

# **ABES ENGINEERING COLLEGE, GHAZIABAD**

**Affiliated To Dr. APJAKTU, Lucknow**

**Department of Computer Science**



**Lab Manual Odd SEM**

**Session 2022-2023**

**Subject Name : Computer Organization Lab**

**Subject Code :BCS-352**

**Semester /Section :IIIrd SEM**

**Faculty Name :**



**ABES ENGINEERING COLLEGE, GHAZIABAD**  
**Department of Computer Science**

**Lab Delivery Schedule**

Program	Sem	Course Name	Course Code	Periods (University)
B.Tech (CS))	III	Computer Organization Lab	BCS352	2

Periods (Actual)	Name of Faculty	Vertical Head	Date of Commencement	Total lab planned	Date of Conclusion
2	Laxmi Saraswat	Dr. Pankaj Kumar Sharma	11/09/2023	12	

S. No	Title of Experiment	KL	CO	Software Used	Page No
1	Introduction to Simulator. Implement different logic gates and verify the truth table.	K3	CO1, CO2	Logisim/Simulator IIT Kharagpur	3 9
2	Implement different logic gates using universal gates.	K3	CO2	Logisim/Simulator IIT Kharagpur	13
3	Implement half adder using basic logic and universal gates. Full adder using basic logic, universal gates and using 2 half adders.	K3	CO2	Logisim/Simulator IIT Kharagpur	18
4	Implementing Binary -to -Gray, Gray -to -Binary code conversions.	K3	CO3	Logisim/Simulator IIT Kharagpur	20
5	Implementing 3–8-line DECODER	K3	CO3	Logisim/Simulator IIT Kharagpur	24
6	Implement 8x1 MULTIPLEXERS	K3	CO3	Logisim/Simulator IIT Kharagpur	28
7	Verify the excitation tables of various SR and T FLIP-FLOPS.	K3	CO3	Logisim/Simulator IIT Kharagpur	31
8	Verify the excitation tables of various JK and D FLIP-FLOPS.	K3	CO3	Logisim/Simulator IIT Kharagpur	35
9	Design a 4bit Universal Shift Register	K4	CO4	Logisim/Simulator IIT Kharagpur	38
10	Design of a 4-bit ARITHMETIC LOGIC UNIT.	K4	CO4	Logisim/Simulator IIT Kharagpur	41
<b>Beyond The Syllabus</b>					
11	Design and implement half and full subtractor	K3	CO1	Logisim/Simulator IIT Kharagpur	44
12	Design and implement Binary to BCD and BCD to Binary converter	K3	CO1	Logisim/Simulator IIT Kharagpur	47

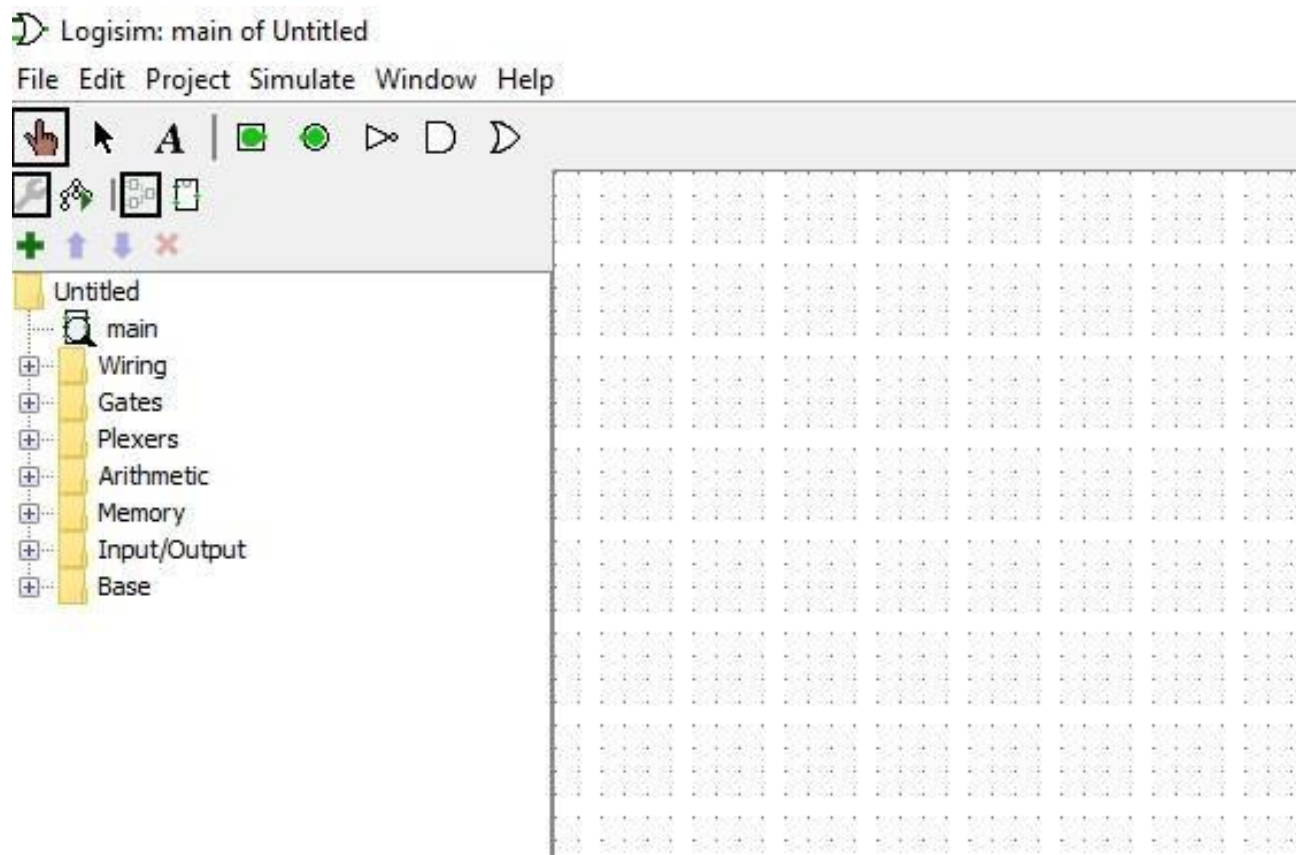
Signature of Faculty

Signature of HoD

## Introduction to Logisim Simulator

When learning computer architecture and logic circuits, you will need a real-world, graphical example of what you are studying. Text and diagrams only go so far. A helpful tool for designing and simulating logic circuits is **Logisim**.

Double click **logisim-win-2.7.1.exe** on your desktop to open Logisim.  
You would see the following screen:



You are at the **main** circuit window by default. If you select an AND gate for example, it has five inputs by default. But you can change the Number of Inputs. Please also remember that the Help menu contains very helpful information such as Tutorial, User's Guide, and Library Reference.

The screenshot shows the Logisim software interface. At the top is the menu bar: File, Edit, Project, Simulate, Window, and Help. Below the menu bar is a toolbar with various icons. A red box highlights the 'Input' and 'Output' icons in the toolbar. To the left is a component list with categories: main, Wiring, Gates, Plexers, Arithmetic, Memory, Input/Output, and Base. The 'Gates' category is selected. In the center is a large grid workspace where a D-shaped component is placed. At the bottom is a configuration table for the selected AND gate.

**Selection: AND Gate**

Facing	East
Data Bits	1
Gate Size	Medium
Number Of Inputs	5
Output Value	0/1
Label	
Label Font	SansSerif Plain 12
Negate 1 (Top)	No
Negate 2	No
Negate 3	No

Annotations in the image include:

- A red box around the 'Help' menu item.
- A red box around the 'Gates' category in the component list.
- Red arrows pointing from the 'Input' and 'Output' icons in the toolbar to the labels 'Input' and 'Output'.
- A red arrow pointing from the 'Gates' category to the text 'Select gates from'.
- Red arrows pointing from the 'Data Bits' and 'Number Of Inputs' fields in the configuration table to the text 'You can select the Data-Bits and Number of inputs you need'.

You can select gates and input/output devices and make connections to make circuits. You can save your project from the **File** menu and select **Save As, test1** here for example. You can also add new circuits to the project **test1** by clicking on the PLUS Sign +. Double click on a circuit name will open it. One click on a circuit name will have the device symbol stick on your mouse so that you could put it somewhere in the circuit window as a device by a click.

Logisim: main of test1

File Edit Project Simulate Window Help

From File menu select Save As, save your project as test1;

You can add more circuits to the project by clicking on the +

Double clicking on each circuit will open it. One click on the file, you will get the device icon for that circuit. You can use one circuit in another circuit.

