = x'yz + xy'z + xyz' + xyz$

$C = z.(x'y + xy') + xy(z' + z)$

$C = z.(x \oplus y) + xy.(1)$

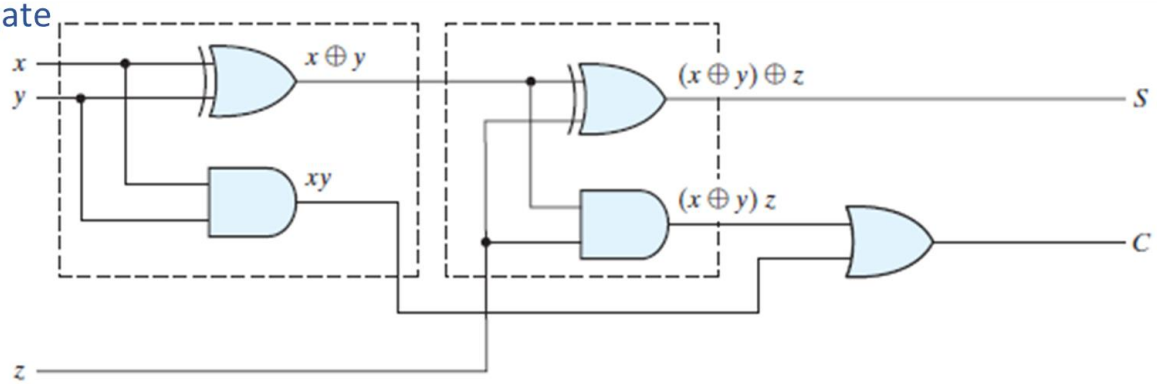
$C = (x \oplus y).z + xy$

❖ Full Adder Boolean Expression:

$S = (x \oplus y) \oplus z$

$C = (x \oplus y).z + x.y$

❖ Implementation of full adder with two half adders and an OR gate



Implement Full adder using NAND only

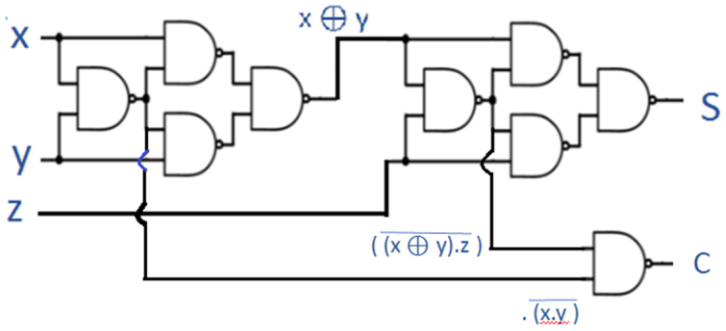
❖ Boolean Expression: $S = (x \oplus y) \oplus z$

$C = (x \oplus y).z + x.y$

❖ Full adder circuit using NAND Gates

$C = \overline{\overline{(x \oplus y).z} \cdot \overline{x.y}}$

$C = \overline{\overline{(x \oplus y).z} \cdot \overline{x.y}}$

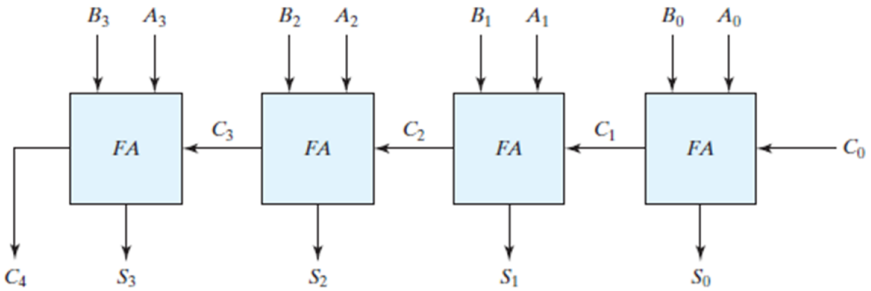


❖ **Four-bit adder**

Using four Full adder Ripple adder circuit is constructed.

