



Name :- Deep Salunkhe

Roll no. :- 21102A0014

Div:- A

Course :- Computer Engineering

Subject :- Digital Logic and Computer Architecture

College :- Vidyalankar Institute of Technology



Vidyalankar Institute of Technology
Wadala(E), Mumbai 400037

CERTIFICATE

This is to certify that

This Term Work in the subject of
Digital Logic and Computer Architecture

Of semester III of Computer Engineering
course (Academic year 22-23)

submitted by

Mr:- Deep Salunkhe

Roll no.- 21102A0014

is accepted by the department.

Professor In Charge

Head of the Department

VIDYALANKAR INSTITUTE OF TECHNOLOGY

Wadala, Mumbai – 400 037

INDEX

Sr. No.	Title of the Experiment	Page	Date	Signature	Remark
1	To study Logisim simulator	1	26/07/22		
2	To implement basic logic gates – AND, OR, NOT	6	26/07/22		
3	To implement Universal logic gates – NAND, NOR	10	23/08/22		
4	To implement XOR gate.	14	23/08/22		
5	To implement Half adder and Full adder	16	6/09/22		
6	To implement 4-bit ripple carry adder	20	6/09/22		
7	To implement Multiplexer and Demultiplexer	23	13/09/22		
8	To implement Decoder and Encoder	27	20/09/22		
9	To implement Flip Flops- SR, JK, D,T	29	27/09/22		
10	To implement Booth's algorithm for multiplication	33	11/10/22		
11	To implement Restoring algorithm for division	39	18/10/22		
12	To implement Non-Restoring algorithm for division	44	2/11/22		

Experiment No. 01

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – Deep Salunkhe

Roll Number– 21102A0014

Division and Batch– Division A, Batch

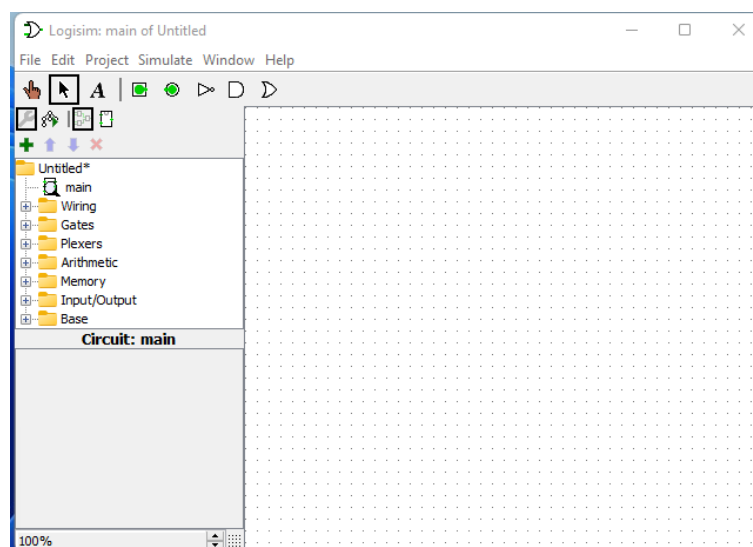
Date of Implementation– 26/07/2022

Experiment Title: To study Logisim simulator

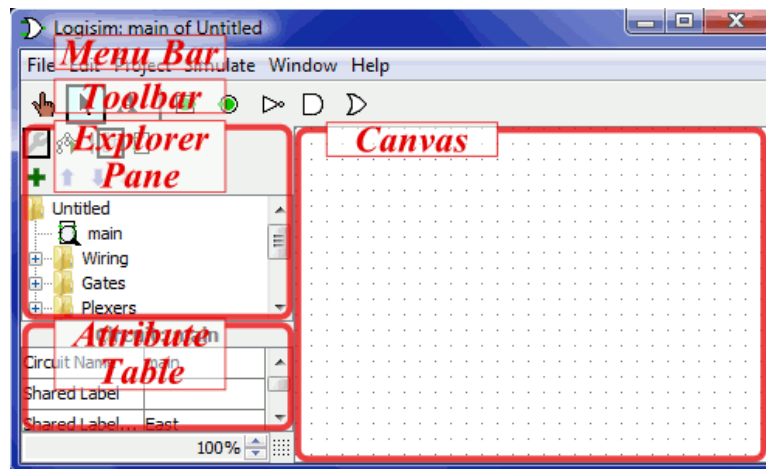
Theory:

Step 0: Orienting yourself

When you start Logisim, you'll see a window like the following. Some of the details may be slightly different since you're likely using a different system than mine.