test1\*

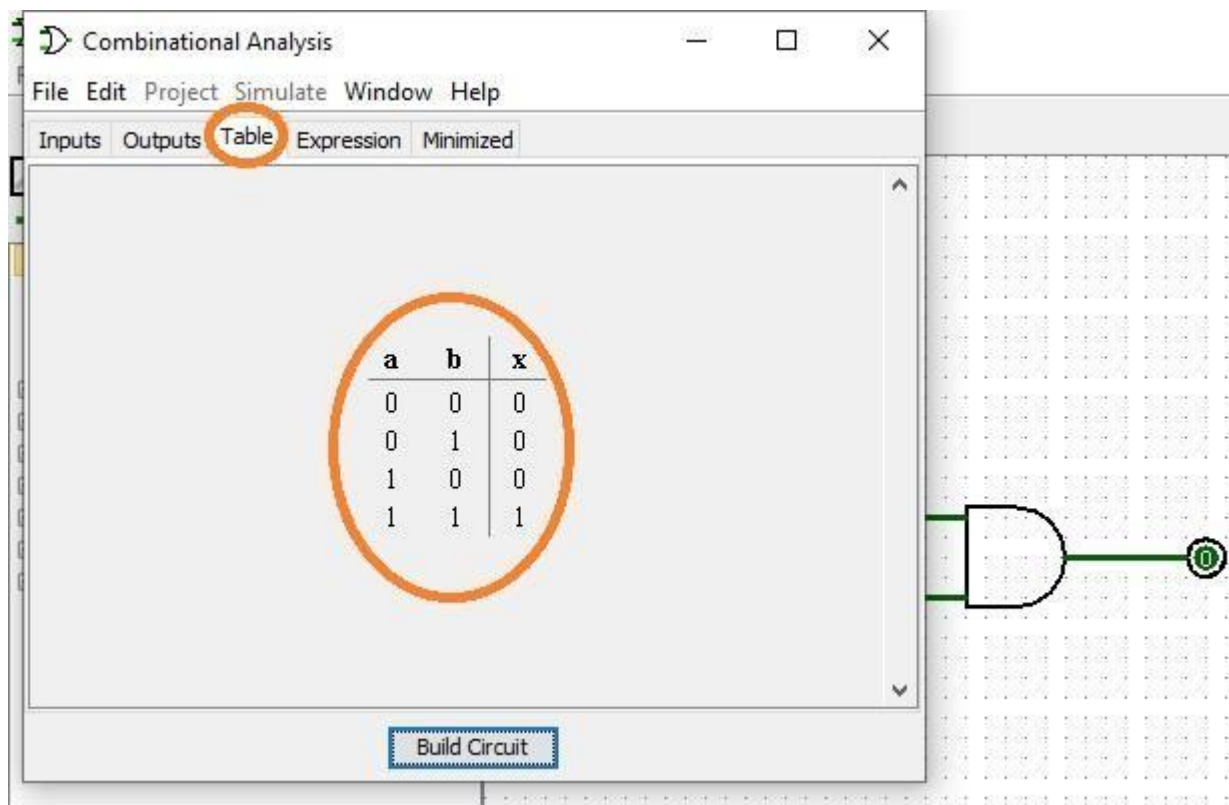
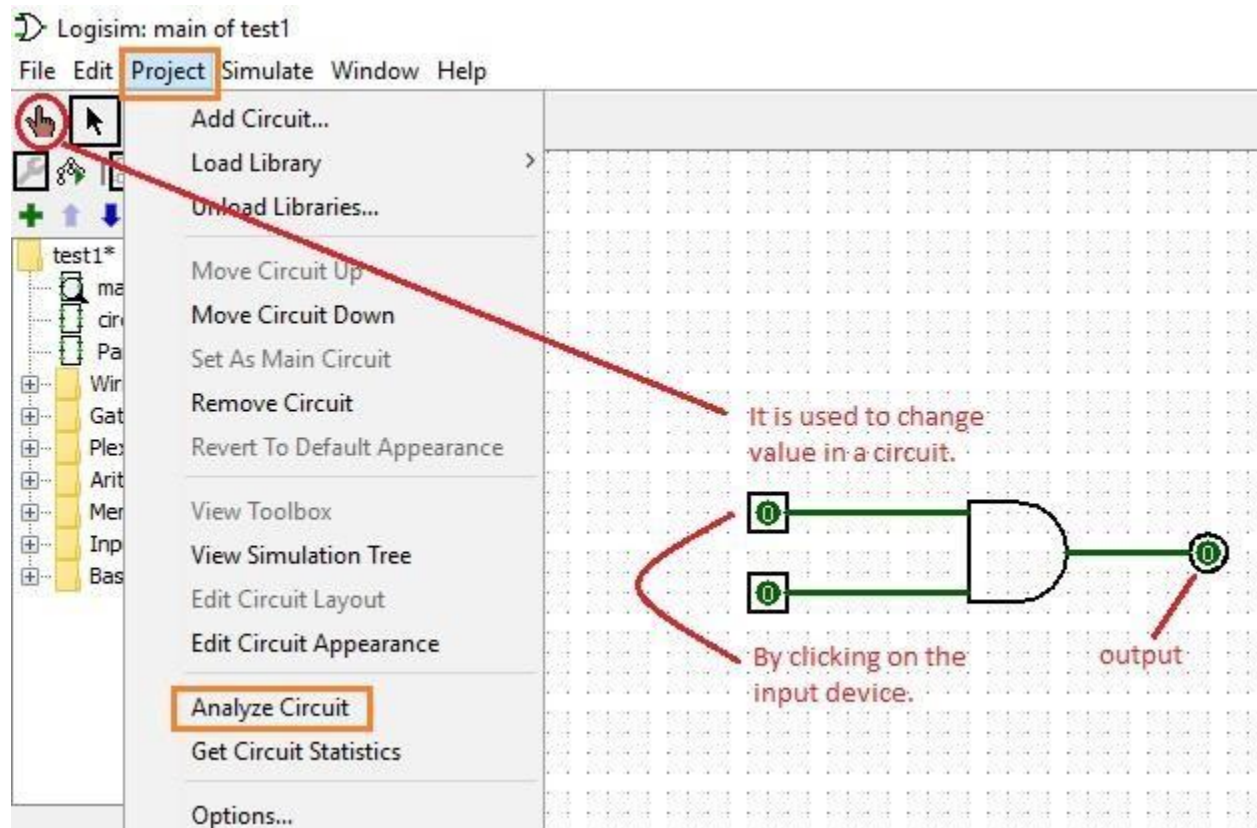
- main
- circuit1
- Part2
- Wiring
- Gates
- Plexers
- Arithmetic
- Memory
- Input/Output
- Base

Circuit: main

Circuit Name	main
Shared Label	
Shared Label Facing	East
Shared Label Font	SansSerif Plain 12

When you get a circuit done, you can test it by changing the input values and see the output values. You can also see the auto generated truth table from the **Project** menu and select **Analyse Circuit** and then select **Table**.





# Introduction to IIT Kharagpur Simulator

This simulator provides an interactive environment for creating and conducting simulated experiments on computer organization and architecture. It supports gate level design to CPU design.

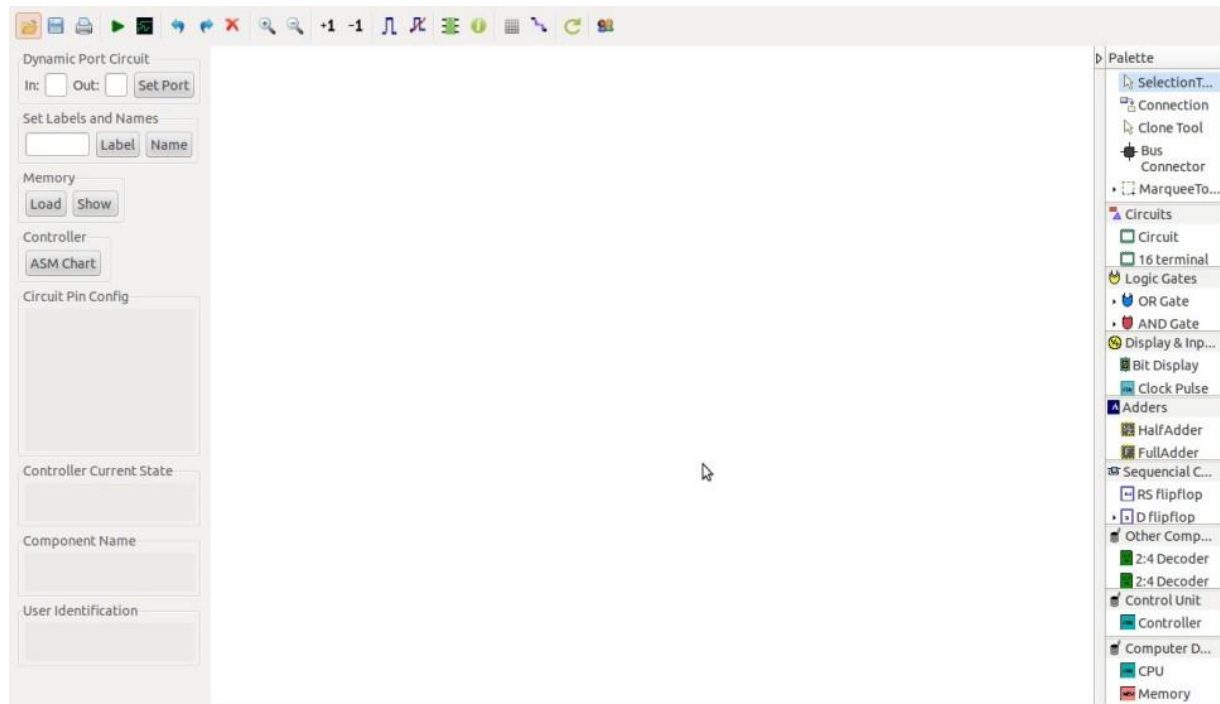


Figure 1. Main interface of the simulator

Features of the simulator: The main features of the simulators are as follows: Logic: The simulator supports 5 valued logics. So, the simulator supports wired AND for bus based design. These 5 states along with their corresponding wire values are as follows:

- True (T) (wire color: blue)
- False (F) (wire color: black)
- High impedance (Z) (wire color: green)
- Unknown (X) (wire color: maroon)
- Invalid (I) (wire color: orange)

The simulator contains

- a palette on the right hand side. This palette contains all the components and tools . Tools are used to act up on the components.
- a toolbar on the top which contains several buttons. These buttons are:
  - *save/open*
  - *simulate* (after creating a circuit, this button has to be pressed to simulate the circuit and to get output)
  - *plot graph* (to plot input-output wave form)
  - *undo/redo*
  - *delete*
  - *zoom in/zoom out*
  - *increment/decrement LED* (for digital LED which can also be used as input and display)
  - *start/stop clock pulse*
  - *to check the name or pin configuration of a component*
  - *changing connection types*
  - *checking the user identification*
- a canvas in the middle where the circuits will be designed.

- 
- A toolbar on the left side which contains the following buttons:
    - *set port* to set the number of input and output ports for a circuit.
    - *Set Label* and *set name* to set the label contents and the name of different components.
    - *load memory* to load the memory content to the inbuilt memory(4 bit address and 12 bit data) for performing the computer design experiment. Data can be load either from file or through form.
    - *Show memory* to show the content of the in built memory.
    - *ASM chart* to load the ASM (algorithmic state machine) chart for a controller.

### **Tools:**

- Different tools are:
    - **Selection tool**- used for selecting components
    - **Marquee tool**- used for selecting many components at a time by dragging the mouse in the design area(editor).
    - **Connection tool**- used for connecting components.
    - **Clonning tool** – used to create cloned components.
-



## Experiment 1

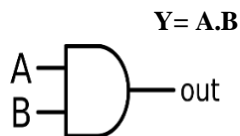
**AIM:** Implement different logic gates and verify the truth table.

### **THEORY:**

Introduction: Logic gates are the basic building blocks of any digital system. Logic gates are electronic circuits having one or more than one input and only one output. The relationship between the input and the output is based on a certain logic. Based on this, logic gates are named as

- 1) AND gate
- 2) OR gate
- 3) NOT gate
- 4) NAND gate
- 5) NOR gate
- 6) Ex-OR gate
- 7) Ex-NOR gate

1) AND gate: The AND gate is an electronic circuit that gives a high output (1) only if all its inputs are high. A dot (.) is used to show the AND operation i.e.  $A.B$  or can be written as  $AB$



**Figure-1: Logic Symbol of AND Gate**

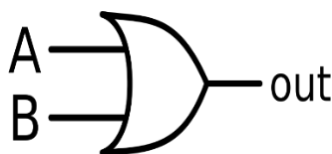
Input		Output
A	B	$Y = A.B$
0	0	0
0	1	0
1	0	0
1	1	1

**Figure-2: Truth Table of AND Gate**

2) OR gate

The OR gate is an electronic circuit that gives a high output (1) if one or more of its inputs are high. A plus (+) is used to show the OR operation.

$$Y = A+B$$



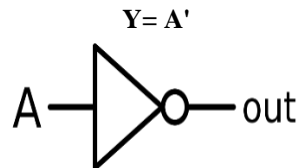
**Figure-4: Logic Symbol of OR Gate**

Input		Output
A	B	$Y=A+B$
0	0	0
0	1	1
1	0	1
1	1	1

**Figure-5: Truth Table of OR Gate**

### 3) NOT gate

The NOT gate is an electronic circuit that produces an inverted version of the input at its output. It is also known as an inverter. If the input variable is A, the inverted output is known as NOT A. This is also shown as A' or A with a bar over the top, as shown at the outputs.



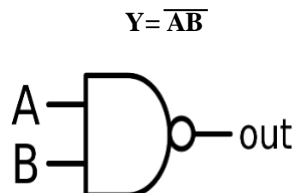
**Figure-7: Logic Symbol of NOT Gate**

Input	Output
A	Y
0	1
1	0

**Figure-8: Truth Table of NOT Gate**

### 4) NAND gate

This is a NOT-AND gate which is equal to an AND gate followed by a NOT gate. The outputs of all NAND gates are high if any of the inputs are low. The symbol is an AND gate with a small circle on the output. The small circle represents inversion.