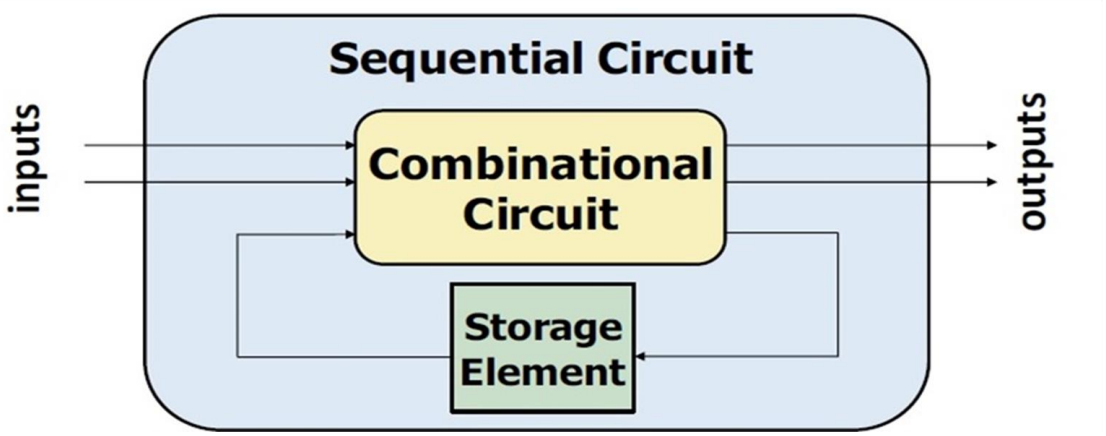
❖ **Example**

```
    1110
A = 1101
+ B = 0111
   10100
```



Introduction: Sequential Circuits

- Combinational Circuit output depend **only** on **present** input.
- We want circuits that produce the output depending on the **current** and **past** input values - Circuits with **memory**.
- How do we design such a circuit that **stores information**?

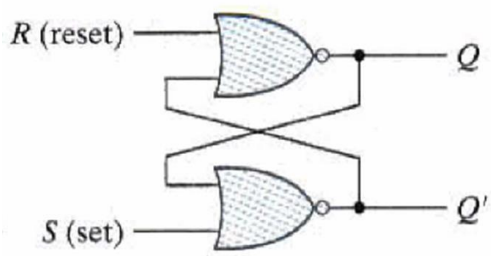


Latches vs. Flip-flops

- *Storage elements that operate with signal levels (rather than signal transitions) are referred to as latches ; those controlled by a clock transition are flip-flops .*
- Latches are said to be level sensitive devices; flip-flops are edge-sensitive devices.
- The two types of storage elements are related because latches are the basic circuits from which all flip-flops are constructed.
- Although latches are useful for storing binary information and for the design of asynchronous sequential circuits, they are not practical for use as storage elements in synchronous sequential circuits.

Types of sequential circuits – SR latch by Using NOR Gates

Circuit Diagram

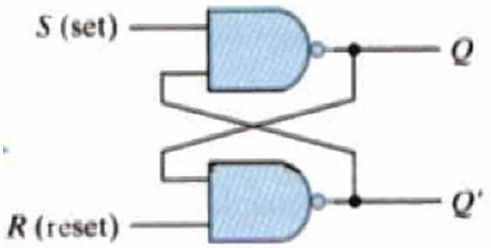


Function Table

S	R	Q	Q'	
1	0	1	0	
0	0	1	0	(after $S = 1, R = 0$)
0	1	0	1	
0	0	0	1	(after $S = 0, R = 1$)
1	1	0	0	(forbidden)

Types of sequential circuits – SR latch by Using NAND Gates

Circuit Diagram

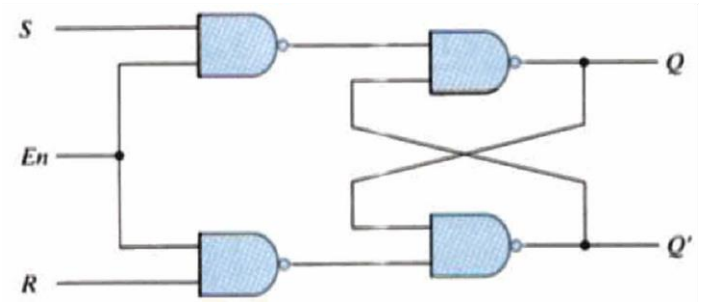


Function Table

S	R	Q	Q'	
1	0	0	1	
1	1	0	1	(after $S = 1, R = 0$)
0	1	1	0	
1	1	1	0	(after $S = 0, R = 1$)
0	0	1	1	(forbidden)