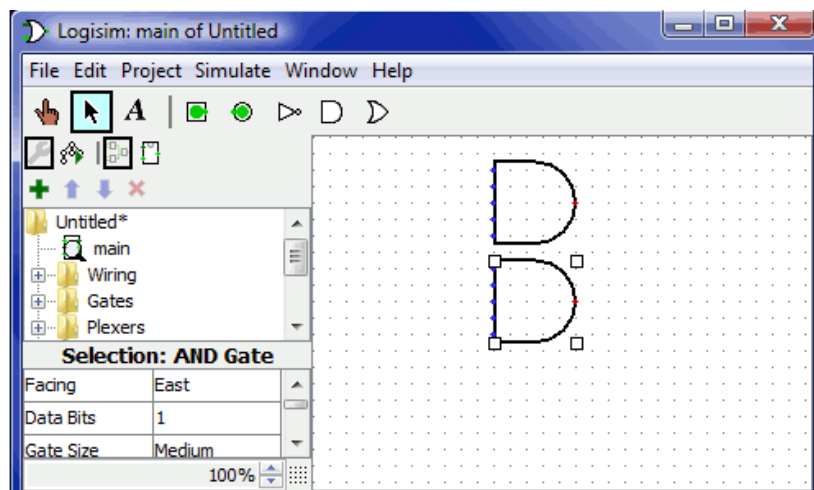


All Logisim is divided into three parts, called the *explorer pane*, the *attribute table*, and the *canvas*. Above these parts are the *menu bar* and the *toolbar*.

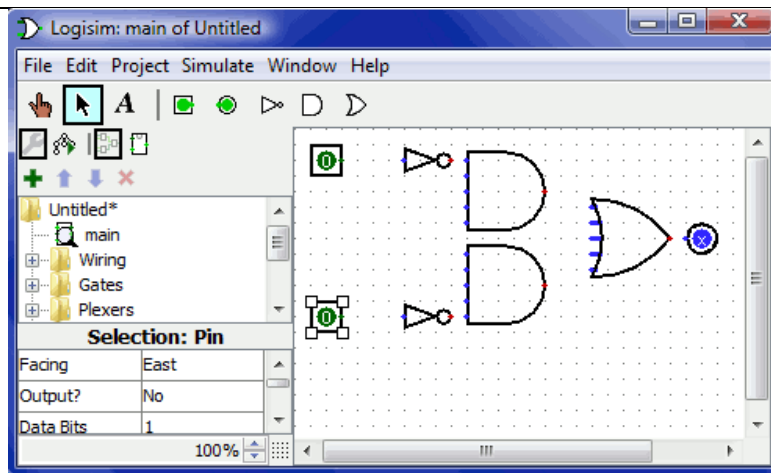


Step 1: Adding gates

Build a circuit by inserting the gates first as a sort of skeleton and then connecting them with wires later. The first thing we'll do is to add the two AND gates. Click on the AND tool in the toolbar (D, the next-to-last tool listed). Then click in the editing area where you want the first AND gate to go. Be sure to leave plenty of room for stuff on the left. Then click the AND tool again and place the second AND gate below it.



Notice the five dots on the left side of the AND gate. These are spots where wires can be attached. Now we want to add the two inputs x and y into the diagram. Select the Input tool (■), and place the pins down.