**Figure-10: Logic Symbol of NAND Gate**

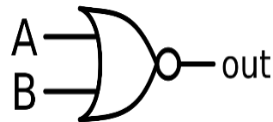
Input	Input	Output
A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

**Figure-11: Truth Table of NAND Gate**

#### 5) NOR gate

This is a NOT-OR gate which is equal to an OR gate followed by a NOT gate. The outputs of all NOR gates are low if any of the inputs are high. The symbol is an OR gate with a small circle on the output. The small circle represents inversion.

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



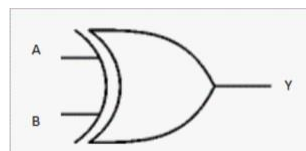
**Figure-13: Logic Symbol of NOR gate**

A	B	F
0	0	1
0	1	0
1	0	0
1	1	0

**Figure-14: Truth Table of NOR gate**

6) Ex-OR gate: The 'Exclusive-OR' gate is a circuit which will give a high output if either, but not both of its two inputs are high. An encircled plus sign ( $\oplus$ ) is used to show the Ex-OR operation.

$$Y = A \oplus B$$



**Figure-16: Logic Symbol of Ex-OR gate**

A	B	A <b>XOR</b> B
0	0	0
0	1	1
1	0	1
1	1	0

**Figure-17: Truth Table of Ex-OR gate**

Ex-OR gate is created from AND, NAND and OR gates. The output is high only when both the inputs are different

#### 7) Ex-NOR gate

The 'Exclusive-NOR' gate circuit does the opposite to the EX-OR gate. It will give a low output if either, but not both of its two inputs are high. The symbol is an EX-OR gate with a small circle on the output. The small circle represents inversion.

$$Y = \overline{A \oplus B}$$



**Figure-19: Logic Symbol of Ex-NOR gate**

XNOR Truth Table		
A	B	Q
0	0	1
0	1	0
1	0	0
1	1	1

**Figure-20: Truth Table of Ex-NOR gate**

Ex-NOR gate is created from AND, NOT and OR gates. The output is high only when both the inputs are same.

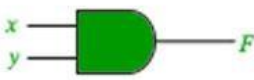







**RESULT:** Implementation of logic gates and verification of truth table are done successfully.

## Experiment-2

**AIM:** Implement different logic gates using universal gates.

### **THEORY:**

In Boolean Algebra, the **NAND** and **NOR** gates are called **universal gates** because any digital circuit can be implemented by using any one of these two i.e. any logic gate can be created using NAND or NOR gates only.

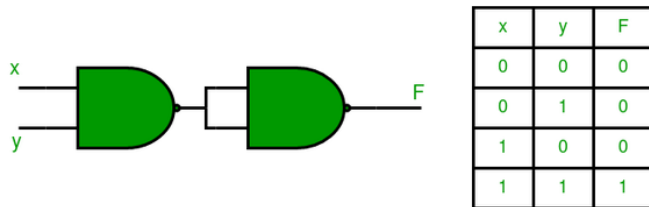
Name	Graphic symbol	Algebraic function	Truth table															
AND		$F = x \cdot y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	0																
0	1	0																
1	0	0																
1	1	1																
OR		$F = x + y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	1
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	1																
Inverter		$F = x'$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table>	x	F	0	1	1	0									
x	F																	
0	1																	
1	0																	
Buffer		$F = x$	<table><tr><th>x</th><th>F</th></tr><tr><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td></tr></table>	x	F	0	0	1	1									
x	F																	
0	0																	
1	1																	
NAND		$F = (xy)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	1																
0	1	1																
1	0	1																
1	1	0																
NOR		$F = (x + y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	0
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	0																
Exclusive-OR (XOR)		$F = xy' + x'y$ $= x \oplus y$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td></tr></table>	x	y	F	0	0	0	0	1	1	1	0	1	1	1	0
x	y	F																
0	0	0																
0	1	1																
1	0	1																
1	1	0																
Exclusive-NOR or equivalence		$F = xy + x'y'$ $= (x \oplus y)'$	<table><tr><th>x</th><th>y</th><th>F</th></tr><tr><td>0</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table>	x	y	F	0	0	1	0	1	0	1	0	0	1	1	1
x	y	F																
0	0	1																
0	1	0																
1	0	0																
1	1	1																



## 1. Implementation of AND Gate using Universal gates.

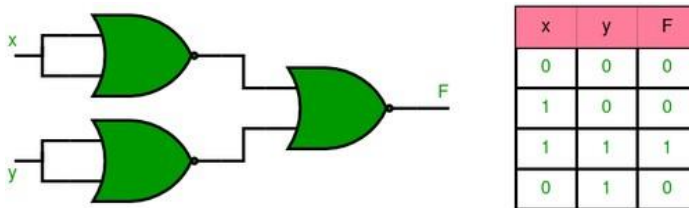
### a) Using NAND Gates

The AND gate can be implemented by using two NAND gates in the below fashion:



### b) Using NOR Gates

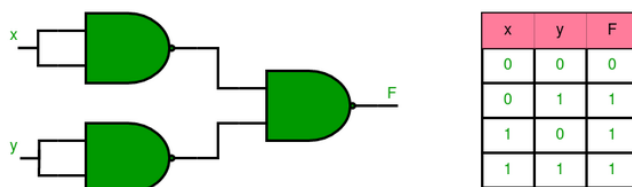
Implementation of AND gate using only NOR gates as shown below:



## 2. Implementation of OR Gate using Universal gates.

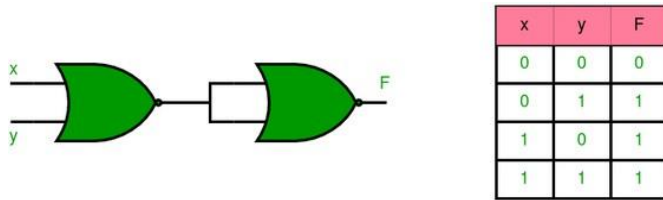
### a) Using NAND Gates

The OR gate can be implemented using the NAND gate as below:



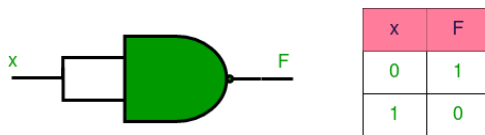
b) Using NOR Gates

Implementation of OR gate using two NOR gates as shown in the picture below:

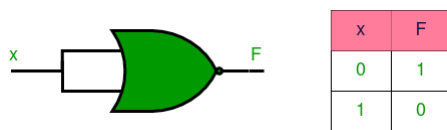


3. Implementation of NOT Gate using Universal gates.

a) Using NAND Gates

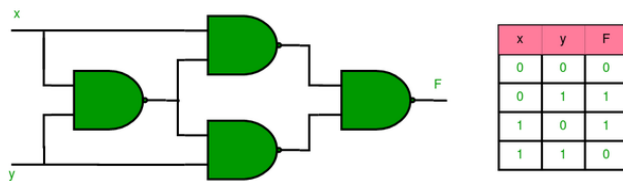


b) Using NOR Gates

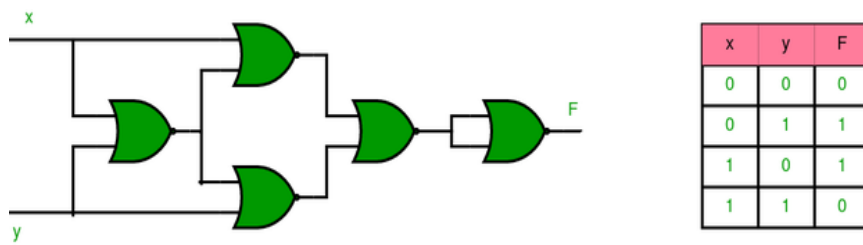


4. Implementation of XOR Gate using Universal gates.

a) Using NAND Gates

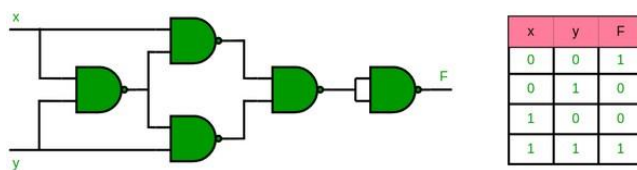


b) Using NOR Gates

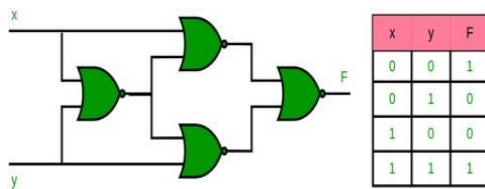


5. Implementation of XNOR Gate using Universal gates.

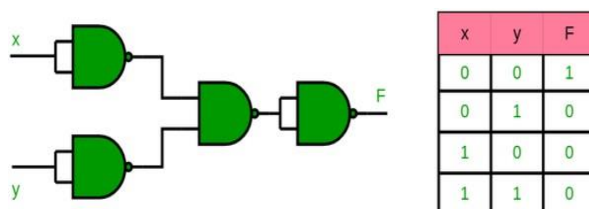
a) Using NAND Gate



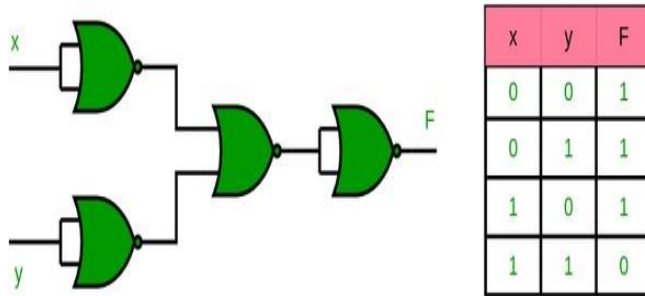
b) Using NOR Gate



6. Implementation of NOR Gate using NAND Gates



### 7. Implementation of NAND Gate using NOR Gates



**RESULT:** Implementation of logic gates using universal gates are done successfully.

### Experiment-3

**AIM:** Implementing HALF ADDER, FULL ADDER using basic logic gates

#### **THEORY:**

The most basic arithmetic operation is the addition of two binary digits. There are four possible elementary operations, namely,

$$0 + 0 = 0$$

$$0 + 1 = 1$$

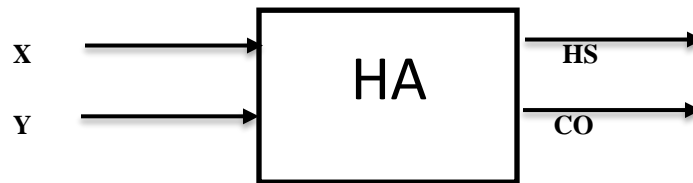
$$1 + 0 = 1$$

$$1 + 1 = 0 \text{ (with 1 as carry)}$$

The first three operations produce a sum of whose length is one digit, but when the last operation is performed the sum is two digits. The higher significant bit of this result is called a carry and lower significant bit is called the sum.

**Half Adder:** A combinational circuit which performs the addition of two bits is called half adder. The input variables designate the augend and the addend bit, whereas the output variables produce the sum and carry bits.