The S-R (Set - Reset) latch with control input

Logic Diagram

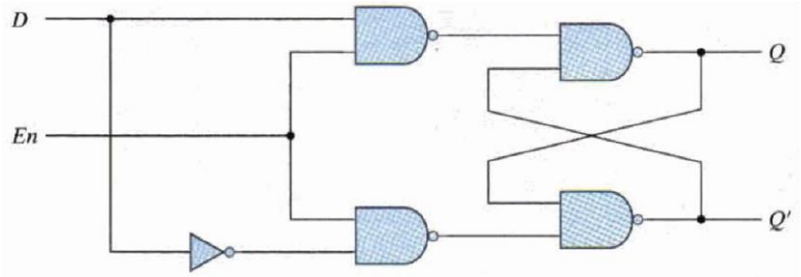


Function Table

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

The D latch (Transparent latch)

Logic Diagram



Function Table

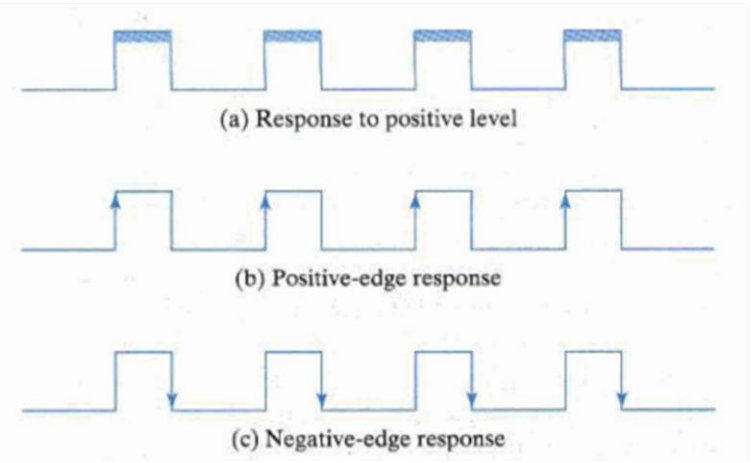
En	D	Next state of Q
0	X	No change
1	0	$Q = 0$; reset state
1	1	$Q = 1$; set state

Flip-Flops

- The state of a latch or flip-flop is switched by a change in the control input.
- This momentary change is called a *trigger*, and the transition it causes is said to trigger the flip-flop.
- The *D* latch with pulses in its control input is essentially a flip-flop that is triggered every time the pulse goes to the logic-1 level.
- As long as the pulse input remains at this level, any changes in the data input will change the output and the state of the latch.

- Flip-flop circuits are constructed in such a way as to make them operate properly when they are part of a sequential circuit that employs a common clock.
- The problem with the latch is that it responds to a change in the *level* of a clock pulse.
- As shown in Fig. (a), a positive level response in the enable input allows changes in the output when the *D* input changes while the clock pulse stays at logic 1.
- The key to the proper operation of a flip-flop is to trigger it only during a signal *transition* .

Clock Response in Latch and Flip – Flop:



Latch will change the Output when the clock remains as logic 1 as shown in figure (a)

Flip-Flops will change the Output when the clock changes from logic 0 to Logic 1 as shown in figure (b)

Edge Triggered D Flip – Flop:

- Construction of an edge-triggered *D* flip-flop uses three *SR* latches as shown in Fig.
- Two latches respond to the external *D* (data) and *Clk* (clock) inputs. The third latch provides the outputs for the flip-flop.
- The *S* and *R* inputs of the output latch are maintained at the logic-1 level when *Clk* = 0.
- This causes the output to remain in its present state. Input *D* may be equal to 0 or 1.
- If *D* = 0 when *Clk* becomes 1, *R* changes to 0. This causes the flip-flop to go to the reset state, making *Q* = 0. If there is a change in the *D* input while *Clk* = 1, terminal *R* remains at 0 because *Q* is 0.
- Thus, the flip-flop is locked out and is unresponsive to further changes in the input.