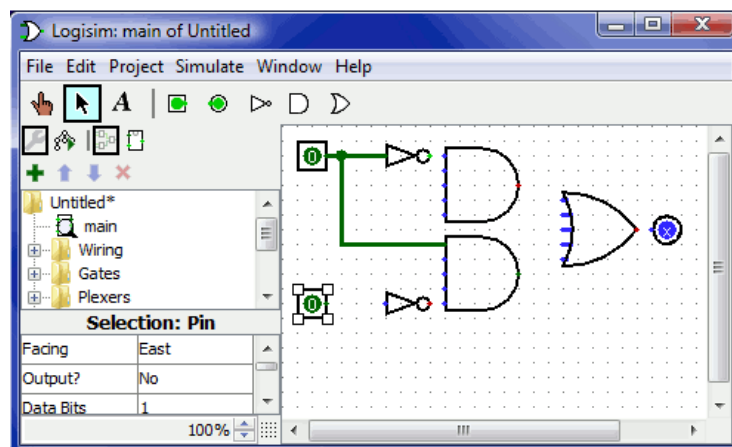


Step 2: Adding wires

After you have all the components blocked out on the canvas, you're ready to start adding wires. Select the Edit Tool (⌘). When the cursor is over a point that receives a wire, a small green circle will be drawn around it. Press the mouse button there and drag as far as you want the wire to go.


Logisim is rather intelligent when adding wires: Whenever a wire ends at another wire, Logisim automatically connects them. You can also "extend" or "shorten" a wire by dragging one of its endpoints using the edit tool.

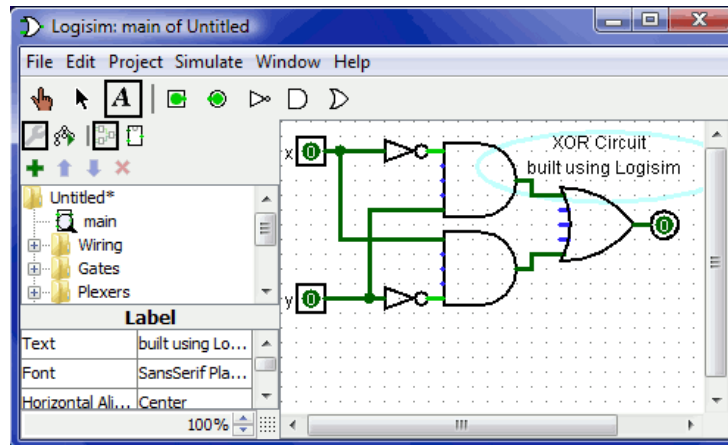
Wires in Logisim must be horizontal or vertical. To connect the upper input to the NOT gate and the AND gate, then, I added three different wires.



Logisim automatically connects wires to the gates and to each other. This includes automatically drawing the circle at a *T* intersection as above, indicating that the wires are connected.

Step 3: Adding text


Adding text to the circuit isn't necessary to make it work; but if you want to show your circuit to somebody (like a teacher), then some labels help to communicate the purpose of the different pieces of your circuit. Select the text tool () . You can click on an input pin and start typing to give it a label.

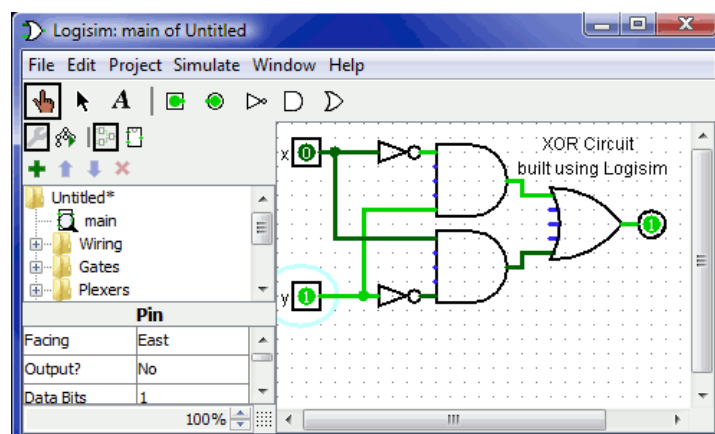


Step 4: Testing your circuit

Our final step is to test our circuit to ensure that it really does what we intended. Logisim is already simulating the circuit.

Note that the input pins both contain 0s; and so does the output pin. This already tells us that the circuit already computes a 0 when both inputs are 0.

Now to try another combination of inputs. Select the poke tool () and start poking the inputs by clicking on them. Each time you poke an input, its value will toggle. For example, we might first poke the bottom input.



