**Practical File**

***Lab Name……………………COA-LAB………….………. Lab Code…………KCS-352………………***

.



**Name………………SAMARPIT DUA………………..**

**Adm.No……2019B111068………………….. Univ. Roll No…………1900321290050.…...…**

**Course ………B.TECH……………………..……Branch…………CEIT…………..……….…………..**

**Sem………………3…..…………….…… Section………………….A.………….…..……..**





**Lab Manual Odd SEM**

**Session 2020-2021**

**Subject Name : Computer Organization Lab**

**Subject Code :KCS-302**

**Semester /Section :IIIrd SEM**

**Faculty Name : Prof. Laxmi Saraswat/ Prof. Poonam Rana**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. **S.No.** | **Experiment Name** | **Date Held** | **Submission Date** | **Grade** |
| 1 | To demonstrate logic gates using simulator. | 07/10/2020 | 07/10/2020 |  |
| 2 | Implementing HALF ADDER and FULL ADDER using basic logic gates. | 14/10/2020 | 14/10/2020 |  |
| 3 | Implementing binary-to-grey, grey-to-binary code conversions. | 21/10/2020 | 21/10/2020 |  |
| 4 | Implementing 3-8 line decoder. | 04/11/2020 | 04/11/2020 |  |
| 5 | Implementing 4\*1 and 8\*1 Multiplexers. | 11/11/2020 | 11/11/2020 |  |
| 6 | Verification of various excitation states of flip-flops. | 18/11/2020 | 18/11/2020 |  |
| 7 | Designing an 8-bit ALU. | 16/12/2020 | 16/12/2020 |  |
| 8 | CPU design. | 23/12/2020 | 23/12/2020 |  |

**ABES Engineering College, Ghaziabad**

**Department of Computer Science**

**List of Experiments**

**(Sign of Faculty) Dr. (Prof.) Pankaj Sharma**

**HOD(CS)**

**Practical-1**

**Practical Name: Design Basic Logic Gates Using Virtual Lab**

* **PRACTICAL STATEMENT OF PRACTICAL:**

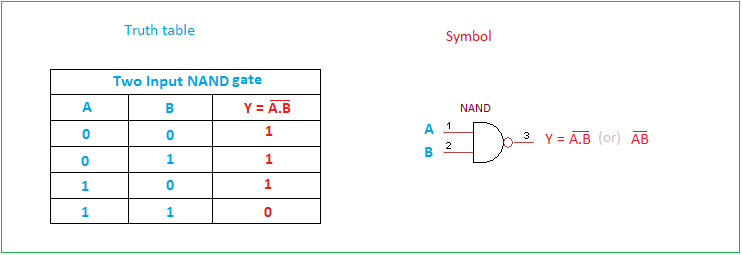
To design basics logic gates using COA simulator by IIT Kharagpur .

* **OBJECTIVE OF PRACTICAL**

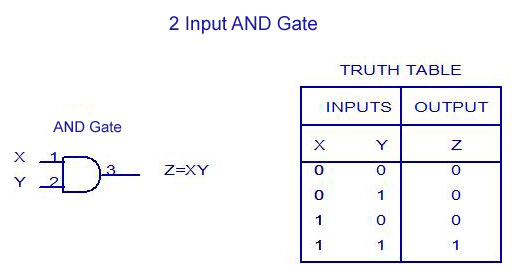
1- To study and understand the basic logic gates.

2- Implement the basic logic gates.

1. **NAND GATE:**



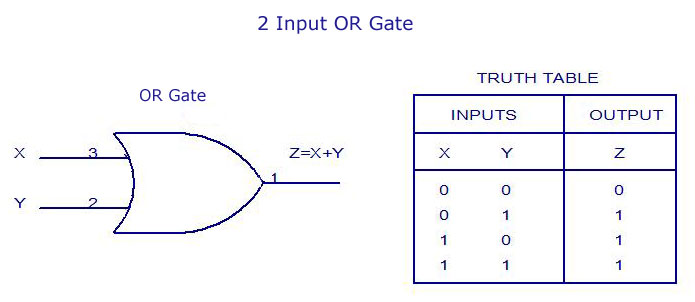
1. **AND GATE:**



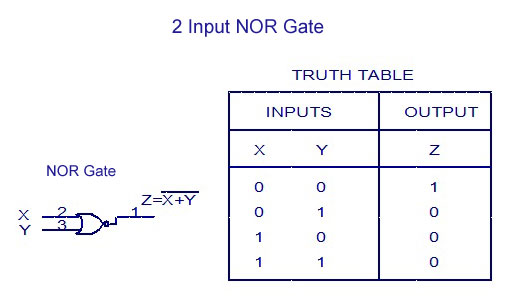
**Roll no : 1900321290050 Date:07-09-2020 page no:02**