- Similarly, if *D* = 1 when *Clk* goes from 0 to 1, *S* changes to 0.
- This causes the circuit to go to the set state, making *Q* = 1.
- Any change in *D* while *Clk* = 1 does not affect the output.

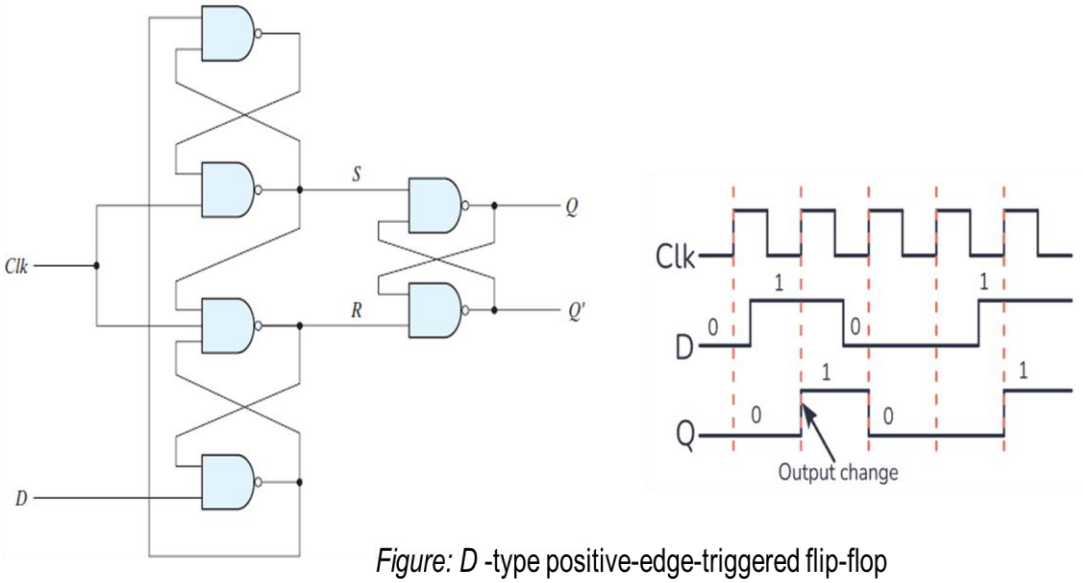
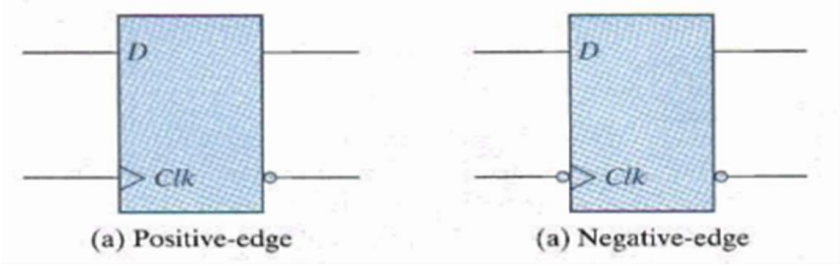


Figure: D -type positive-edge-triggered flip-flop

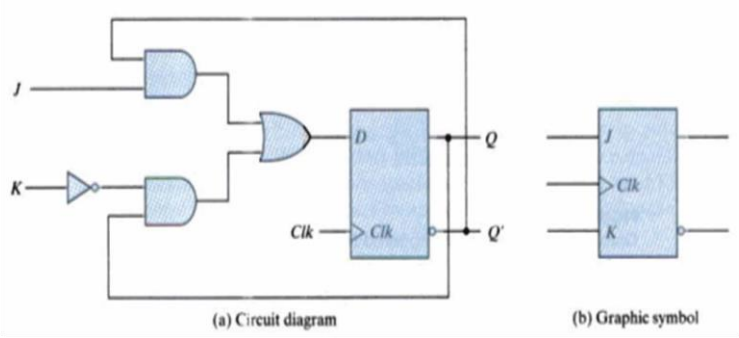
Positive & Negative Edge Triggered 'D' Flip – Flop:



In D Flip-Flops, Absence of the bubble in front of the Clk indicates Positive edge triggered as shown in figure (a)

In D Flip-Flops, Presence of the bubble in front of the Clk indicates Positive edge triggered as shown in figure (b)

JK Flip – Flop using D flip flop



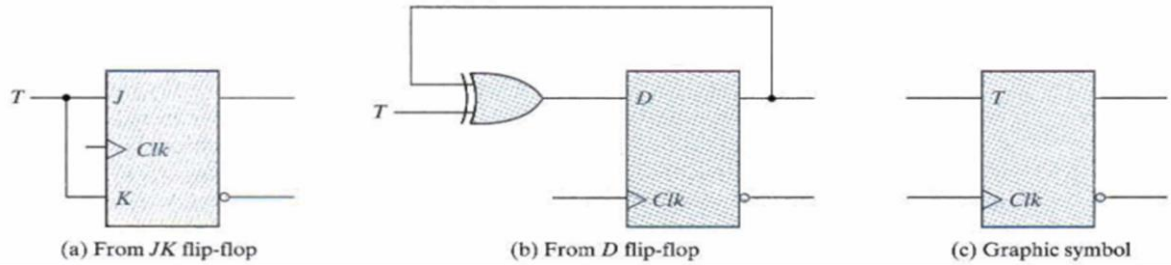
Flip-Flop Characteristic Tables

JK Flip-Flop

J	K	Q(t + 1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	Set
1	1	Q'(t)	Complement

- The Inputs are J-K along with the Clk
  - J-K inputs comes after its inventors Jack Kilby.

T Flip – Flop:



T Flip-Flop using JK Flip-Flop & D Flip-Flop

- T Flip-Flop is a **synchronous** device, where high to low or low to high transitions is passed through **clock signal** which **changes** the output state of Flip-Flop.

T Flip-Flop

T	Q(t + 1)	
0	Q(t)	No change
1	Q′(t)	Complement

Advantages of T flip-flop:

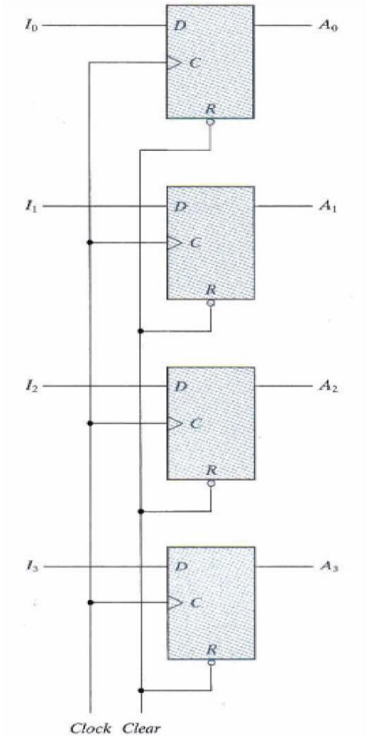
- These Flip-Flops has a **toggle input** and a **clock**.
- They are used for designing the **counters**.

Registers:

- A *register* is a group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information.
- An *n* -bit register consists of a group of *n* flip-flops capable of storing *n* bits of binary information.
- In addition to the flip-flops, a register may have combinational gates that perform certain data-processing tasks.
- In its broadest definition, a register consists of a group of flip-flops together with gates that affect their operation.
- The flip-flops hold the binary information, and the gates determine how the information is transferred into the register.

4-bit Register:

- Various types of registers are available commercially.
- The simplest register is one that consists of only flip-flops, without any gates.
- Following Figure (next slide) shows such a register constructed with four *D* -type flip-flops to form a four-bit data storage register.
- The common clock input triggers all flip-flops on the positive edge of each pulse, and the binary data available at the four inputs are transferred into the register.
- The four outputs can be sampled at any time to obtain the binary information stored in the register.
- The input Clear goes to the active-low R (reset) input of all four flip-flops. When this input goes to 0, all flip-flops are reset asynchronously.
- The R inputs must be maintained at logic 1 (i.e., de-asserted) during normal clocked operation.