So far, we have tested the first two rows of our truth table, and the outputs (0 and 1) match the desired outputs.

a	b	x
0	0	0
0	1	1
1	0	1
1	1	0

Implementation: The above all steps of Logisim are practised.

Conclusion:-

Logisim allows you to design and simulate digital circuits. It is intended as an educational tool, to help you learn how circuits work. Logisim helps us to get familiar with working of logic gates virtually. By following simple steps, one can build complex circuits easily and efficiently.

Experiment No. 02

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name– Deep Salunkhe

Roll Number– 21102A0014

Division and Batch– Division A, Batch

Date of Implementation– 26/07/2022

Experiment Title: To implement basic logic gates – AND, OR, NOT

Theory:

AND Gate

An AND gate is a logic gate having two or more inputs and a single output. An AND gate operates on logical multiplication rules. In this gate, if either of the inputs is low (0), then the output is also low. If all the inputs are high (1), then the output will also be high.

OR Gate

An OR gate is a logical gate that produces inclusive disjunction. The function of an OR gate is to find the maximum between the inputs which are binary in nature. It is one of the basic gates used in Boolean algebra and electronic circuits like transistor-transistor logic, and complementary metal-oxide semiconductors make use of it.

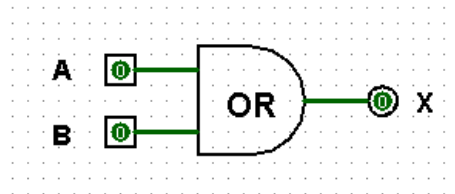
NOT Gate

The **NOT gate** is a single input single output gate. This gate is also known as Inverter because it performs the inversion of the applied binary signal, i.e., it converts 0 into 1 or 1 into 0. In other words, the gate which has a high input signal only when their input signal is low such type of gate is known as the not gate.

Implementation :-

AND Gate-

Circuit

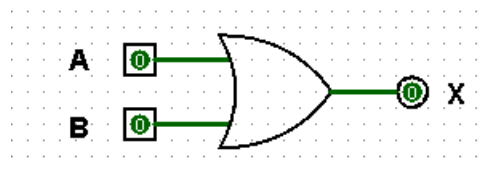


Truth Table

a	b	x
0	0	0
0	1	0
1	0	0
1	1	1

OR Gate-

Circuit

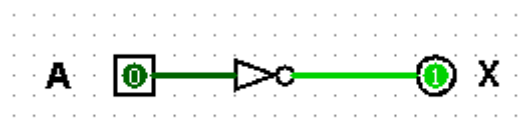


Truth Table

a	b	x
0	0	0
0	1	1
1	0	1
1	1	1

NOT Gate-

Circuit



Truth Table

a	x
0	1
1	0

Conclusion:

In this experiment we understood the working of AND, OR and NOT gates.

AND gate- If any input value in the AND gate is set to 0, then it will always return low output (0).

OR gate- If any input value in the OR gate is set to 1, then it will always return High output (1).

NOT gate- If any input value in the NOT gate is set to 0, then it will always return 1. And vice versa

Experiment No. 03

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivasa
Assisting Teachers	Prof. Avinash Shrivasa

Student Name– Deep Salunkhe

Roll Number– 21102A0014

Division and Batch– Division A, Batch

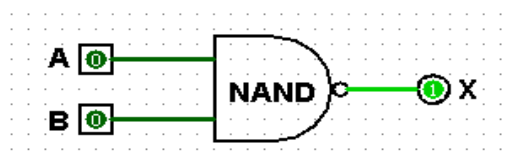
Date of Implementation– 23/08/2022

Experiment Title: To implement Universal logic gates – NAND, NOR

Theory:

NAND Gate

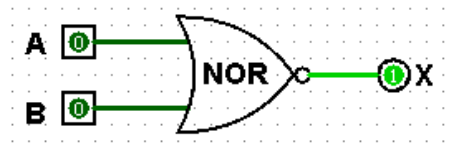
A **NAND gate (NOT-AND)** is a logic gate which produces an output which is false only if all its inputs are true; thus, its output is complement to that of an AND gate. A LOW (0) output results only if all the inputs to the gate are HIGH (1); if any input is LOW (0), a HIGH (1) output results. A NAND gate is made using transistors and junction diodes. By De Morgan's Law, a two-input NAND gate's logic may be expressed as $A \bullet B = A + B$.