**Practical Name : Design Basic Logic Gates Using Virtual Lab**

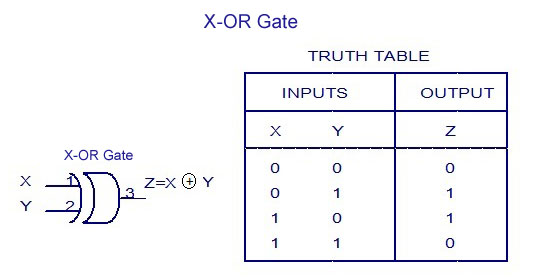
1. **OR GATE:**



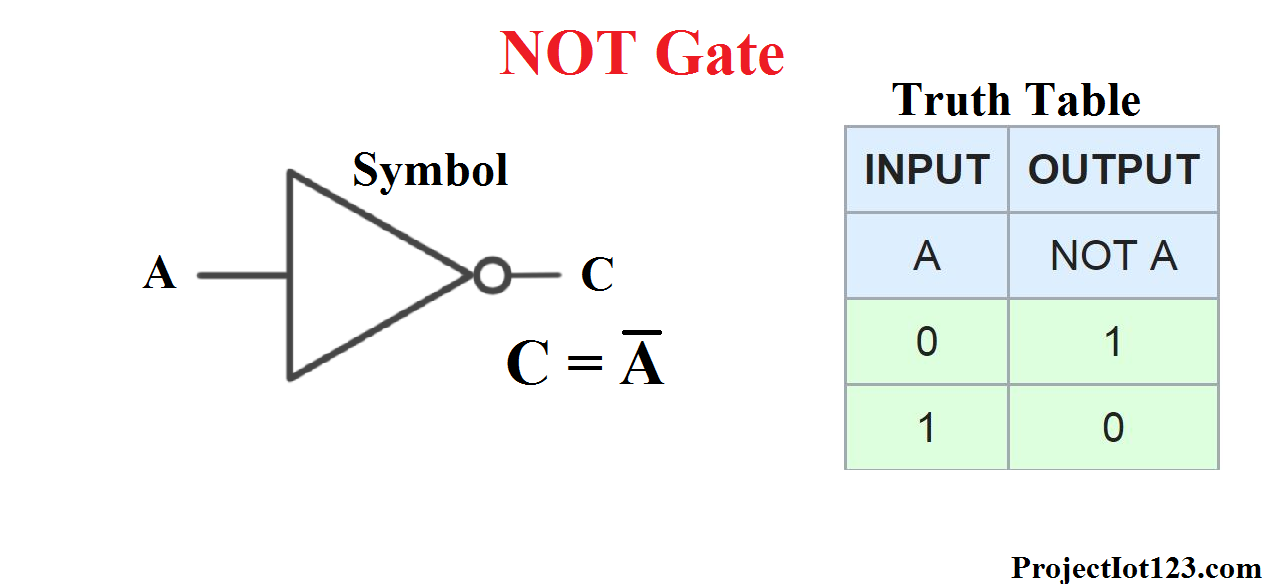
1. **NOR GATE:**



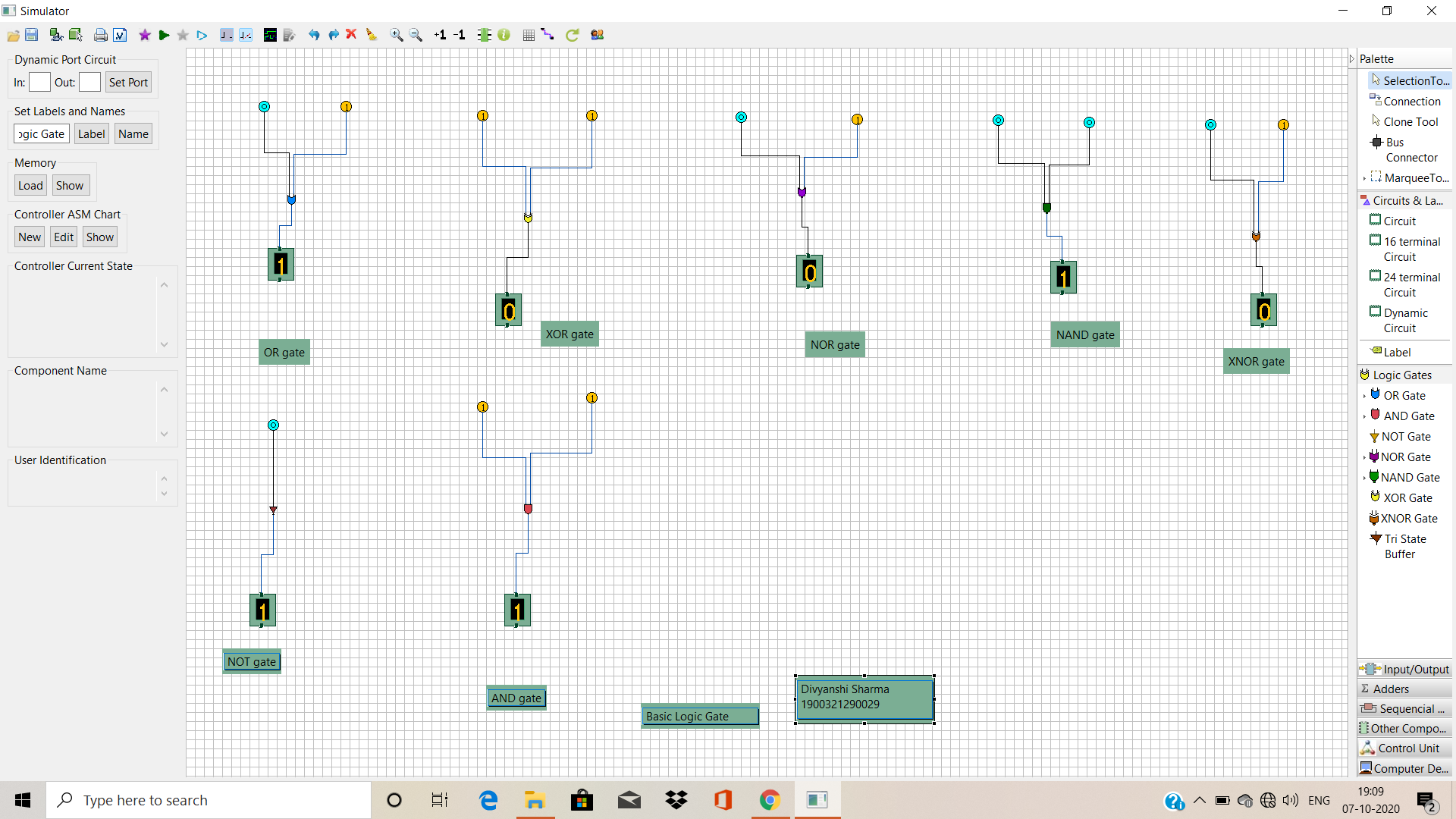
1. **XOR GATE:**



**6. NOT GATE:**



**Implementation :**

****

**PRACTICAL – 2**

1. **PROBLEM STATEMENT:** **Implementing HALF ADDER, FULL ADDER using basic logic gates.**
2. **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 (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.

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.

**Truth Table:**

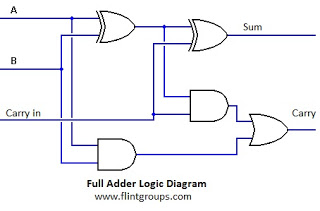
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Inputs** | | | **Outputs** | |
| **X** | **Y** | Cin | **S** | **Cout** |
| **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **1** | **1** | **0** |
| **0** | **1** | **0** | **1** | **0** |
| **0** | **1** | **1** | **0** | **1** |
| **1** | **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,**

**SUM = A’B’C + A’BC’ + AB’C’ + ABC= A B C**

**CARRY = A’BC + AB’C + ABC’ +ABC=AB+(A B)C**

1. **Logic Diagram:**

****

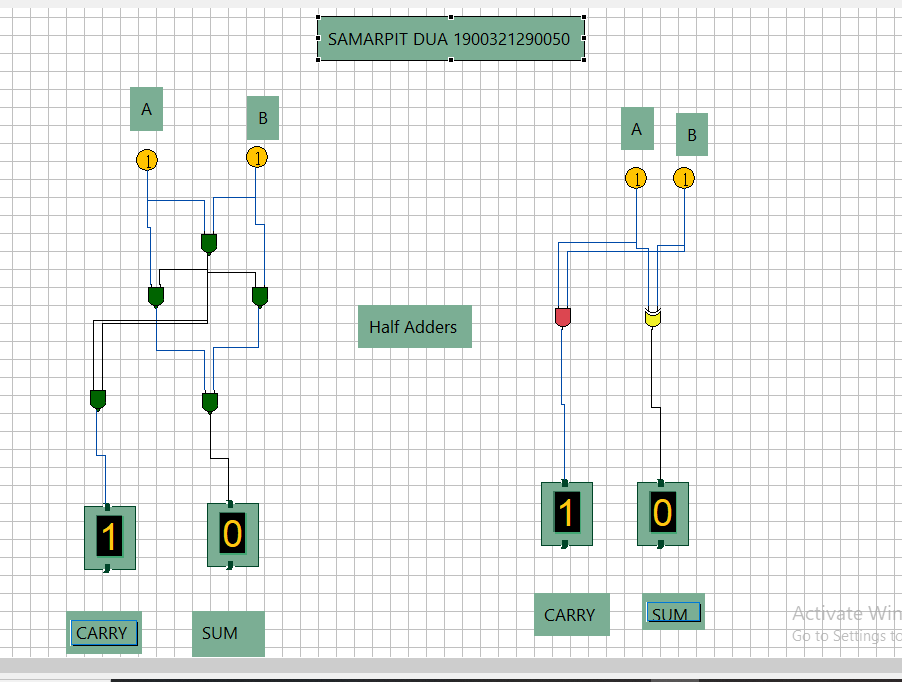
1. **Simulation:-**

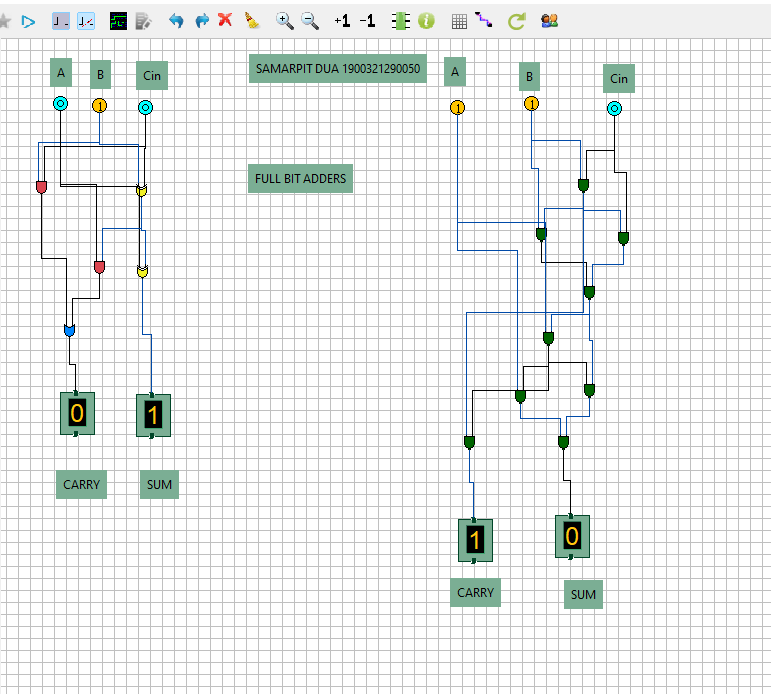
**Using simulator produced by IIT-Khadagpur we can simulator to implement half adder & full adder can verify results/truth table of half/full adder.**

**1. Half Adder Simulation :- First make Half adder using logic gates in editor window then connect input and output display element for verifying truth table of adder as shown in screenshot of simulator edit window.**

**2. Full Adder Simulation:- Make full adder using logic gates in simulator editor window as shown below in screenshot and verify truth table of full adder by simulating diagram.**

**5. IMPLEMENTATION:**

****



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

**Practical -3**

1. **PROBLEM STATEMENT:** In this we have to convert binary number into gray code

and vice versa.

1. **OBJECTIVE :** Implementing Binary -to -Gray, Gray -to -Binary code conversions. Using COA Simulator in the following ways

* Binary to Gray code using simple logic gates
* Gray to Binary using simple logic gates
* Binary to Gray code using NAND gates
* Gray to Binary code using NAND gates
* Binary to Gray code using NOR gates
* Gray to Binary code using NOR gates

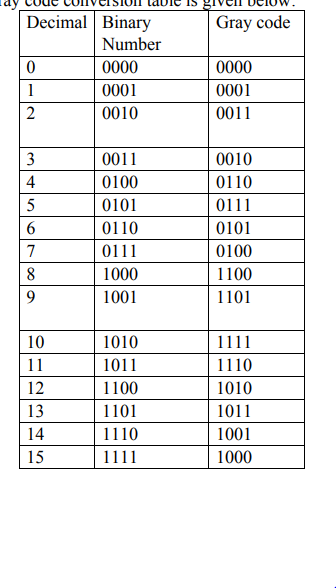
1. **DEFINITION OF GRAY CODE AND BINARY CODE:**

**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

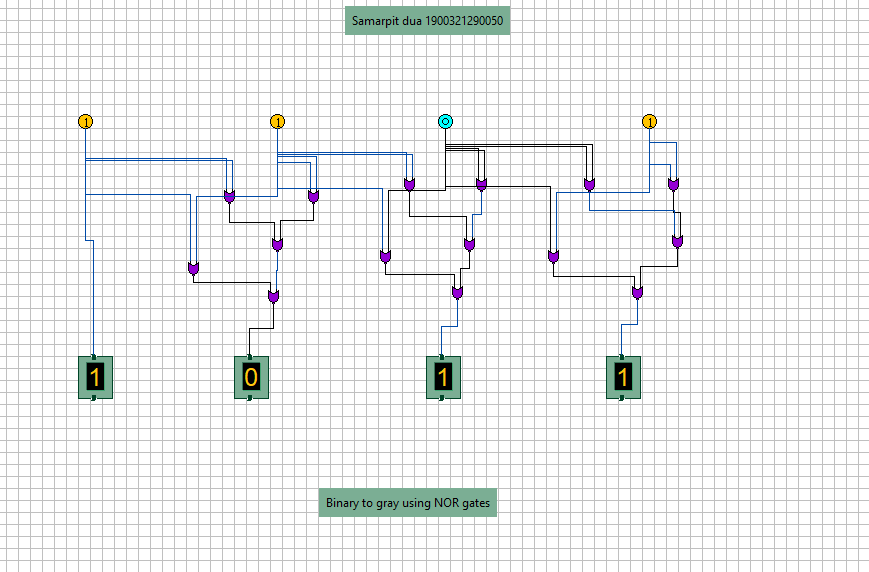
is to say that two adjacent code numbers differ from each other by only one bit.Gray code is the most popular of the unit distance codes, but it is not suitable for arithmetic operations. Gray code has some applications in analog to digital converters, as well as being used for error correction in digital communication. Gray code can bedifficult to understand initially, but becomes much easier to understand when we look at tables