- The transfer of new information into a register is referred to as *loading* or *updating* the register.
- If all the bits of the register are loaded simultaneously with a common clock pulse, we say that the loading is done *in parallel* .
- A clock edge applied to the *C* inputs of the register of Figure, will load all four inputs in parallel.
- In this configuration, if the contents of the register must be left unchanged, the inputs must be held constant or the clock must be inhibited from the circuit.

SHIFT REGISTERS (SISO)

- A register capable of shifting the binary information held in each cell to its neighboring cell, in a selected direction, is called a *shift register*.
- The logical configuration of a shift register consists of a chain of flip-flops in cascade, with the output of one flip-flop connected to the input of the next flip-flop.
- All flip-flops receive common clock pulses, which activate the shift of data from one stage to the next.
- There are four different types of shift registers:
  - Serial In Serial Out (SISO)
  - Serial In Parallel Out (SIPO)
  - Parallel In Serial Out (PISO)
  - Parallel In Parallel Out (PIPO)

- The simplest possible shift register (SISO) is one that uses only flip-flops, as shown in the following Figure .
- The output of a given flip-flop is connected to the *D* input of the flip-flop at its right.
- This shift register is unidirectional (left-to-right).
- Each clock pulse shifts the contents of the register one bit position to the right.

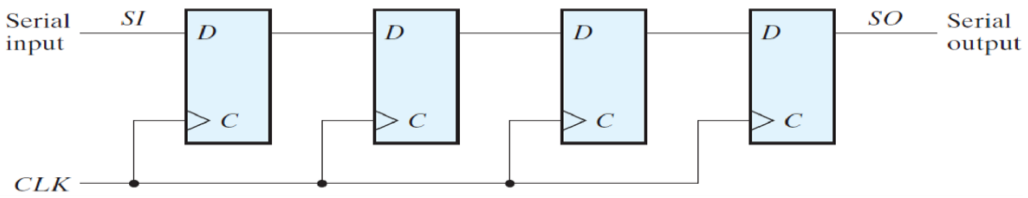
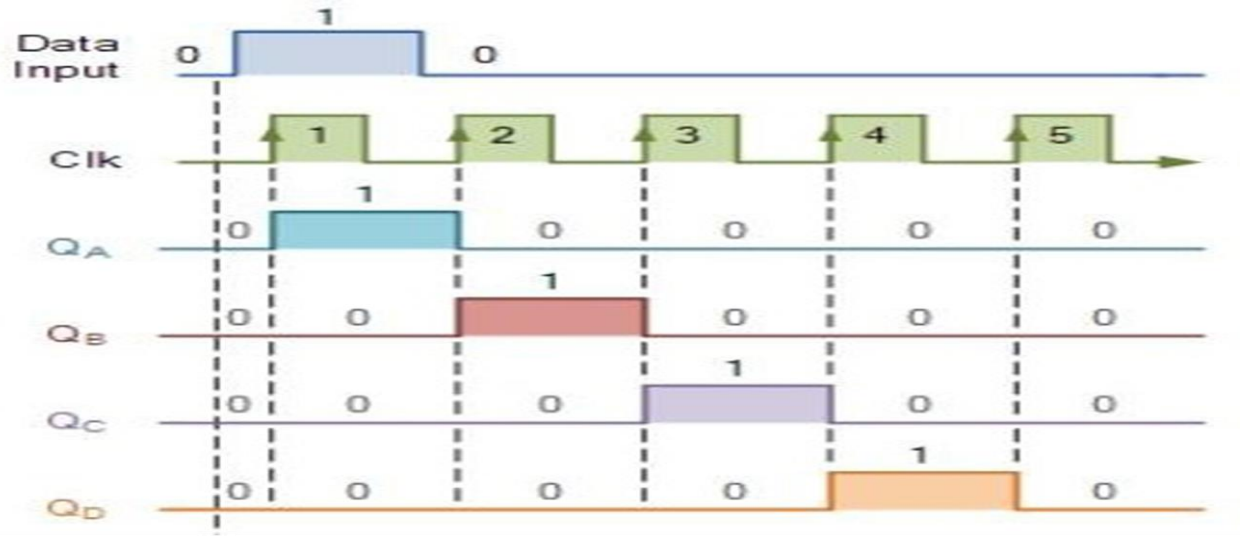