a	b	x
0	0	1
0	1	1
1	0	1
1	1	0

NOR Gate

The **NOR gate** is a digital logic gate that implements logical NOR. A HIGH output (1) results if both the inputs to the gate are LOW (0); if one or both input is HIGH (1), a LOW output (0) results. NOR is the result of the negation of the OR operator. It can also in some senses be seen as the inverse of an AND gate.

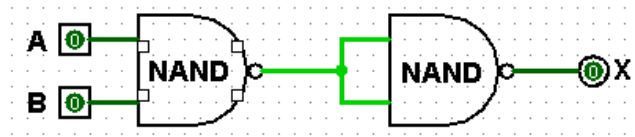


a	b	x
0	0	1
0	1	0
1	0	0
1	1	0

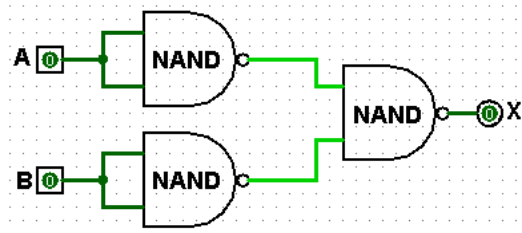
Implementation

NAND Gate-

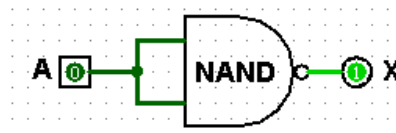
- AND using NAND-



- OR using NAND-

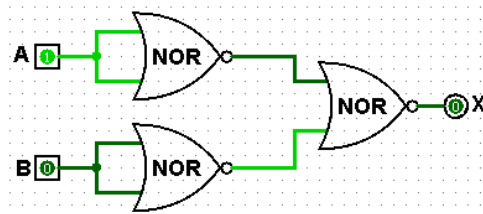


- NOT using NAND-

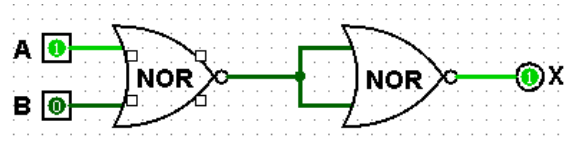


NOR Gate-

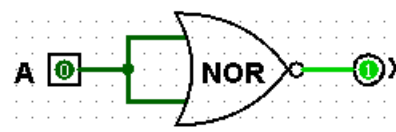
- AND using NOR-



- OR using NOR-



- NOT using NOR-



Conclusion:

In this experiment we understood the working of NAND and NOR gates.

NAND gate- If any input value in the NAND gate is set to 0, then it will always return High output (1).

NOR gate- If any input value in the NOR gate is set to 1, then it will always return Low output (0).