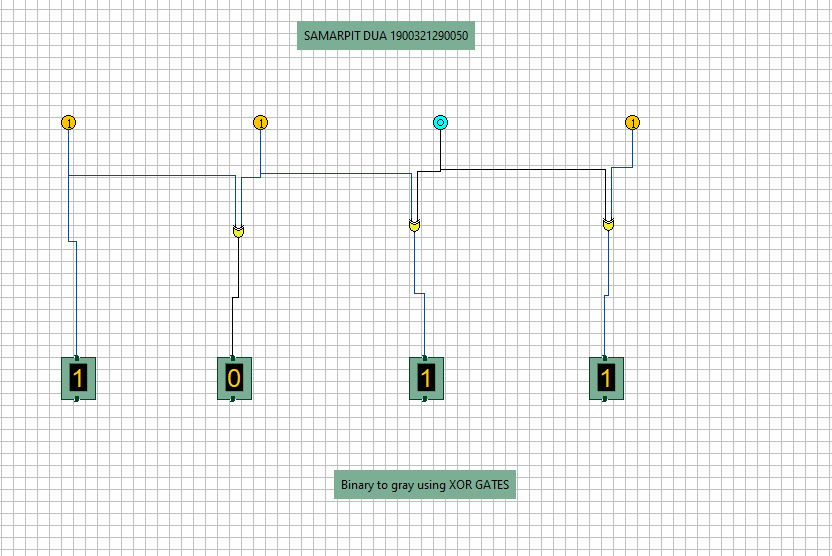
**Binary to Gray Code Converter**The logical circuit which converts the binary code to equivalent gray code is known as **binary to gray code converter**. An n-bit gray code can be obtained by reflecting

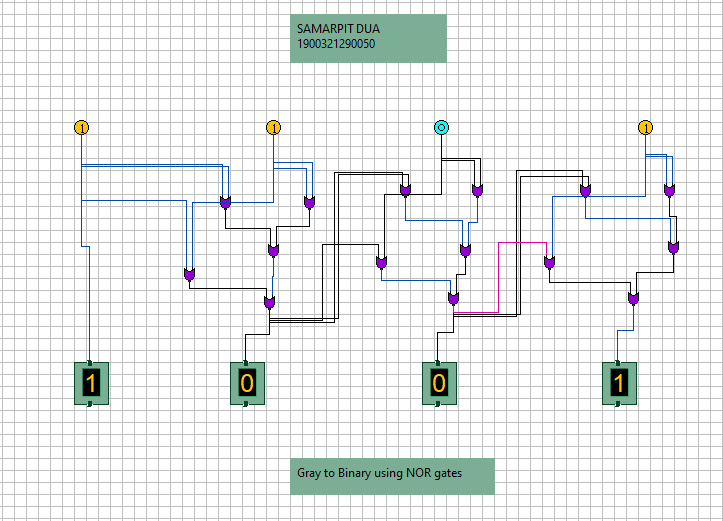
an n-1 bit code about an axis after 2n-1 rows and putting the MSB (Most Significant Bit) of 0 above the axis and the MSB of 1 below the axis.**TABLE:**

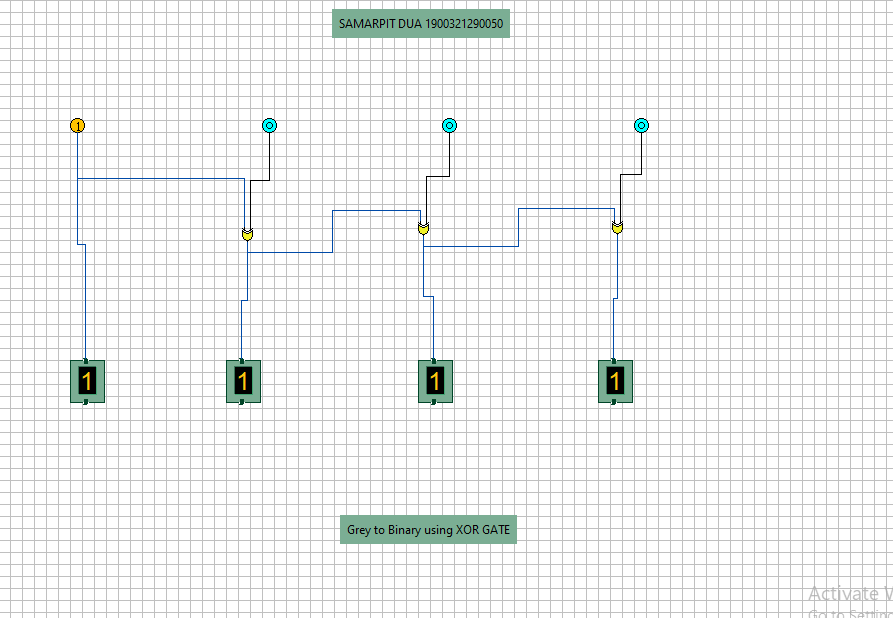


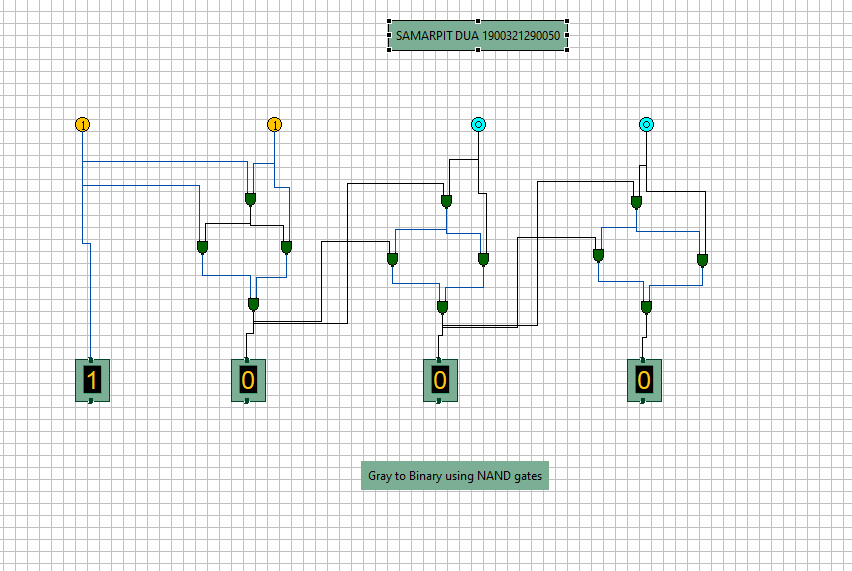
1. **IMPLEMENTATION:**

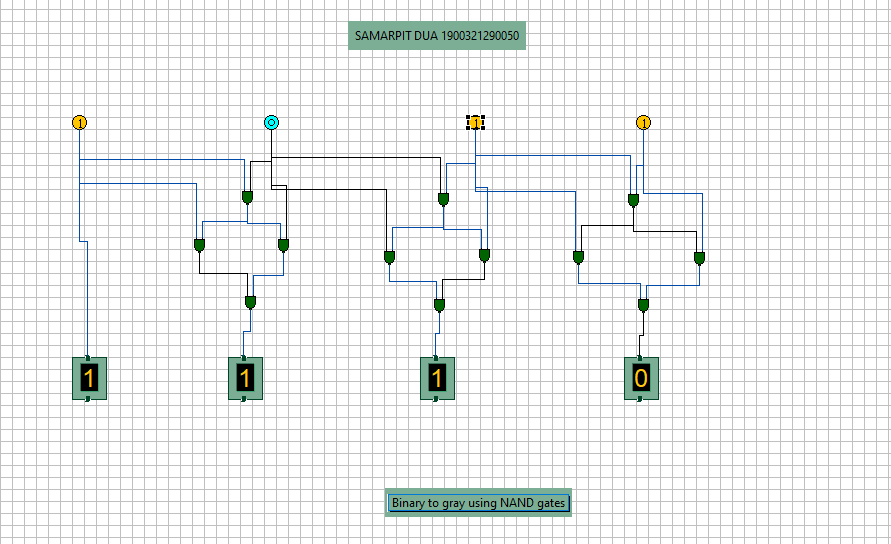
****

****

****

****

****

****

**6.RESULT:**Thus with the help of the logic gates(XOR,NAND,NOR) as shown, we can easily convert GRAY code to BINARY code and visa-versa.