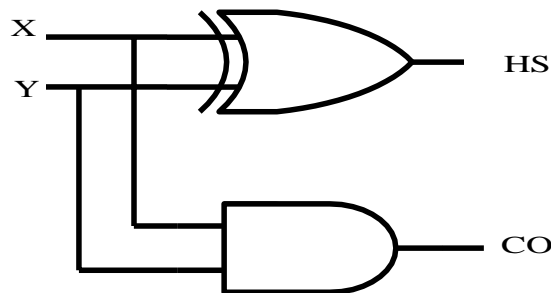
#### **Block Diagram:**



#### **Truth Table:**

S.No	Input		Output	
	X	Y	HS	CO
1	0	0	0	0
2	0	1	1	0
3	1	0	1	0
4	1	1	0	1

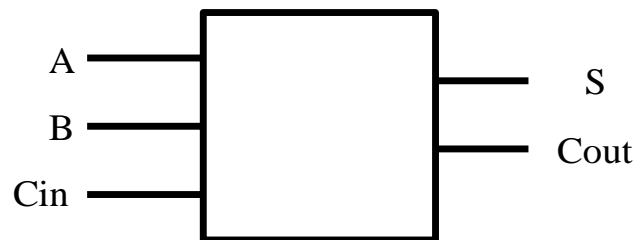
#### **Logic Diagram:**





**Full Adder:** A combinational circuit which performs the arithmetic sum of three input bits is called full adder. The three input bits include two significant bits and a previous carry bit. A full adder circuit can be implemented with two half adders and one OR gate.

**Block Diagram:**



**Truth Table:**

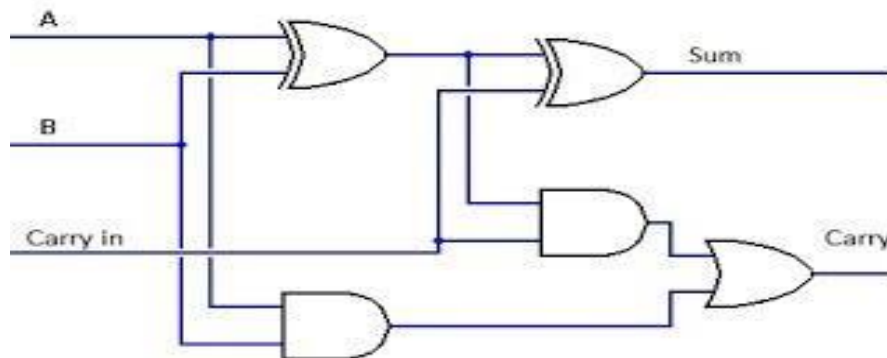
Inputs			Outputs	
X	Y	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

From the truth table, the expression for sum and carry bits of the output can be obtained as,

$$\text{SUM} = A'B'C + A'BC' + AB'C' + ABC = A \oplus B \oplus C$$

$$\text{CARRY} = A'BC + AB'C + ABC' + ABC = AB + (A \oplus B)C$$

**Logic Diagram:**



**RESULT:** Implementation of HALF ADDER & FULL ADDER using basic logic gates has been done in simulator.

## EXPERIMENT 4

**AIM:** Implementing Binary -to -Gray, Gray -to -Binary code conversions. Using COA Simulator

### **THEORY:**

Definition of Binary and Gray code:

**Gray code** – also known as **Cyclic Code**, **Reflected Binary Code (RBC)**, **Reflected Binary (RB)** or **Grey code** – is defined as an ordering of the binary number system such that each incremental value can only differ by one bit. In gray code, while traversing from one step to another step only one bit in the code group changes. That is to say that two adjacent code numbers differ from each other by only one bit.

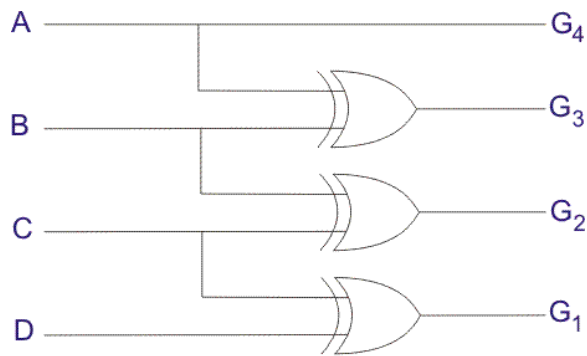
Gray code is the most popular of the unit distance codes, but it is not suitable for arithmetic operations. Gray code has some applications in analog to digital converters, as well as being used for error correction in digital communication. Gray code can be difficult to understand initially, but becomes much easier to understand when looking at the gray code tables below.

### **Binary to Gray Code Converter:**

The logical circuit which converts the binary code to equivalent gray code is known as **binary to gray code converter**. An n-bit gray code can be obtained by reflecting an n-1 bit code about an axis after  $2^{n-1}$  rows and putting the MSB (Most Significant Bit) of 0 above the axis and the MSB of 1 below the axis. Reflection of Gray codes is shown below.

The 4 bit binary to gray code conversion table is given below:

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



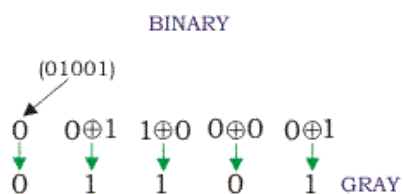
Logic Circuit for Binary to Gray Code Converter

### How to Convert Binary to Gray Code

1. The MSB (Most Significant Bit) of the gray code will be exactly equal to the first bit of the given binary number.
2. The second bit of the code will be exclusive-or (XOR) of the first and second bit of the given binary number, i.e if both the bits are same the result will be 0 and if they are different the result will be 1.
3. The third bit of gray code will be equal to the exclusive-or (XOR) of the second and third bit of the given binary number. Thus the binary to gray code conversion goes on. An example is given below to illustrate these steps.

### Binary to Gray Code Conversion Example

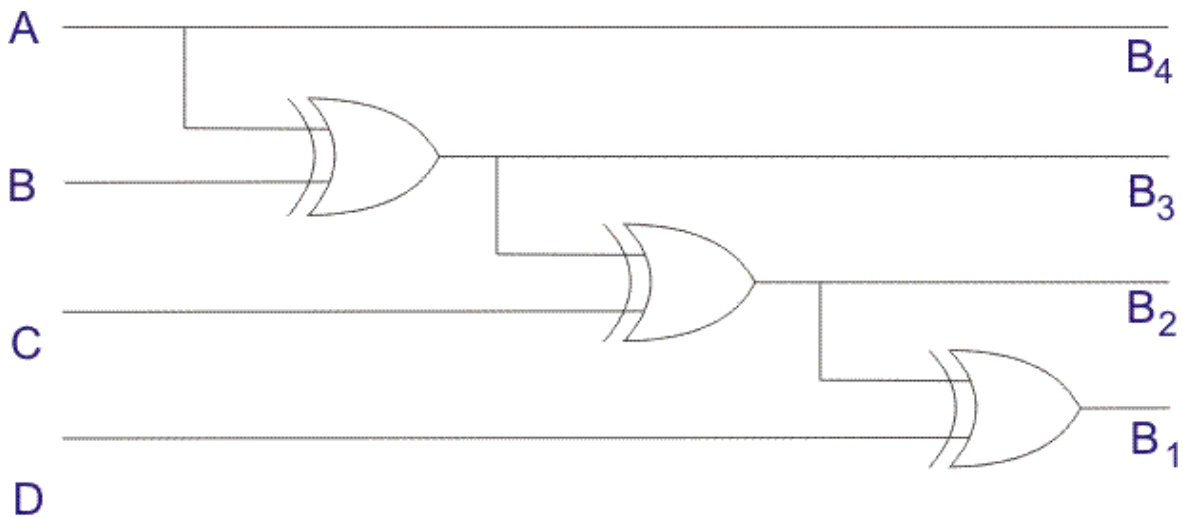
1. The MSB is kept the same. As the MSB of the binary is 0, the MSB of the gray code will be 0 as well (first gray bit)
2. Next, take the XOR of the first and the second binary bit. The first bit is 0, and the second bit is 1. The bits are different so the resultant gray bit will be 1 (second gray bit)
3. Next, take the XOR of the second and third binary bit. The second bit is 1, and the third bit is 0. These bits are again different so the resultant gray bit will be 1 (third gray bit)
4. Next, take the XOR of third and fourth binary bit. The third bit is 0, and the fourth bit is 0. As these are the same, the resultant gray bit will be 0 (fourth gray bit)
5. Lastly, take the XOR of the fourth and fifth binary bit. The fourth bit is 0, and the fifth bit is 1. These bits are different so the resultant gray bit will be 1 (fifth gray bit)
6. Hence the result of binary to gray code conversion of 01001 is complete, and the equivalent gray code is 01101.



## Gray to Binary Code Converter

In a **gray to binary code converter**, the input is gray code and output is its equivalent binary code.  
Gray to binary Code Conversion Table

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

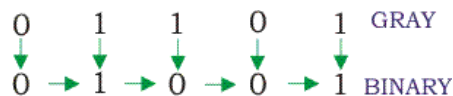


Logic Circuit for Gray to Binary Code Converter

### Gray Code to Binary Conversion

**Gray code to binary conversion** is again a very simple and easy process. Following steps can make your idea clear on this type of conversions.

1. The MSB of the binary number will be equal to the MSB of the given gray code.
2. Now if the second gray bit is 0, then the second binary bit will be the same as the previous or the first bit. If the gray bit is 1 the second binary bit will alter. If it was 1 it will be 0 and if it was 0 it will be 1.
3. This step is continued for all the bits to do **Gray code to binary conversion**.



Some other applications of gray code:

- Boolean circuit minimization
- Communication between clock domains
- Error correction
- Genetic algorithms
- Mathematical puzzles
- Position encoders

### Advantages of Gray Code

---

- Better for error minimization in converting analog signals to digital signals
- Reduces the occurrence of “Hamming Walls” (an undesirable state) when used in genetic algorithms
- Can be used to in to minimize a logic circuit
- Useful in clock domain crossing

### Disadvantages of Gray Code

---

- Not suitable for arithmetic operations
- Limited practical use outside of a few specific applications

**RESULT:** Implementation of Binary -to -Gray, Gray -to -Binary code conversions is successfully done.



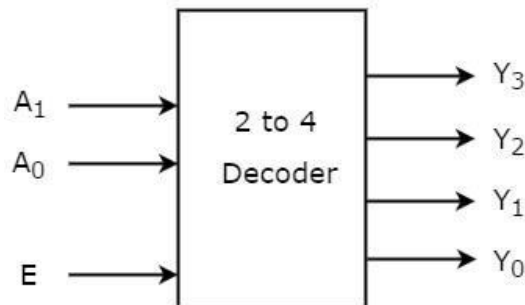
## Experiment 5

**AIM:** Implementing 3–8-line Decoder.

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

### 2 to 4 Decoder

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



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

Enable	Inputs		Outputs			
E	A1	A0	Y3	Y2	Y1	Y0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