FIGURE: Four-bit shift register



Timing Diagram for SISO register

Clock Pulse No	QA	QB	QC	QD
0	0	0	0	0
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	0	0	0	0

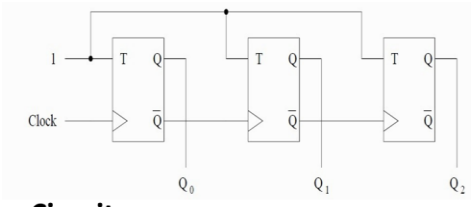
Data movement through a shift register

Synchronous and Asynchronous counters (Introduction):

- A register (group of flip-flops) that goes through a prescribed sequence of states upon the application of input pulses is called a *counter* .
- The input pulses may be clock pulses, or they may originate from some external source and may occur at a fixed interval of time or at random.
- 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: ripple (or asynchronous) counters and synchronous counters.
- In a ripple counter, a flip-flop output transition serves as a source for triggering other flip-flops.
- In other words, the *C* input of some or all flip-flops are triggered, not by the common clock pulses, but rather by the transition that occurs in other flip-flop outputs.
- In a synchronous counter, the *C* inputs of all flip-flops receive the common clock.

3 bit Asynchronous up- counter



Circuit diagram

Number of clock pulses	Q_2 (MSB)	Q_1	Q_0 (LSB)
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	1	0	0
5	1	0	1
6	1	1	0
7	1	1	1

Timing Diagram

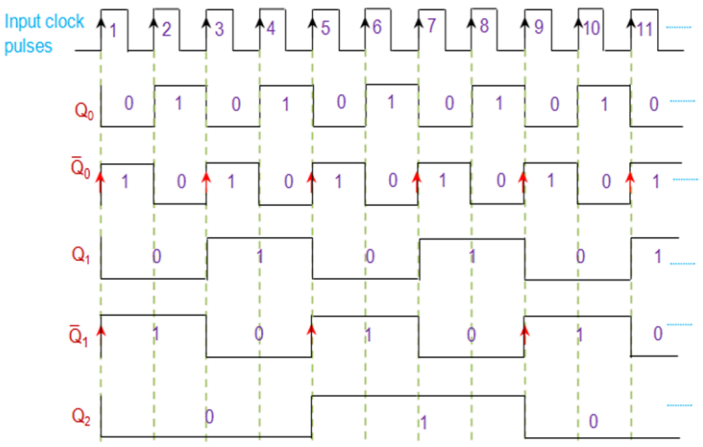
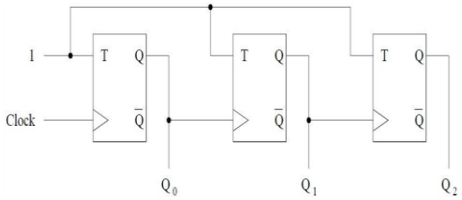
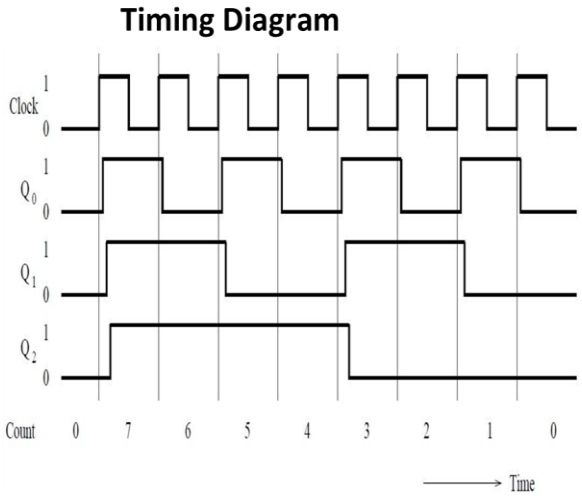


Figure 4 Timing diagram for 3-bit asynchronous positive edge triggered up-counter

3 bit Asynchronous down - counter:





Truth Table

Number of clock pulses	Q_2 (MSB)	Q_1	Q_0 (LSB)
0	0	0	0
1	1	1	1
2	1	1	0
3	1	0	1
4	1	0	0
5	0	1	1
6	0	1	0
7	0	0	1