**Practical-4**.

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

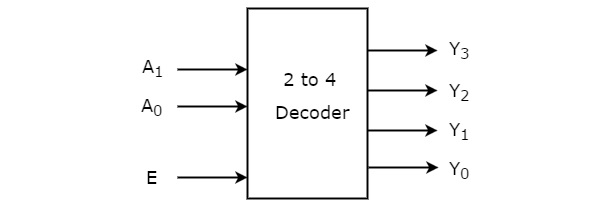
**Theory:-** Decoder is a combinational circuit that has ‘n’ input lines and maximum of 2n output lines. One of these outputs will be active High based on the combination of inputs present, when the decoder is enabled. That means decoder detects a particular code. The outputs of the decoder are nothing 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 & A and four outputs Y , Y , Y & Y .

1 0 3 2 1 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.

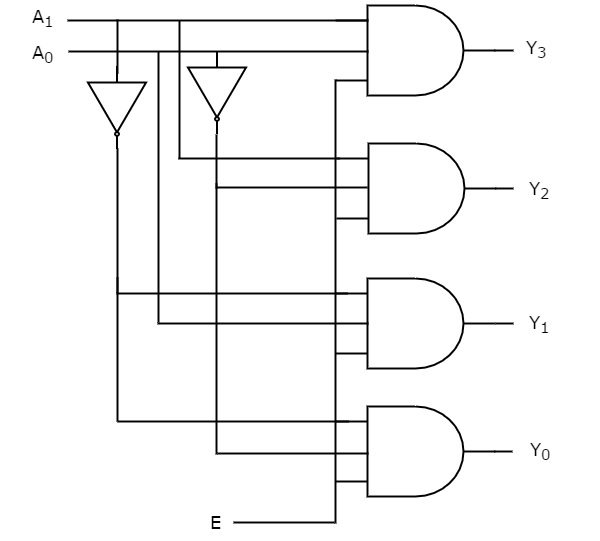
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Enable | Inputs | | | Outputs | | | | |
| E | | A1 | A0 | Y3 | Y2 | | Y1 | Y0 |
| 0 | | x | X | 0 | | 0 | 0 | 0 |
| 1 | | 0 | 0 | 0 | | 0 | 0 | 1 |
| 1 | | 0 | 1 | 0 | | 0 | 1 | 0 |
| 1 | | 1 | 0 | 0 | | 1 | 0 | 0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

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

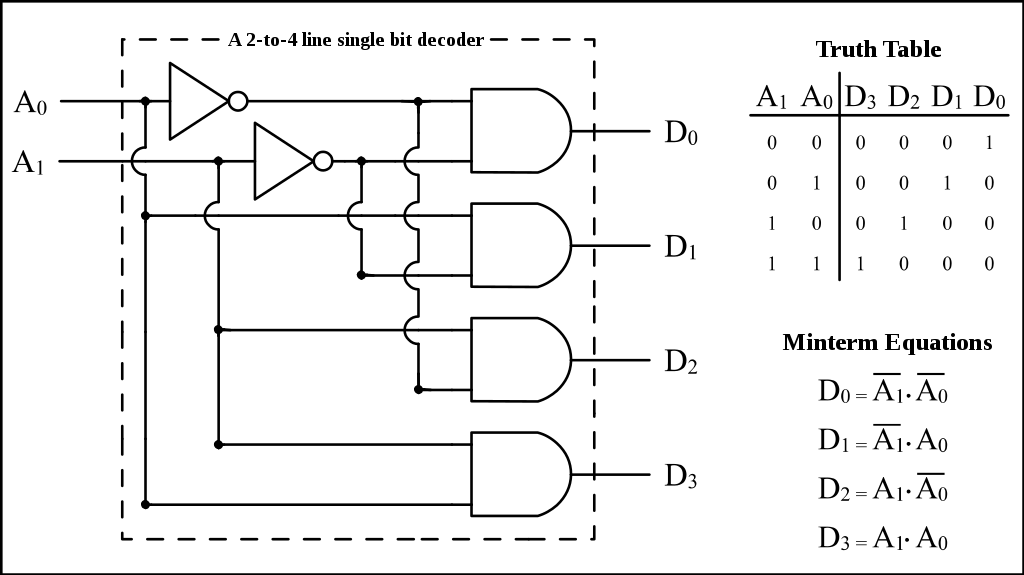
Y0=EA’1.A’0

Y1=EA’1.A0 Y2=EA1.A’0 Y3=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 & A , when enable, E is equal to one. If enable, E is zero, then all the

1 0

outputs of decoder will be equal to zero.

Similarly, 3 to 8 decoder produces eight min terms of three input variables A , A &

2 1

A and 4 to 16 decoder produces sixteen min terms of four input variables A , A ,

0 3 2 A & A .

1 0

**3 to 8 Decoder**



Similarly 3 to 8 Decoder has three inputs A , A & A and eight outputs, Y to Y .

2 1 0 7 0

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

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

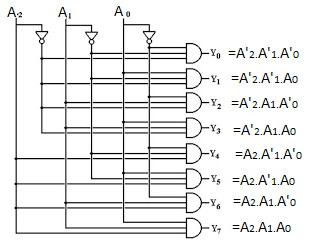
|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Enable | Inputs | | | Outputs | | | | | | | |
| E | A2 | A1 | A0 | Y7 | Y6 | Y5 | Y4 | Y3 | Y2 | Y1 | Y0 |
| 0 | X | x | x | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 0 | 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

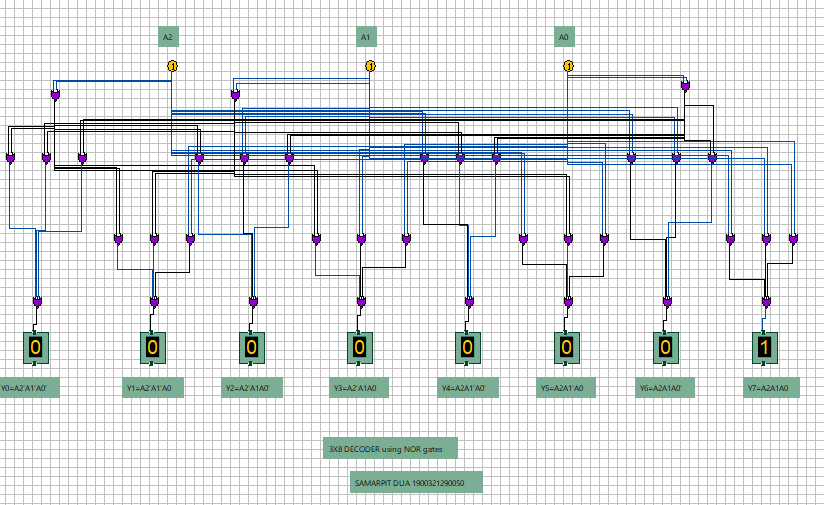
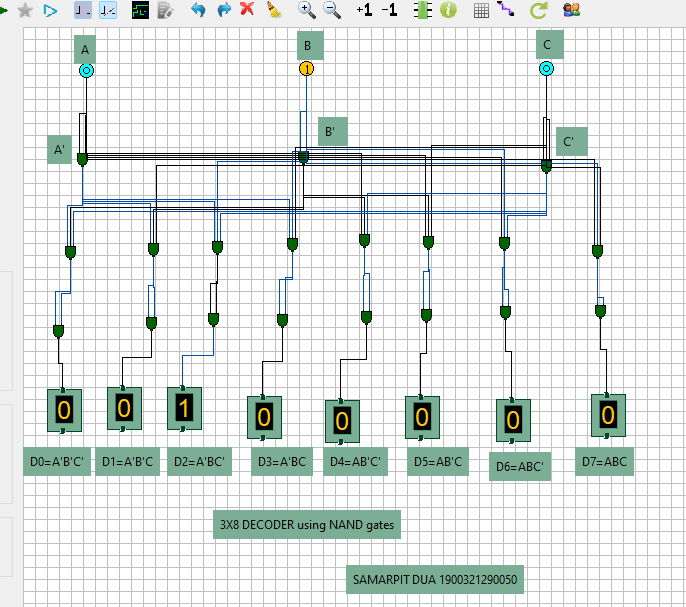
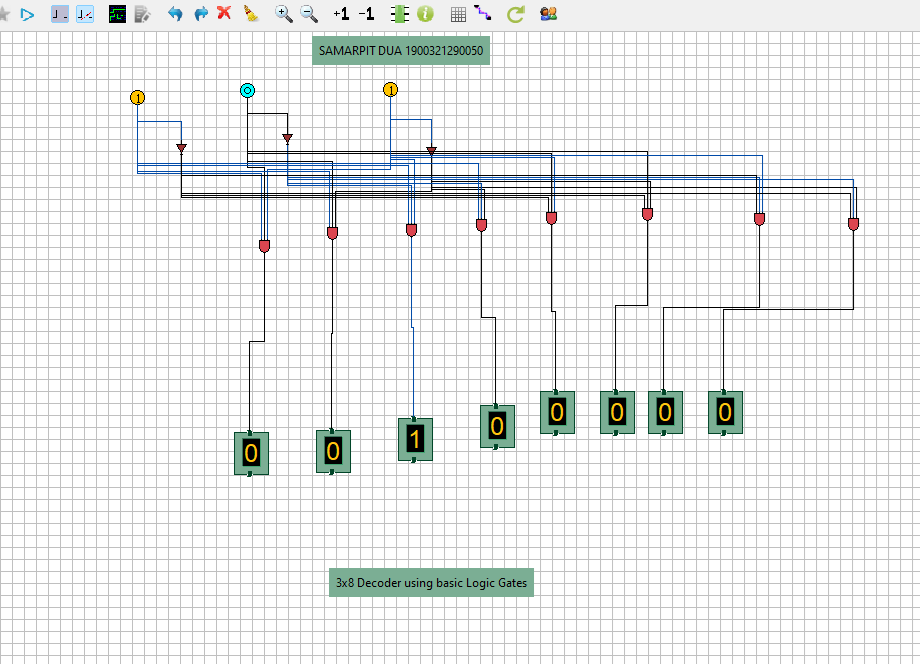
Y0=E A’2.A’1.A’0