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

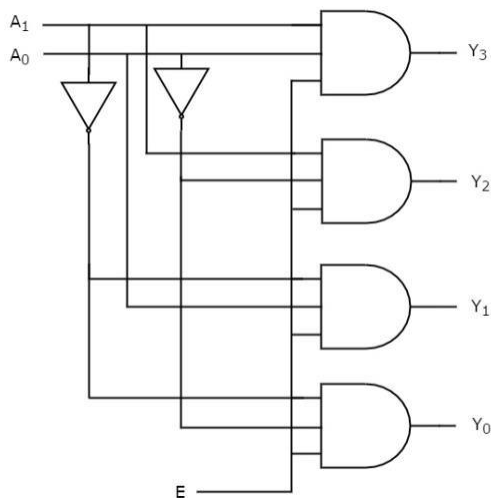
$$Y_0 = EA'1.A'0$$

$$Y_1 = EA'1.A0$$

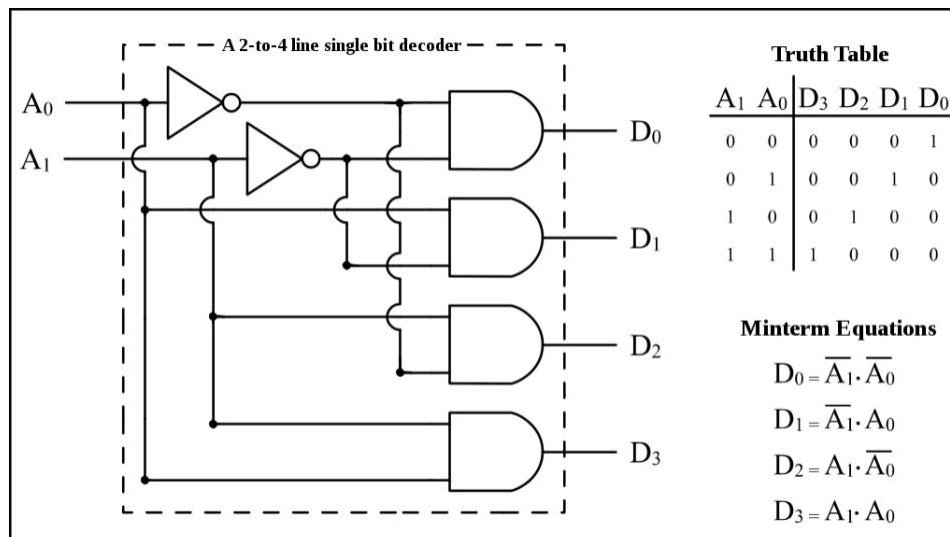
$$Y_2 = EA1.A'0$$

$$Y_3 = EA1.A0$$

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



By neglecting Enable bit we can design 2-4 Decoder as shown below-



Therefore, the outputs of 2 to 4 decoder are nothing but the **min terms** of two input variables A<sub>1</sub> & A<sub>0</sub>, when enable, E is equal to one. If enable, E is zero, then all the outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces eight min terms of three input variables A<sub>2</sub>, A<sub>1</sub> & A<sub>0</sub> and 4 to 16 decoder produces sixteen min terms of four input variables A<sub>3</sub>, A<sub>2</sub>, A<sub>1</sub> & A<sub>0</sub>.

### 3 to 8 Decoder

---

Similarly 3 to 8 Decoder has three inputs  $A_2$ ,  $A_1$  &  $A_0$  and eight outputs,  $Y_7$  to  $Y_0$ .

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

The **Truth table** of 3 to 8 decoder is shown below.

Enable	Inputs			Outputs							
E	$A_2$	$A_1$	$A_0$	$Y_7$	$Y_6$	$Y_5$	$Y_4$	$Y_3$	$Y_2$	$Y_1$	$Y_0$
0	x	x	x	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0	0	0	0	1
1	0	1	0	0	0	0	0	0	0	1	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0	0	0	0

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

$$Y_0 = E \cdot A_2' \cdot A_1' \cdot A_0'$$

$$Y_1 = E \cdot A_2' \cdot A_1' \cdot A_0$$

$$Y_2 = E \cdot A_2' \cdot A_1 \cdot A_0'$$

$$Y_3 = E \cdot A_2' \cdot A_1 \cdot A_0$$

$$Y_4 = E \cdot A_2 \cdot A_1' \cdot A_0'$$

$$Y_5 = E \cdot A_2 \cdot A_1' \cdot A_0$$

$$Y_6 = E \cdot A_2 \cdot A_1 \cdot A_0'$$

$$Y_7 = E \cdot A_2 \cdot A_1 \cdot A_0$$

27



27

## EXPERIMENT 6

**AIM:** Design 4\*1 and 8\*1 Multiplexers using COA simulator.

### Theory:

Multiplexers:

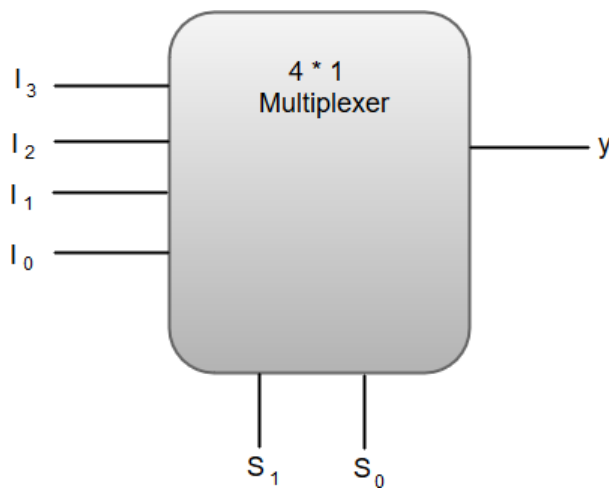
A Multiplexer (MUX) can be described as a combinational circuit that receives binary information from one of the  $2^n$  input data lines and directs it to a single output line.

The selection of a particular input data line for the output is decided on the basis of selection lines.

The multiplexer is often called as data selector since it selects only one of many data inputs.

Note: A  $2^n$ -to-1 multiplexer has  $2^n$  input data lines and  $n$  input selection lines whose bit combinations determine which input data are selected for the output.

Block diagram of 4\*1 Multiplexer



Out of these four input data lines, a particular input data line will be connected to the output based on the combination of inputs present at these two selection lines.

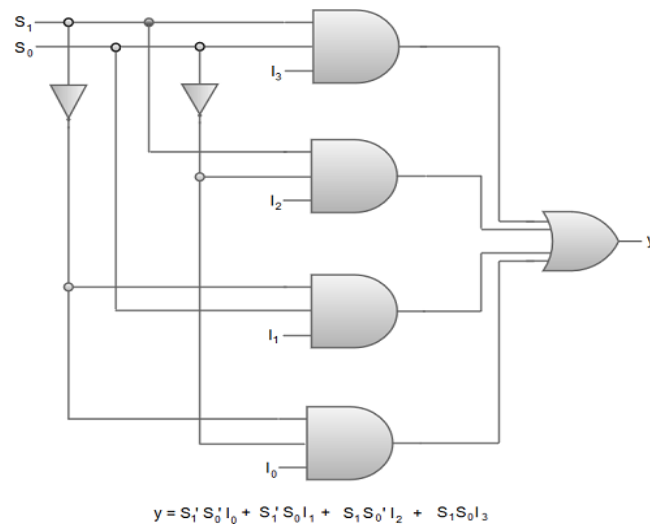
Truth Table of 4\*1 Multiplexer

S1	S0	Y
0	0	I0
0	1	I1
1	0	I2
1	1	I3

From the function table, we can write the Boolean function for the output (y) as:

$$y = S1'S0'I0 + S1'S0'I1 + S1S0'I2 + S1S0I3$$





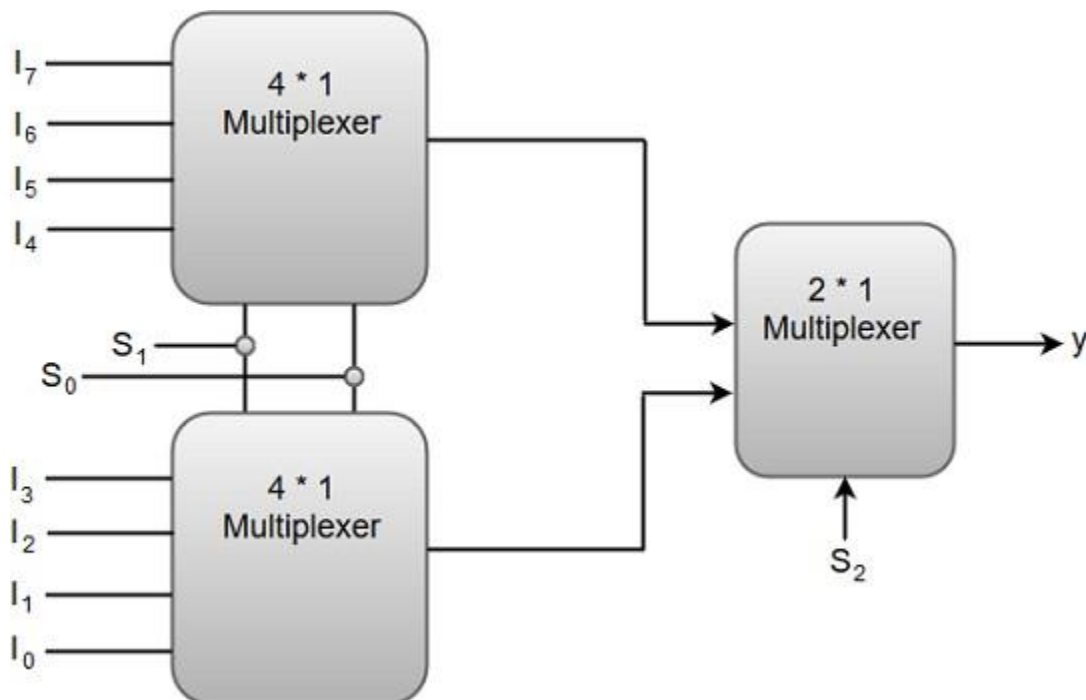
Digital Circuit Diagram of 4\*1 Multiplexer.

We can also implement higher order multiplexers using lower order multiplexers. For instance, let us implement an **8\*1 multiplexer** using two 4\*1 multiplexers and a 2\*1 multiplexer.

The two 4\*1 multiplexers are required in the first stage to get the eight input data lines.

A 2\*1 multiplexer is required in the second stage to converge the outputs generated at first stage into a single output.

The following image shows the block diagram of an 8\*1 multiplexer designed using two 4\*1 multiplexers and a single 2\*1 multiplexer.



	S1	S0	y
0	0	0	I0
0	0	1	I1
0	1	0	I2
0	1	1	I3
1	0	0	I4
1	0	1	I5
1	1	0	I6
1	1	1	I7

Function Table of 8\*1 Multiplexer.

**RESULT:** Implementation of multiplexers are done successfully.

## Experiment 7

**AIM:** Verify the excitation tables of various SR and T FLIP-FLOPS.

**THEORY:** Logic circuits for digital systems are either combinational or sequential. The output of combinational circuits depends only on the current inputs. In contrast, sequential circuit depends not only on the current value of the input but also upon the internal state of the circuit. Basic building blocks (memory elements) of a sequential circuit are the flip-flops (FFs). A flip-flop is a device which stores a single *bit* (binary digit) of data; one of its two states represents a "one" and the other represents a "zero". Such data storage can be used for storage of *state*, and such a circuit is described as sequential logic

There are four Flip-Flops as follows-

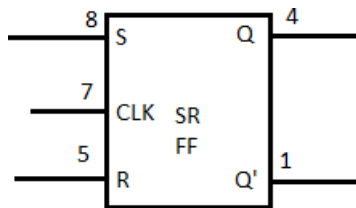
- SR Flip-Flop
- D Flip-Flop
- JK Flip-Flop
- T Flip-Flop

### **SR Flip-Flop**

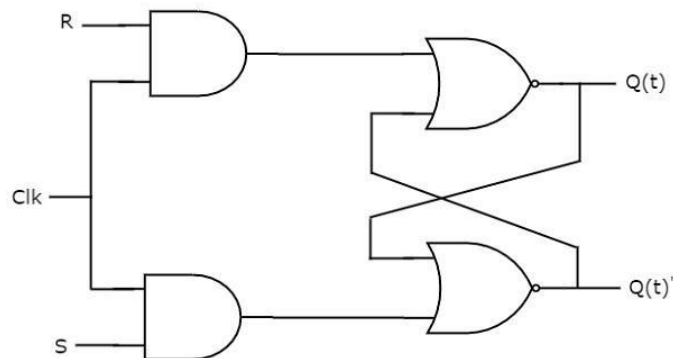
---

SR flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, SR latch operates with enable signal.