Experiment No. 04

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number – 21102A0014

Division and Batch – Division A, Batch

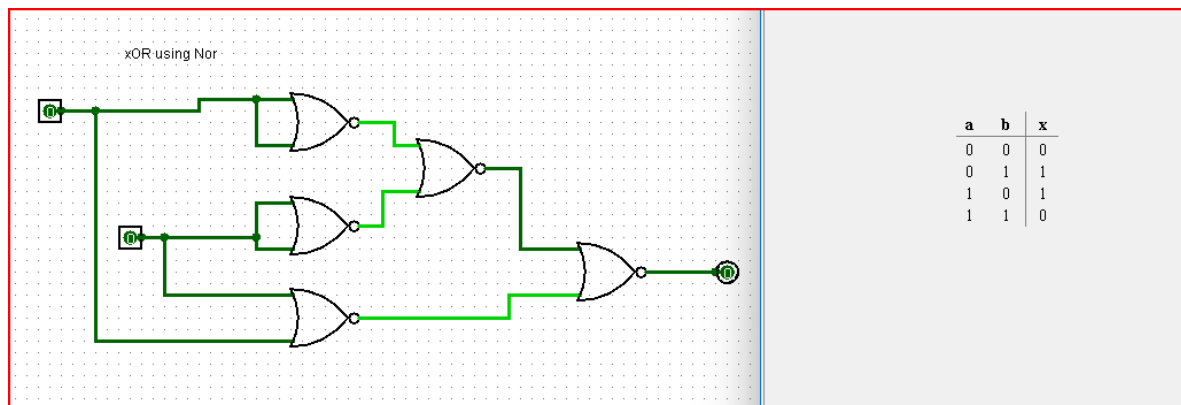
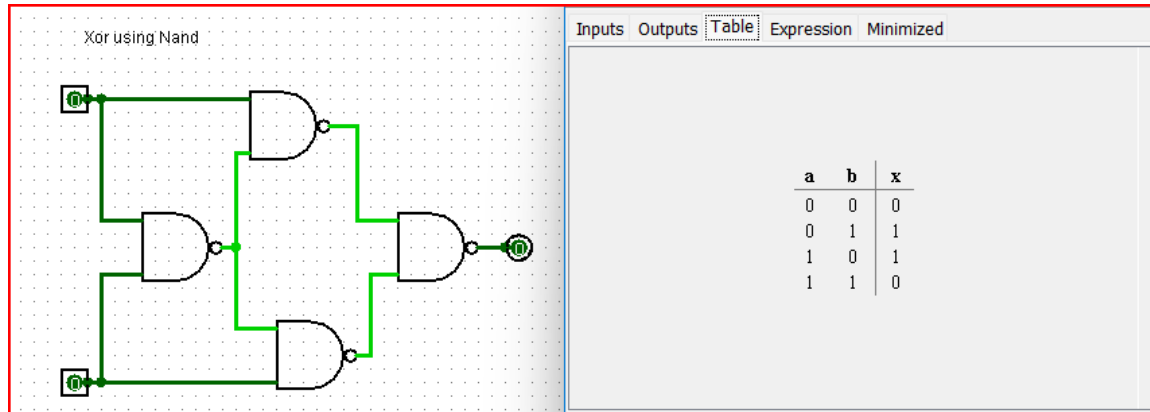
Date of Implementation – 23/08/2022

Experiment Title: To implement XOR gate.

Theory:

NAND and NOR gates are also known as universal gates as all the gates can be recreated using these gates. Here, we have created XOR Gates.

Implementation:-



Conclusion:

In this experiment we understood the working of XOR gate using NAND & NOR.