Y1=E A’2A’1.A0 Y2=E A’2A1.A’0 Y3=E A’2A1.A0 Y4=E A2A’1.A’0 Y5=E A2A’1.A0 Y6=E A2A1.A’0 Y7=E A2A1.A0

By ignoring Enable bit we can design 3-8 line Decoder as shown in Diagram below-



**Simulation:- 3-8 Decoders are simulated in simulator, screenshots are given below as-**



**Practical -5**

## EXPERIMENT NO. 5

**EXPERIMENT NAME: Implementing 4\*1 and 8\*1 Multiplexers**

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

## 

**OBJECTIVE:** To implement the working of 4\*1 and 8\*1 Multiplexer.

### **APPARATUS REQUIRED:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sr. no. | Name of  Equipments/components/software | Specification/range/rating/ version | Quantity |
| 1 | COA Simulator by IIT Kharagpur |  |  |

**Multiplexers:**

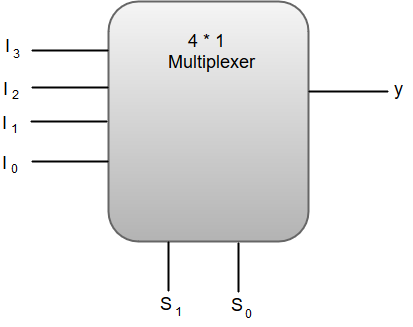
A Multiplexer (MUX) can be described as a combinational circuit that receives binary information from one of the 2^n input data lines and directs it to a single output line.

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



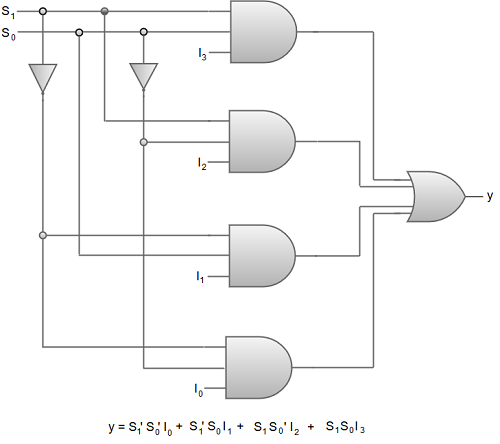
Out of these four input data lines, a particular input data line will be connected to the output based on the combination of inputs present at these two selection lines.

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



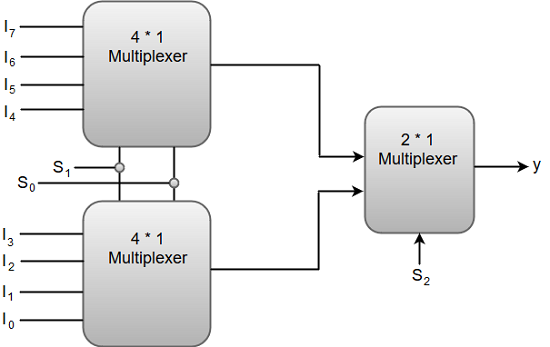
Digital Circuit Diagram of 4\*1 Multiplexer.

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.

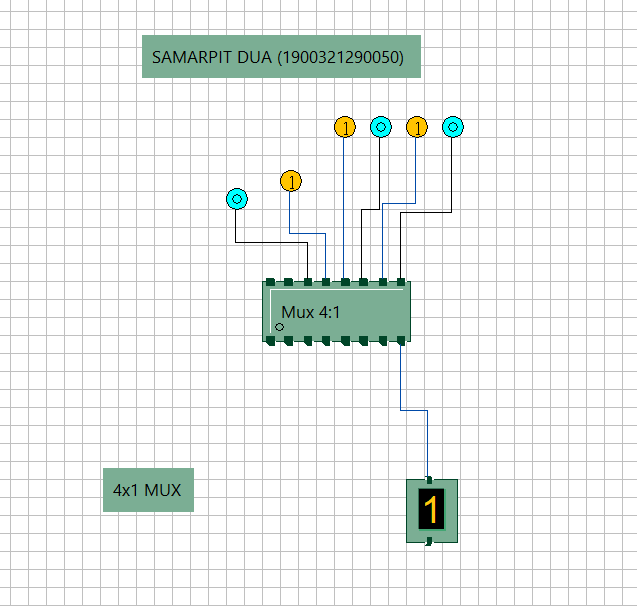
The following image shows the block diagram of an 8\*1 multiplexer designed using two 4\*1 multiplexers and a single 2\*1 multiplexer.

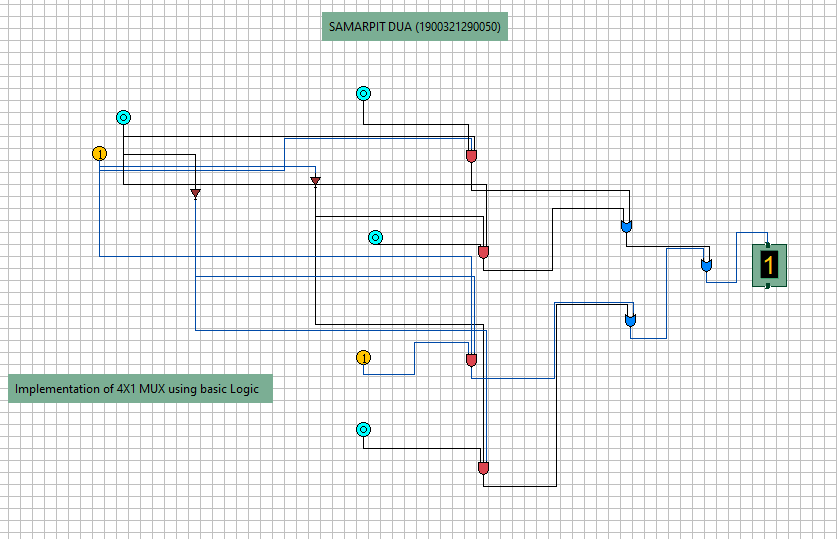


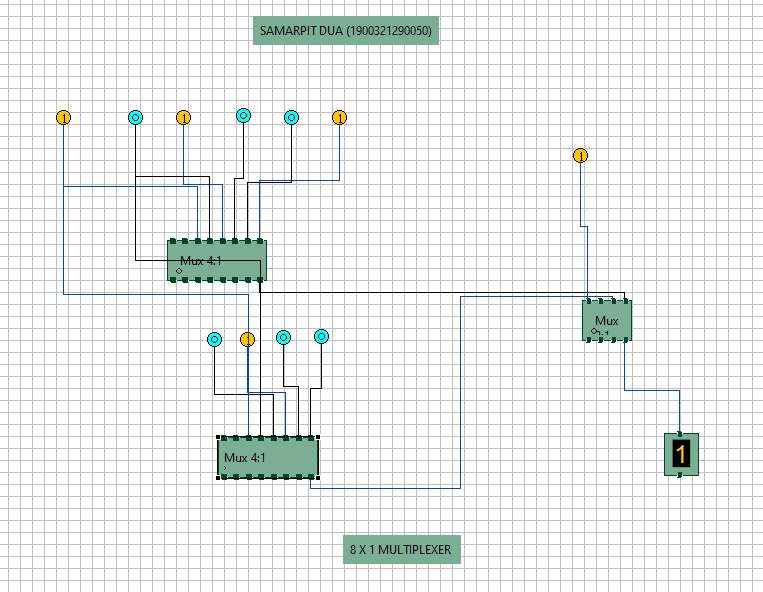
|  |  |  |  |
| --- | --- | --- | --- |
|  | **S1** | **S0** | **Y** |
| 0 | 0 | 0 | 10 |
| 0 | 0 | 1 | I1 |
| 0 | 1 | 0 | I2 |
| 0 | 1 | 1 | I3 |
| 1 | 0 | 0 | I4 |
| 1 | 0 | 1 | I5 |
| 1 | 1 | 0 | I6 |
| 1 | 1 | 1 | 17 |

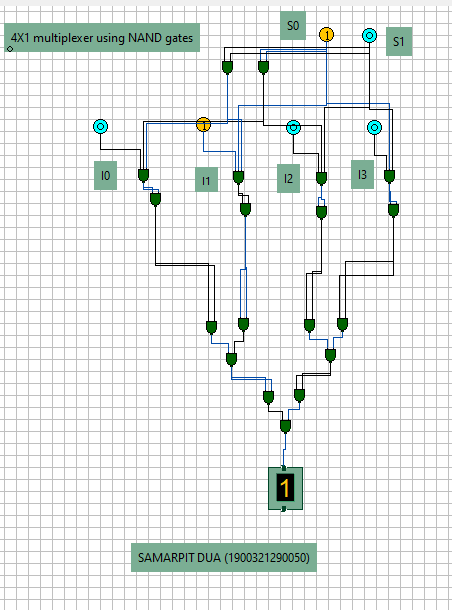
Function Table of 8\*1 Multiplexer.