#### **Block Diagram of SR Flip Flop & Pin Diagram:**



The **circuit diagram** of SR flip-flop is shown in the following figure.



This circuit has two inputs S & R and two outputs Qt & Qt'. The operation of SR flipflop is similar to SR Latch. But, this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table shows the **state table** of SR flip-flop.

S	R	$Q_{t+1}$
0	0	$Q_t$
0	1	0
1	0	1
1	1	-

Here,  $Q_t$  &  $Q_{t+1}$  are present state & next state respectively. So, SR flip-flop can be used for one of these three functions such as Hold, Reset & Set based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of SR flip-flop.

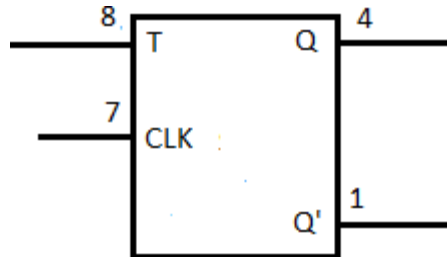
Present Inputs		Present State	Next State
S	R	$Q_t$	$Q_{t+1}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	x
1	1	1	x

The **simplified expression** for next state  $Q_{t+1}$  is

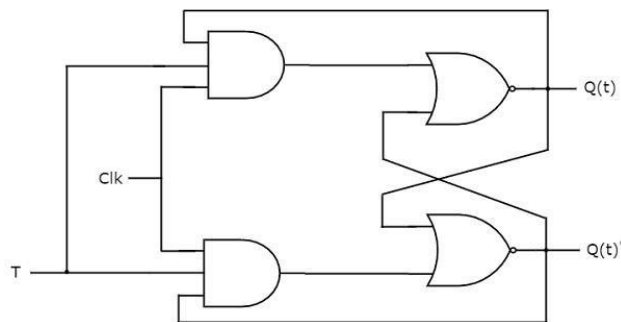
$$Q(t+1) = S + R'Q(t) \quad Q(t+1) = S + R'Q(t)$$

## T Flip-Flop

T flip-flop is the simplified version of JK flip-flop. It is obtained by connecting the same input 'T' to both inputs of JK flip-flop. It operates with only positive clock transitions or negative clock transitions.



The **circuit diagram** of T flip-flop is shown in the following figure.



This circuit has single input T and two outputs  $Q_t$  &  $Q_t'$ . The operation of T flip-flop is same as that of JK flip-flop. Here, we considered the inputs of JK flip-flop as  $J = T$  and  $K = T$  in order to utilize the modified JK flip-flop for 2 combinations of inputs. So, we eliminated the other two combinations of J & K, for which those two values are complement to each other in T flip-flop.

The following table shows the **state table** of T flip-flop.

T	$Q_{t+1}$
0	$Q_t$
1	$Q_t'$

Here,  $Q_t$  &  $Q_{t+1}$  are present state & next state respectively. So, T flip-flop can be used for one of these two functions such as Hold, & Complement of present state based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of T flip-flop.

Inputs	Present State	Next State
T	Q <sub>t</sub>	Q <sub>t+1</sub>
0	0	0
0	1	1
1	0	1
1	1	0

From the above characteristic table, we can directly write the **next state equation** as

$$Q(t+1) = T'Q(t) + TQ(t)' \Rightarrow Q(t+1) = T'Q(t) + TQ(t)'$$

$$\Rightarrow Q(t+1) = T \oplus Q(t) \Rightarrow Q(t+1) = T \oplus Q(t)$$

The output of T flip-flop always toggles for every positive transition of the clock signal, when input T remains at logic High 1. Hence, T flip-flop can be used in **counters**.

In this chapter, we implemented various flip-flops by providing the cross coupling between NOR gates. Similarly, you can implement these flip-flops by using NAND gates.

**RESULT:** Verification of SR and T FLIP-FLOPS are done successfully.

## Experiment 8

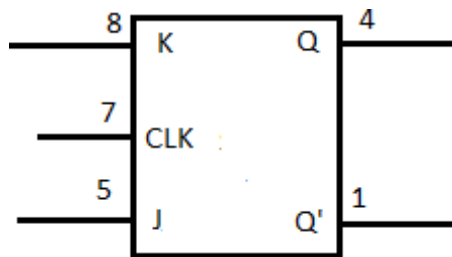
**AIM:** Verify the excitation tables of various JK and D FLIP-FLOPS.

### Theory:

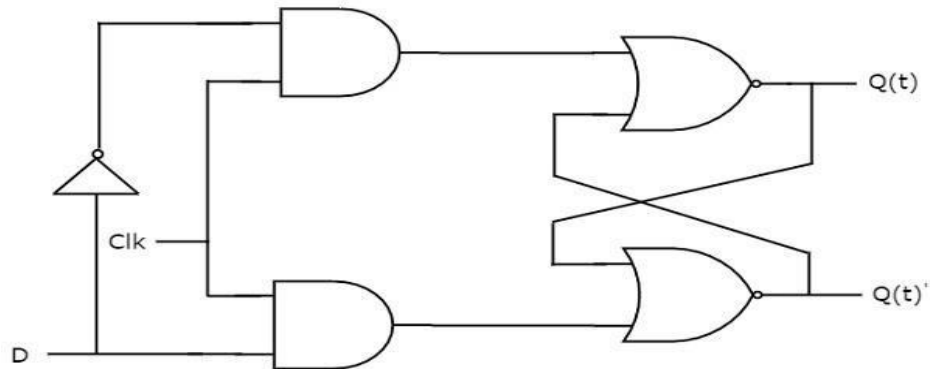
#### D Flip-Flop

D flip-flop operates with only positive clock transitions or negative clock transitions. Whereas, D latch operates with enable signal. That means, the output of D flip-flop is insensitive to the changes in the input, D except for active transition of the clock signal.

#### Block Diagram of SR Flip Flop & Pin Diagram



The **circuit diagram** of D flip-flop is shown in the following figure.



This circuit has single input D and two outputs Qtt & Qtt'. The operation of D flip-flop is similar to D Latch. But, this flip-flop affects the outputs only when positive transition of the clock signal is applied instead of active enable.

The following table shows the **state table** of D flip-flop.

D	$Q_{t+1}$
0	0
1	1

Therefore, D flip-flop always Hold the information, which is available on data input, D of earlier positive transition of clock signal. From the above state table, we can directly write the next state equation as

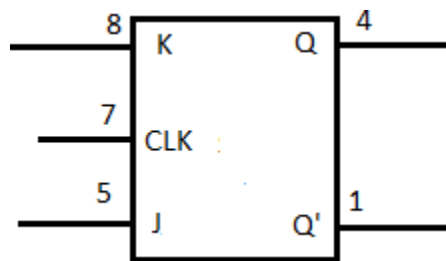
$$Q_{t+1} = D$$

Next state of D flip-flop is always equal to data input, D for every positive transition of the clock signal. Hence, D flip-flops can be used in registers, **shift registers** and some of the counters.

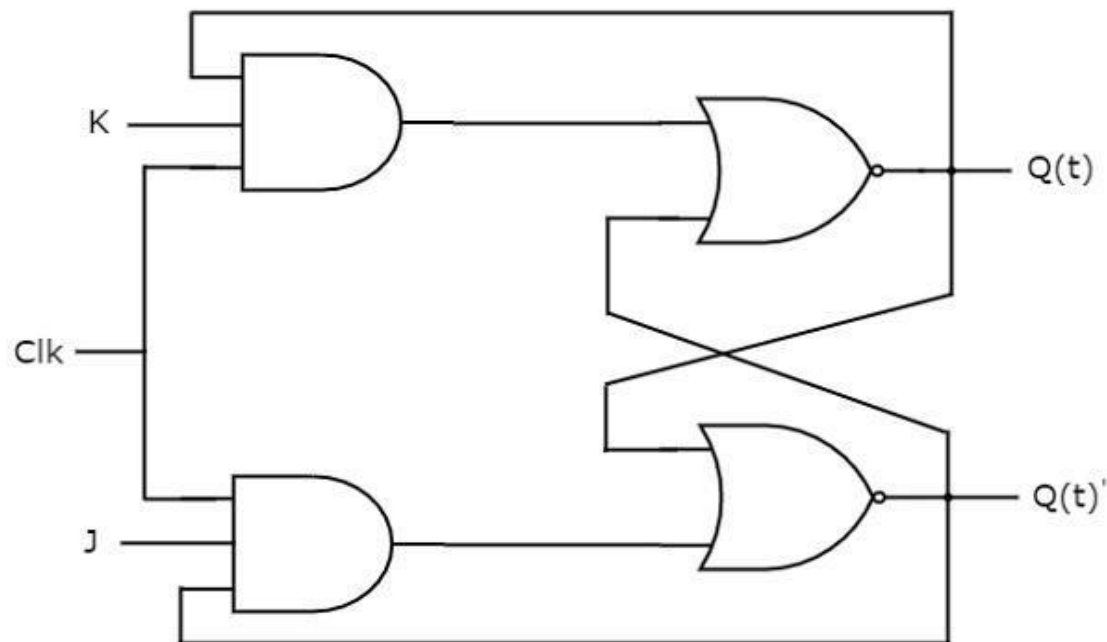
## JK Flip-Flop

JK flip-flop is the modified version of SR flip-flop. It operates with only positive clock transitions or negative clock transitions.

### Block Diagram of SR Flip Flop & Pin Diagram



The **circuit diagram** of JK flip-flop is shown in the following figure.



This circuit has two inputs J & K and two outputs  $Q_t$  &  $Q_t'$ . The operation of JK flip-flop is similar to SR flip-flop. Here, we considered the inputs of SR flip-flop as  $S = J Q_t'$  and  $R = K Q_t$  in order to utilize the modified SR flip-flop for 4 combinations of inputs.

The following table shows the **state table** of JK flip-flop.



<b>J</b>	<b>K</b>	<b>Q<sub>t+1</sub></b>
0	0	Q <sub>t</sub>
0	1	0
1	0	1
1	1	Q <sub>t</sub> '

Here, Q<sub>t</sub> & Q<sub>t+1</sub> are present state & next state respectively. So, JK flip-flop can be used for one of these four functions such as Hold, Reset, Set & Complement of present state based on the input conditions, when positive transition of clock signal is applied. The following table shows the **characteristic table** of JK flip-flop.