Experiment No. 05

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number –21102A0014

Division and Batch – Division A, Batch

Date of Implementation – 6/09/22

Experiment Title:- To implement Half adder and Full adder

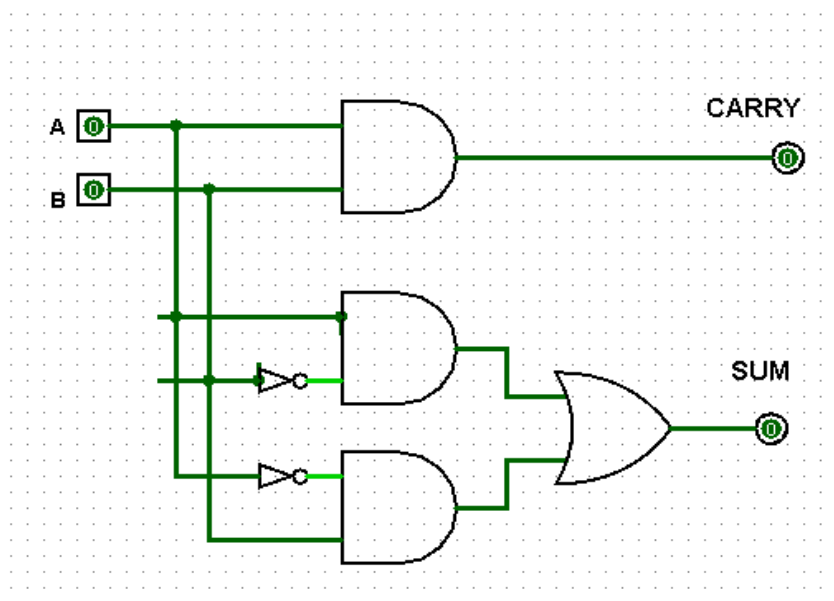
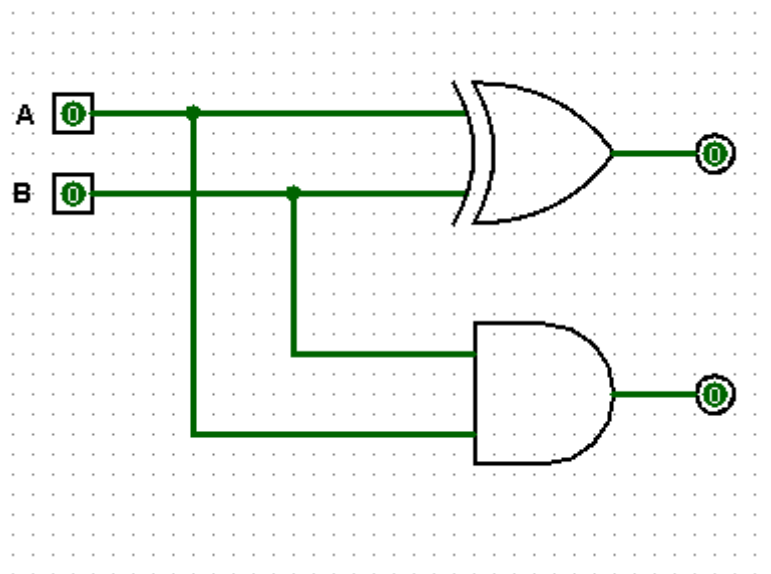
HALF ADDER CIRCUIT:

The half adder adds two single binary digits A and B . It has two outputs, sum (S) and carry (C). The carry signal represents an into the next digit of a multi-digit addition. The value of the sum is $2C + S$. The simplest half-adder design, pictured on the right, incorporates an XOR gate for S and an AND gate for C . The Boolean logic for the sum will be $A'B + AB'$ whereas for the carry (C) will be AB . With the addition of an OR gate to combine their carry outputs, two half adders can be combined to make a full adder.^[2] The half adder adds two input bits and generates a carry and sum, which are the two outputs of a half adder. The output variables are the sum and carry.

FULL ADDER CIRCUIT:

A full adder can also be constructed from two half adders by connecting A and B to the input of one-half adder, then taking its sum-output S as one of the inputs to the second half adder and C_{in} as its other input, and finally the carry outputs from the two half-adders are connected to an OR gate. The sum-output from the second half adder is the final sum output (S) of the full adder and the output from the OR gate is the final carry output (C_{out}).