**IMPLEMENTATION:**

****

****

****

****

Practical-6

**AIM:-** Verify the excitation tables of various 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

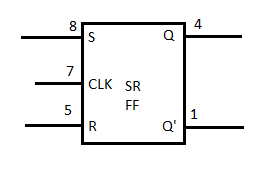
There are four Flip-Flops as follows-

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

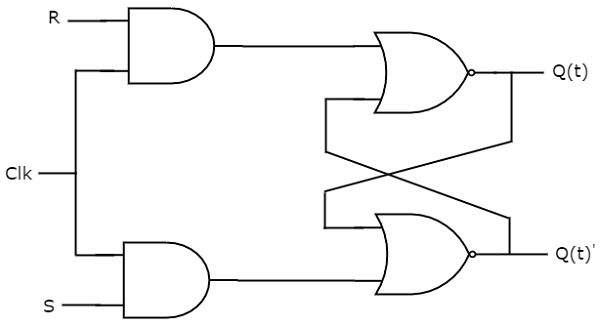
## SR Flip-Flop

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 | Qt |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | - |

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Present Inputs** | | **Present State** | **Next State** |
| **S** | **R** | **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 |

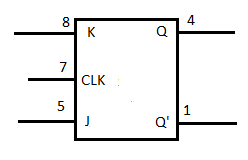
The **simplified expression** for next state Qt+1 is

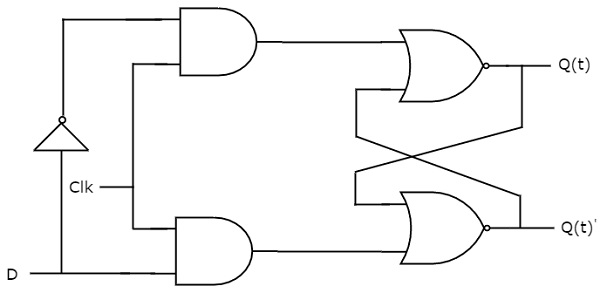
Q(t+1)=S+R′Q(t)Q(t+1)=S+R′Q(t)

## 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** | **Qt + 1t + 1** |
| 0 | 0 |
| 1 | 1 |

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

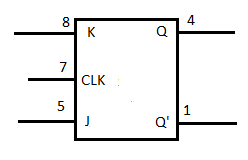
Qt+1t+1 = D

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.

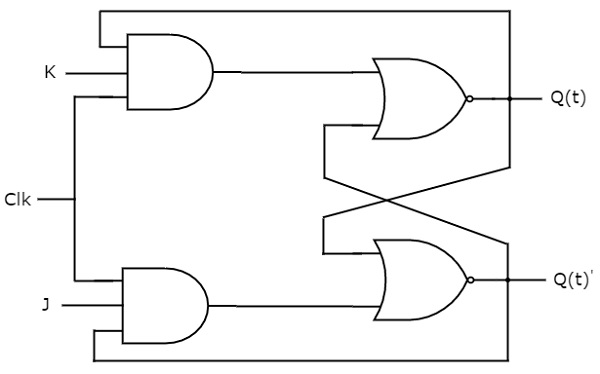
## JK Flip-Flop

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 Qt & Qt’. The operation of JK flip-flop is similar to SR flip-flop. Here, we considered the inputs of SR flip-flop as **S = J Q**t**’** and **R = KQ**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.

|  |  |  |
| --- | --- | --- |
| **J** | **K** | **Q**t+1 |
| 0 | 0 | Qt |
| 0 | 1 | 0 |
| 1 | 0 | 1 |
| 1 | 1 | Qt' |

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

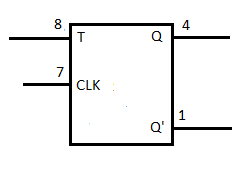
|  |  |  |  |
| --- | --- | --- | --- |
| **Present Inputs** | | **Present State** | **Next State** |
| **J** | **K** | **Q**t | **Q**t+1 |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

The **simplified expression** for next state Qt+1 is

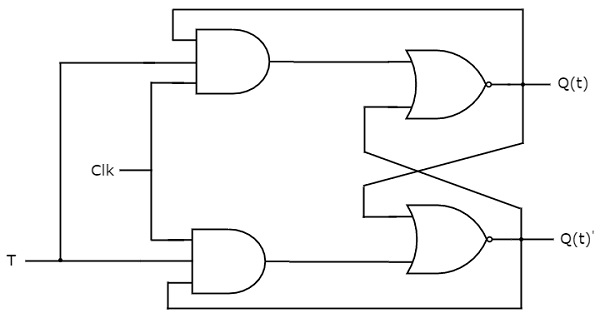
Q(t+1)=JQ(t)′+K′Q(t)Q(t+1)=JQ(t)′+K′Q(t)

## 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 Qtt &Qtt’. 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 | Qt |
| 1 | Qt’ |

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

|  |  |  |
| --- | --- | --- |
| **Inputs** | **Present State** | **Next State** |
| **T** | **Q**t | **Q**t+1 |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

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

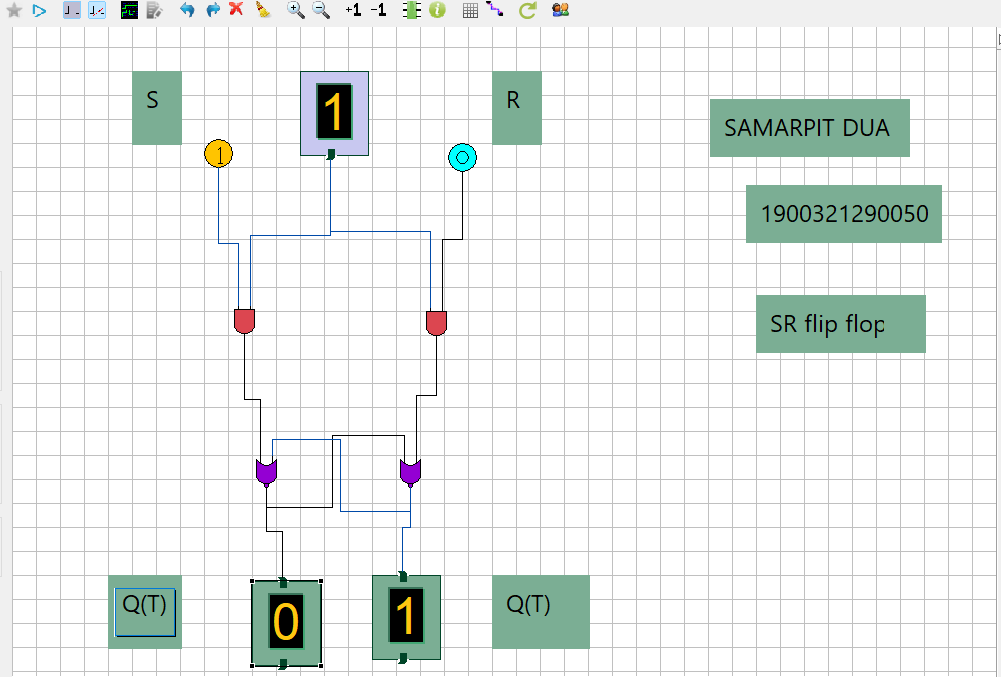
Q(t+1)=T′Q(t)+TQ(t)′Q(t+1)=T′Q(t)+TQ(t)′

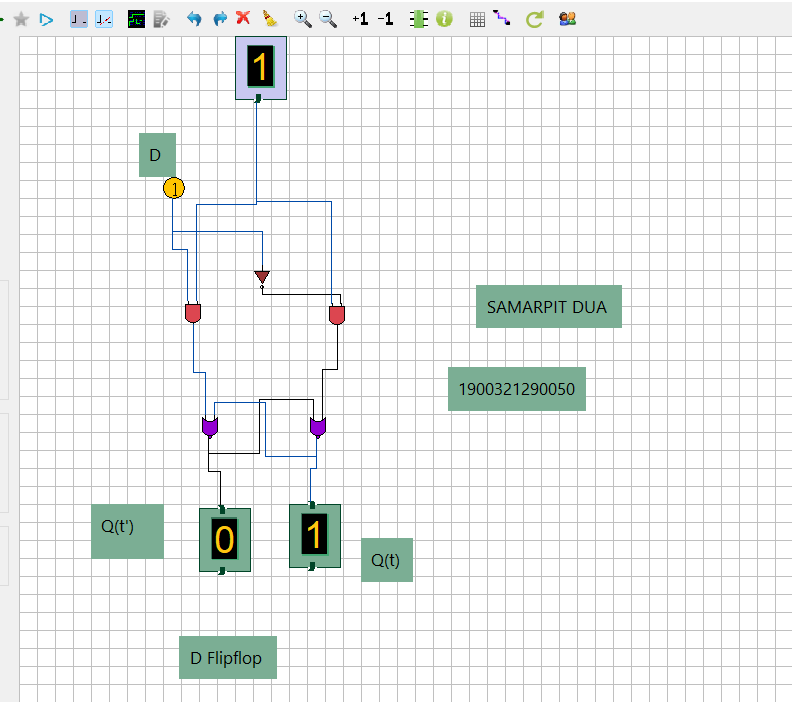
⇒Q(t+1)=T⊕Q(t)⇒Q(t+1)=T⊕Q(t)

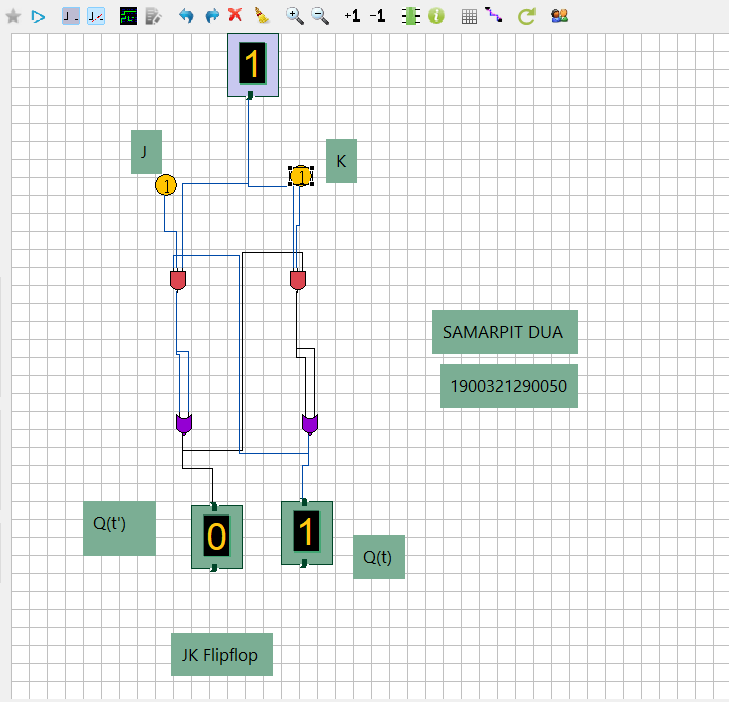
The output of T flip-flop always toggles for every positive transition of the clock signal, when input T remains at logic High 11. Hence, T flip-flop can be used in **counters**.