<b>Present Inputs</b>		<b>Present State</b>	<b>Next State</b>
<b>J</b>	<b>K</b>	<b>Q<sub>t</sub></b>	<b>Q<sub>t+1</sub></b>
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

The simplified expression for next state  $Q_{t+1} = JQ(t)' + K'Q(t)Q(t+1) = JQ(t)' + K'Q(t)$

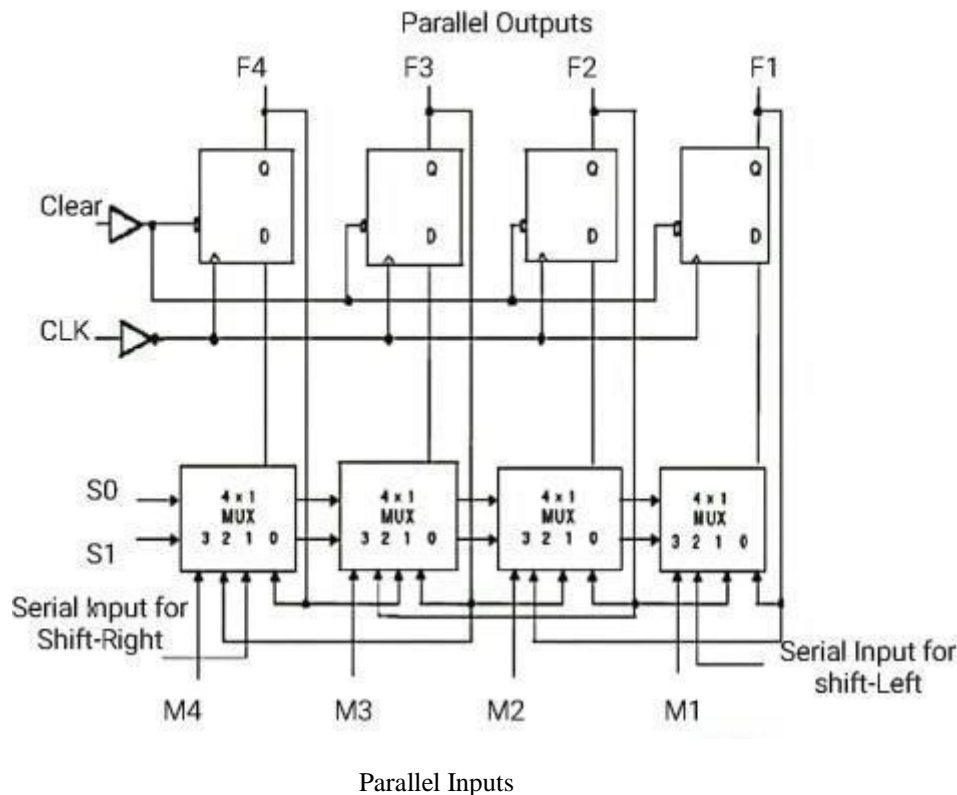
**RESULT:** Verification of JK and D FLIP-FLOPS are done successfully.

## Experiment 9

**AIM:** Design a 4bit Universal Shift Register

### **THEORY:**

The design of a 4-bit universal shift register using multiplexers and flip-flops is shown below.



### Universal Shift Register Design

- S0 and S1 are the selected pins that are used to select the mode of operation of this register. It may be shift left operation or shift right operation or parallel mode.
- Pin-0 of first 4×1 Mux is fed to the output pin of the first flip-flop. Observe the connections as shown in the figure.
- Pin-1 of the first 4X1 MUX is connected to serial input for shift right. In this mode, the register shifts the data towards the right.
- Similarly, pin-2 of 4X1 MUX is connected to the serial input for shift-left. In this mode, the universal shift register shifts the data towards the left.
- M1 is the parallel input data given to the pin-3 of the first 4×1 MUX to provide parallel mode operation and stores the data into the register.
- Similarly, remaining individual parallel input data bits are given to the pin-3 of related 4X1MUX to provide parallel loading.
- F1, F2, F3, and F4 are the parallel outputs of Flip-flops, which are associated with the 4×1 MUX.

### Universal Shift Register Working:

- From the above figure, selected pins the mode of operation of the universal shift register. Serial input shifts the data towards the right and left and stores the data within the register.

- Clear pin and CLK pin are connected to the flip-flop.
- M0, M1, M2, M3 are the parallel inputs while F0, F1, F2, F3 are the parallel outputs of flip-flops
- When the input pin is active HIGH, then the universal shift register loads / retrieve the data in parallel. In this case, the input pin is directly connected to 4×1 MUX
- When the input pin (mode) is active LOW, then the universal shift register shifts the data. In this case, the input pin is connected to 4×1 MUX via NOT gate.
- When the input pin (mode) is connected to GND (Ground), then the universal shift register acts as a Bi-directional shift register.
- To perform the shift-right operation, the input pin is fed to the 1st AND gate of the 1st flip-flop via serial input for shift-right.
- To perform the shift-left operation, the input pin is fed to the 8th AND gate of the last flip-flop via input M.
- If the selected pins S0= 0 and S1 = 0, then this register doesn't operate in any mode. That means it will be in a Locked state or no change state even though the clock pulses are applied.
- If the selected pins S0 = 0 and S1 = 1, then this register transfers or shifts the data to left and stores the data.
- If the selected pins S0 = 1 and S1 = 0, then this register shifts the data to right and hence performs the shift-right operation.
- If the selected pins S0 = 1 and S1 = 1, then this register loads the data in parallel. Hence it performs the parallel loading operation and stores the data.

S0	S1	Mode of Operation
0	0	Locked state (No change)
0	1	Shift-Left
1	0	Shift-Right
1	1	Parallel Loading

From the above table, we can observe that this register operates in all modes with serial/parallel inputs using 4×1 multiplexers and flip-flops.

#### Advantages

The **advantages of a universal shift register** include the following.

- This register can perform 3 operations such as shift-left, shift-right, and parallel loading.
- Stores the data temporarily within the register.
- It can perform serial to parallel, parallel to serial, parallel to parallel and serial to serial operations.
- It can perform input-output operations in both the modes serial and parallel.
- A Combination of the unidirectional shift register and bidirectional shift register gives the universal shift register.
- This register acts as an interface between one device to another device to transfer the data.

#### Applications

The **applications of a universal shift register** include the following.

- Used in micro-controllers for I/O expansion

- Used as a serial-to-serial converter
- Used as a parallel-to-parallel data converter
- Used as a serial-to-parallel data converter.
- Used in serial – to – serial data transfer
- Used in parallel data transfer.
- Used as a memory element in digital electronics like computers.
- Used in time delay applications
- Used as frequency counters, binary counters, and Digital clocks
- Used in data manipulation applications.

**RESULT:** Implementation of 4-bit Universal Shift Register is done successfully.

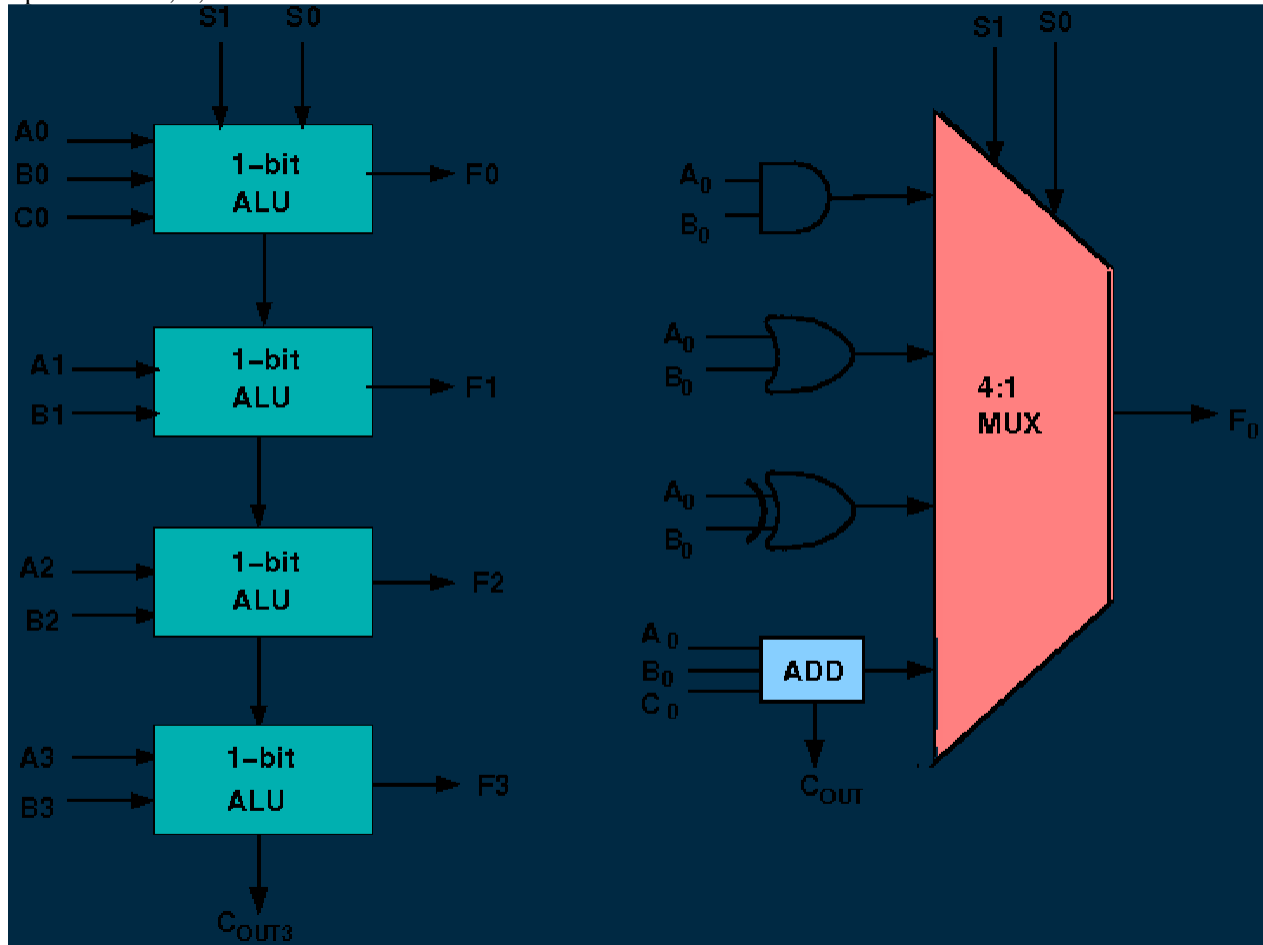
## Experiment 10

**AIM:** Design of an 4-bit ARITHMETIC LOGIC UNIT.

### **THEORY:**

Design of ALU:

ALU or Arithmetic Logical Unit is a digital circuit to do arithmetic operations like addition, subtraction, division, multiplication and logical operations like and, or, xor, nand, nor etc. A simple block diagram of a 4 bit ALU for operations and, or, xor and Add is shown here :



The 4-bit ALU block is combined using 4 1-bit ALU block

**Design Issues:**

The circuit functionality of a 1-bit ALU is shown here, depending upon the control signal  $S_1$  and  $S_0$  the circuit operates as follows:

for Control signal  $S_1 = 0$ ,  $S_0 = 0$ , the output is **A And B**,

for Control signal  $S_1 = 0$ ,  $S_0 = 1$ , the output is **A Or B**,

for Control signal  $S_1 = 1$ ,  $S_0 = 0$ , the output is **A Xor B**,

for Control signal  $S_1 = 1$ ,  $S_0 = 1$ , the output is **A Add B**.