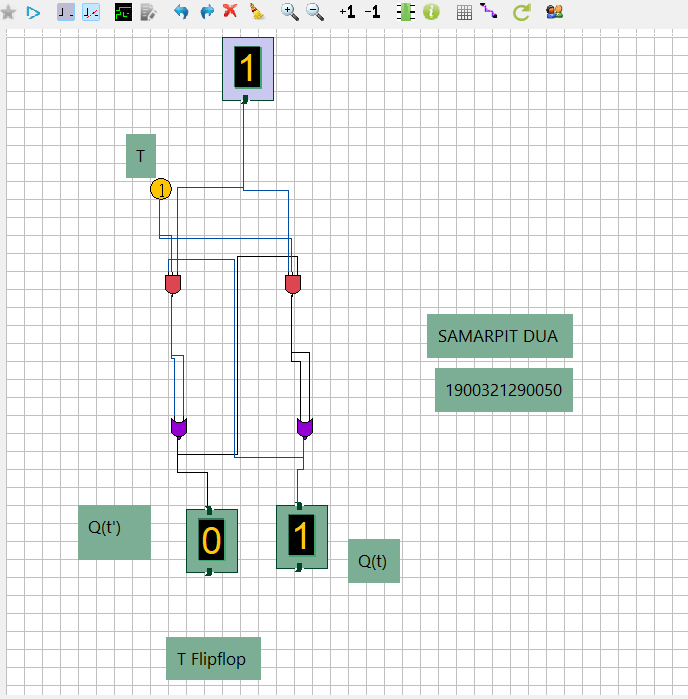
In this chapter, we implemented various flip-flops by providing the cross coupling between NOR gates. Similarly, you can implement these flip-flops by using NAND gates.

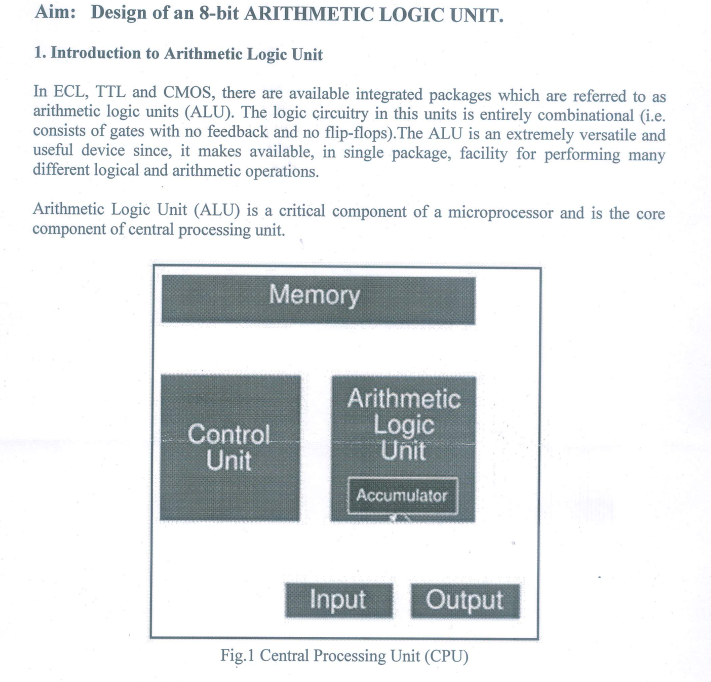
IMPLEMENTATION:



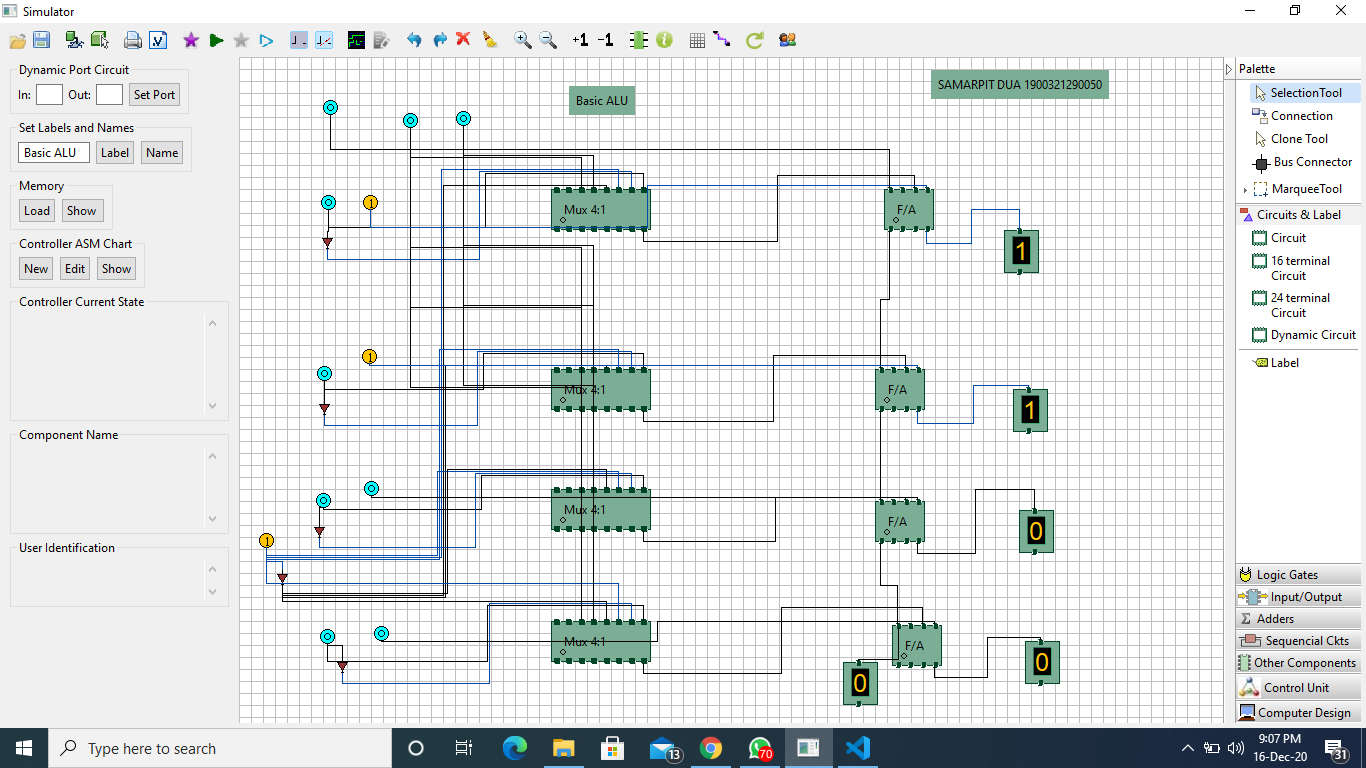
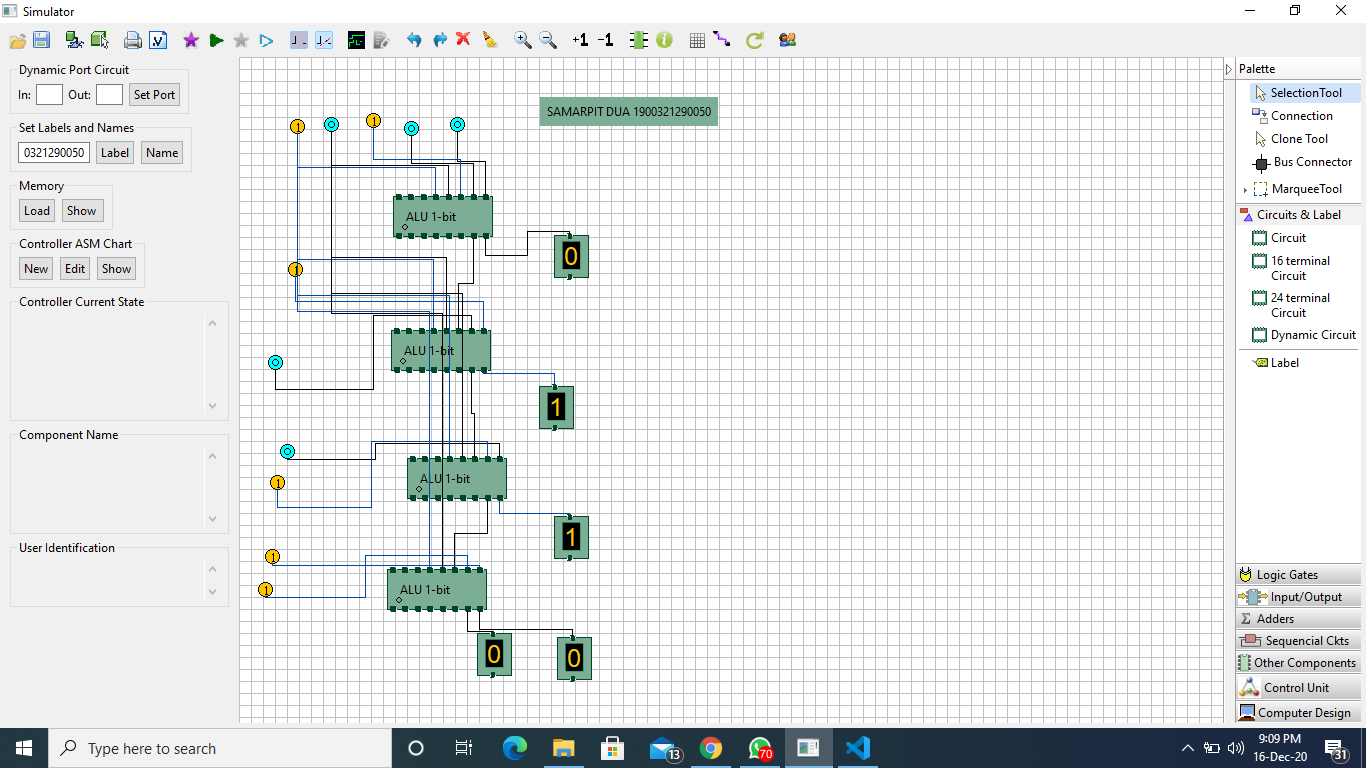