**The truth table for 16-bit ALU with capabilities similar to 74181 is shown here:**

MODE SELECT				F <sub>N</sub> FOR ACTIVE HIGH OPERANDS	
INPUTS				LOGIC	ARITHMETIC (NOTE 2)
S3	S2	S1	S0	(M = H)	(M = L) (C <sub>n</sub> =L)
L	L	L	L	A'	A
L	L	L	H	A'+B'	A+B
L	L	H	L	A'B	A+B'
L	L	H	H	Logic 0	minus 1
L	H	L	L	(AB)'	A plus AB'
L	H	L	H	B'	(A + B) plus AB'
L	H	H	L	$A \oplus B$	A minus B minus 1
L	H	H	H	AB'	AB minus 1
H	L	L	L	A'+B	A plus AB
H	L	L	H	$(A \oplus B)'$	A plus B
H	L	H	L	B	(A + B') plus AB

H	L	H	H	AB	AB minus 1
H	H	L	L	Logic 1	A plus A (Note 1)
H	H	L	H	A+B'	(A + B) plus A
H	H	H	L	A+B	(A + B') plus A
H	H	H	H	A	A minus 1

Required functionality of ALU (inputs and outputs are active high)

The L denotes the logic low and H denotes logic high.

**RESULT:** Implementation of ALU is done successfully.

## Experiment 11

**AIM:** Design and implement half and full subtractor

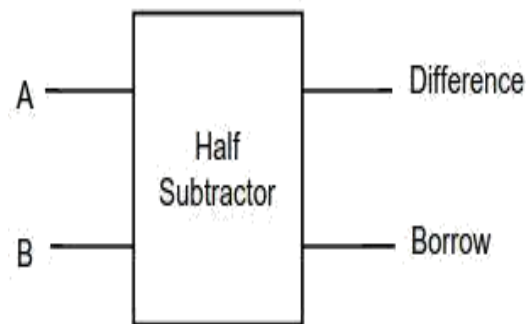
### **THEORY:**

Subtractor circuits take two binary numbers as input and subtract one binary number input from the other binary number input. Similar to adders, it gives out two outputs, difference and borrow (carry-in the case of Adder). There are two types of subtractors.

1. Half Subtractor
2. Full Subtractor

#### 1) Half Subtractor

The half-subtractor is a combinational circuit which is used to perform subtraction of two bits. It has two inputs, A (minuend) and B (subtrahend) and two outputs Difference and Borrow. The logic symbol and truth table are shown below.

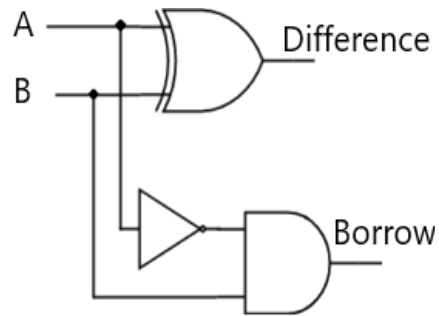


**Figure-1: Logic Symbol of Half subtractor**

Inputs		Outputs	
A	B	Difference	Borrow
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

**Figure-2: Truth Table of Half subtractor**





**Figure-3: Circuit Diagram of Half subtractor**

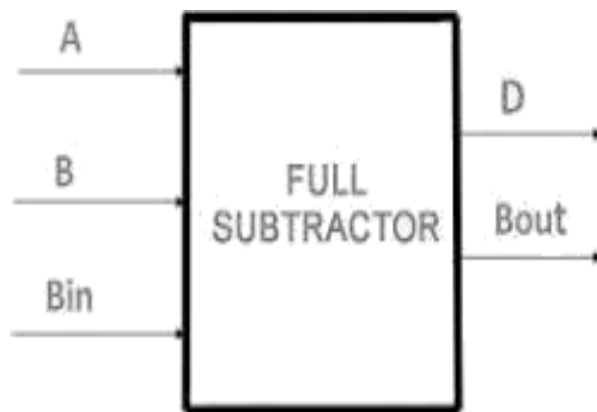
From the above truth table we can find the boolean expression.

$$\begin{aligned}\text{Difference} &= A \oplus B \\ \text{Borrow} &= A' B\end{aligned}$$

From the equation we can draw the half-subtractor circuit as shown in the figure 3.

## 2) Full Subtractor

A full subtractor is a combinational circuit that performs subtraction involving three bits, namely A (minuend), B (subtrahend), and Bin (borrow-in). It accepts three inputs: A (minuend), B (subtrahend) and a Bin (borrow bit) and it produces two outputs: D (difference) and Bout (borrow out). The logic symbol and truth table are shown below.



**Figure-4: Logic Symbol of Full subtractor**

A	B	B <sub>in</sub>	D	B <sub>out</sub>
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

Figure-5: Truth Table of Full subtractor

From the above truth table we can find the boolean expression.

$$D = A \oplus B \oplus B_{in}$$

$$B_{out} = A' B_{in} + A' B + B B_{in}$$

From the equation we can draw the Full-subtractor circuit as shown in the figure 6.

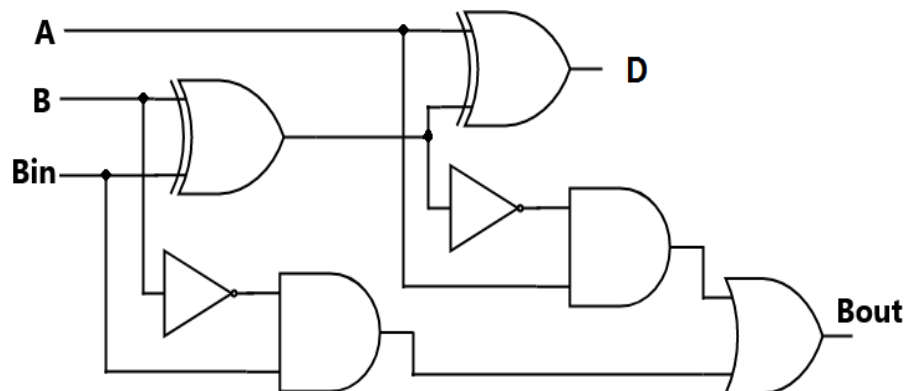


Figure-6: Circuit Diagram of Full subtractor

**RESULT:** Implementation of half and full subtractor is done successfully.

## Experiment 12

**AIM:** Design and implement Binary to BCD and BCD to Binary converter.

### **THEORY:**

Binary to BCD code converter

In BCD code, 0 to 9 numbers represent the equivalent binary numbers. For the numbers above 10, LSB of a decimal number is represented by its equivalent binary number and MSB of a decimal number is also represented by their equivalent [binary numbers](#).

For example, the BCD code of 12 is represented as

LSB      MSB  
12  
0001 0010  
The BCD code for 12 is 10010

The following truth table shows the conversion between the binary code input and the BCD code output. As you see from the table, the 4-bit binary number is converted into 5-bit BCD code. Decimal code is added in the table to understand the equivalence of Binary and BCD code.

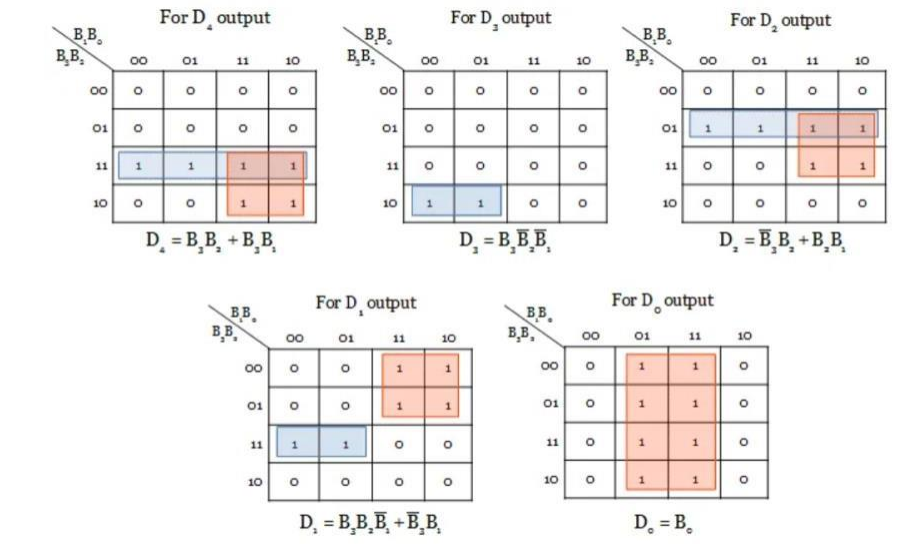
Decimal Number	Binary code (Input)				BCD code (Output)				
	B <sub>3</sub>	B <sub>2</sub>	B <sub>1</sub>	B <sub>0</sub>	D <sub>4</sub>	D <sub>3</sub>	D <sub>2</sub>	D <sub>1</sub>	D <sub>0</sub>
0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	1
2	0	0	1	0	0	0	0	1	0
3	0	0	1	1	0	0	0	1	1
4	0	1	0	0	0	0	1	0	0
5	0	1	0	1	0	0	1	0	1
6	0	1	1	0	0	0	1	1	0
7	0	1	1	1	0	0	1	1	1
8	1	0	0	0	0	1	0	0	0
9	1	0	0	1	0	1	0	0	1
10	1	0	1	0	1	0	0	0	0
11	1	0	1	1	1	0	0	0	1
12	1	1	0	0	1	0	0	1	0
13	1	1	0	1	1	0	0	1	1
14	1	1	1	0	1	0	1	0	0
15	1	1	1	1	1	0	1	0	1

The converter has 5 outputs D<sub>0</sub>, D<sub>1</sub>, D<sub>2</sub>, D<sub>3</sub> and D<sub>4</sub>. From the truth table, the [minterms](#) can be obtained for each output.

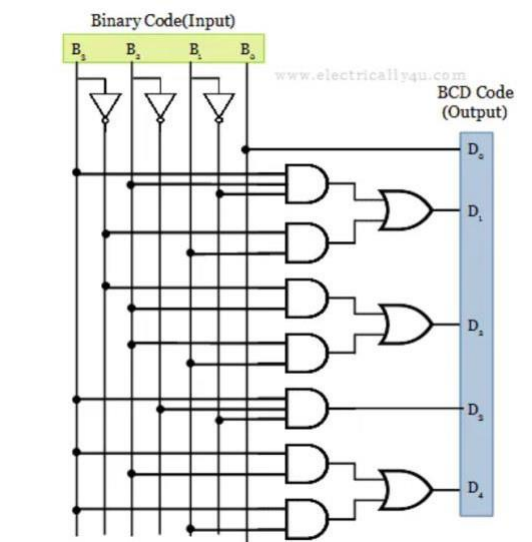
$$D_4 = \sum m(10, 11, 12, 13, 14, 15), D_3 = \sum m(8, 9), D_2 = \sum m(4, 5, 6, 7, 14, 15), D_1 = \sum m(2, 3, 6, 7, 12, 13), D_0 = \sum m(1, 3, 5, 7, 9, 11, 13, 15)$$

The minterms are plotted in the [karnaugh map](#) and the simplified boolean expressions are obtained.

[Minimize a boolean function using K-map](#)



The digital logic circuit for Binary to [BCD code](#) converter is designed from the simplified output expressions obtained from karnaugh map.



**RESULT:** Design and implementation of Binary to BCD is done successfully.