**Practical - 7**

Implementation :



**Practical -8**

**1. PRACTICAL STATEMENT OF PRACTICAL: To design a CPU using COA simulator.**

**2. THEORY:**

**At the top level, a computer consists of a CPU (central processing unit), memory, I/O components, with one or more modules of each type. These modules are interconnected in a specific manner to achieve the basic functionality of a computer i.e. executing programs. At the top level a computer system can be described as follows:**

**• describing the external behavior of each component i.e. the data and the control signals that ir exchanges with other components**

**• describing the interconnection structure and the controls required**

**We are considering the Von Neumann architecture. Some of the basic features of this architecture are as follows:**

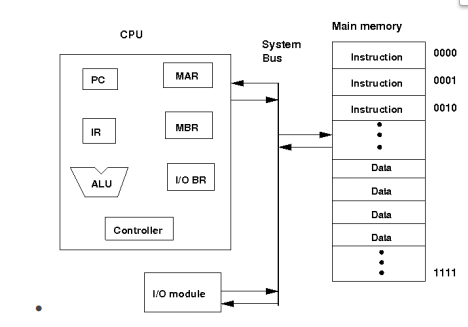
**• data and instructions are stored in a single read-write memory**

**• the contents of the memory are addressable by location**

**• execution occurs in a sequential manner (unless explicitly specified) from one instruction to the next**

**Top level components and interactions among them:**

**• CPU exchanges data with memory. For this CPU uses two internal resisters. Following is a block diagram of a basic computer system:**

****

**o Memory address register (MAR) which specifies the address for the next read or write**

**o Memory buffer register (MBR) which contains the data to be written into memory or receives the data read from memory**

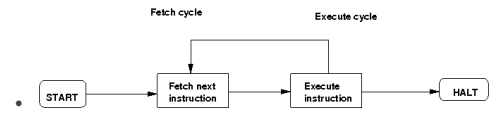
**o I/O buffer (I/O BR) register is used for the exchange of data between an I/O module and the CPU**

**• A memory module consists of a set of a locations defined by sequentially numbered addresses. Each location contains a binary number that cam be interpreted as either an instruction or data.**

**• An I/O module transfers data from external devices to CPU and memory and vice-versa**

**• The basic function of a computer is to execute a program which consists of a set of instructions stored in the memory. Processing required for a single instruction is called an instruction cycle which consist instruction fetch and instruction execute. A register called program counter (PC) holds the address of the next instruction. Unless told otherwise the processor always increments PC after each instruction fetch so that the next instruction is fetched in sequence. The fetched instruction is fetched into a register called instruction register (IR).**

**• The basic instruction cycle is shown in the following figure:**

****

**• After fetching an instruction, processor executes the instruction by doing some processing on the data which may involve arithmetic and logic unit (ALU), specified by the instruction, then processor writes back the result (if any) to the memory.**

**This experiment provides a single instruction CPU with built-in controller. A working memory has been provided to check the working principle of the CPU. The single instruction which this CPU supports is SBN (subtract and branch if negative). The format of this instruction is:**

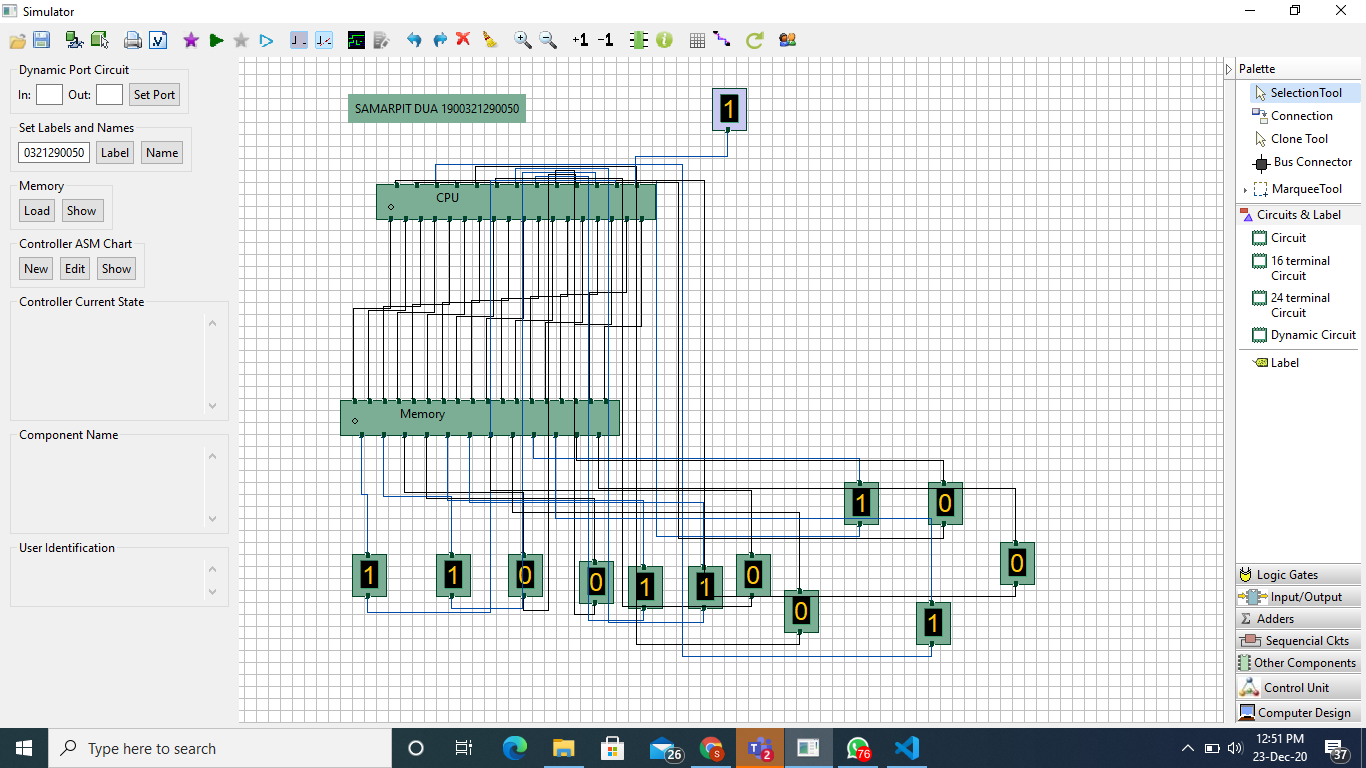
**• SBN a,b,c Mem[a] = Mem[a] - Mem[b] if(Mem[a] < 0) goto c**

**• a, b, c are 4 bit addresses**

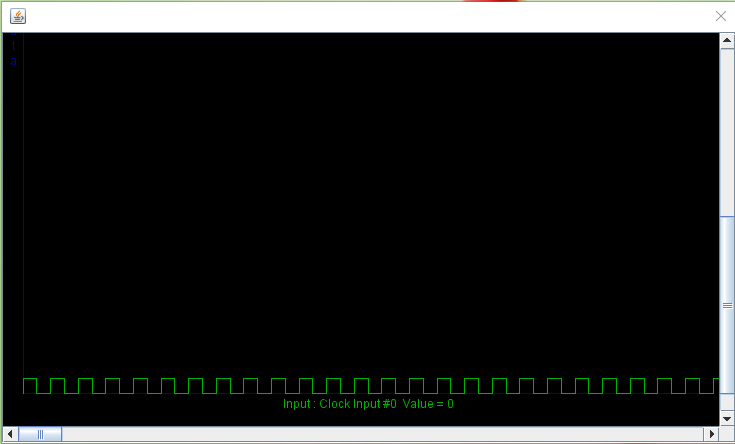
**• Mem[a] denotes the contents of memory address a**

**In this experiment, no I/O module has been included for the purpose of simplicity. The working memory should contain the program and data in binary format. The CPU executes the program. For halting, it uses self-loop, no other halt instruction is provided**

**Implementation :**

****

**Result:**

****