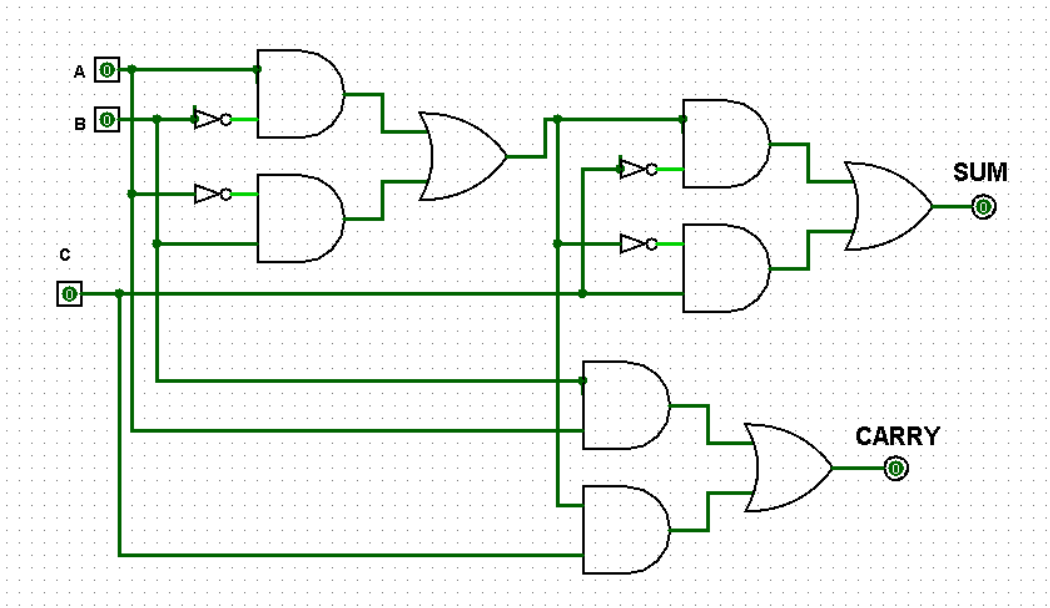
HALF ADDER CIRCUIT



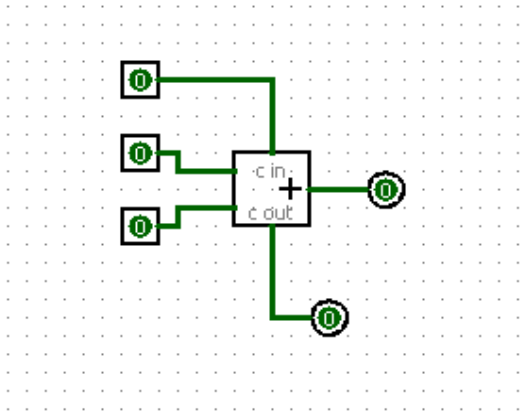
HALF ADDER TRUTH TABLE

a	b	x	y
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

FULL ADDER CIRCUIT USING TWO HALF ADDER



ONE BIT FULL ADDER



FULL ADDER TRUTH TABLE

a	b	c	x	y
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

Conclusion:-

In half adder is used to add two single-digit binary numbers and results into a two-digit output. It is named as such because putting two half adders together with the use of an OR gate results in a full adder.

Experiment No. 06

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number –21102A0014

Division and Batch – Division A, Batch

Date of Implementation – 6/09/22

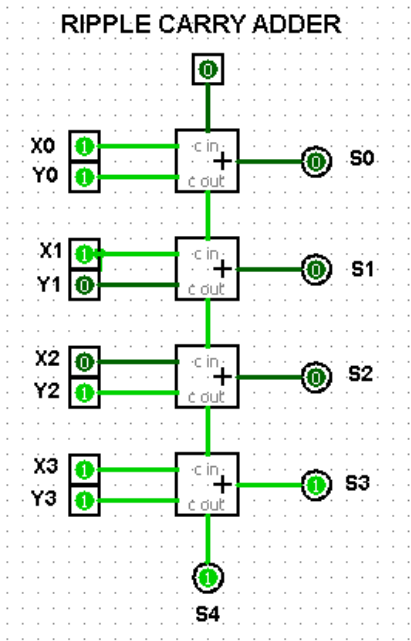
Experiment Title :-To implement 4-bit ripple carry adder

THEORY:-

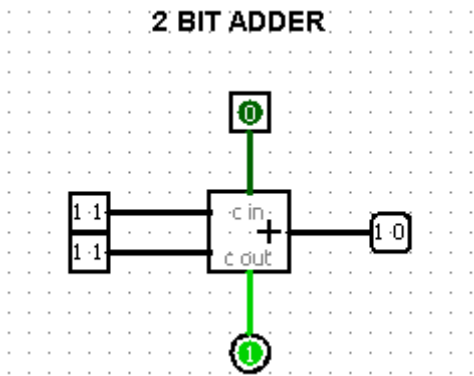
RIPPLE CARRY ADDER:

Multiple full adder circuits can be cascaded in parallel to add an N-bit number. For an N- bit parallel adder, there must be N number of full adder circuits. A ripple carry adder is a logic circuit in which the carry-out of each full adder is the carry in of the succeeding next most significant full adder. It is called a ripple carry adder because each carry bit gets rippled into the next stage. In a ripple carry adder the sum and carry out bits of any half adder stage is not valid until the carry in of that stage occurs.

RIPPLE CARRY ADDER CIRCUIT

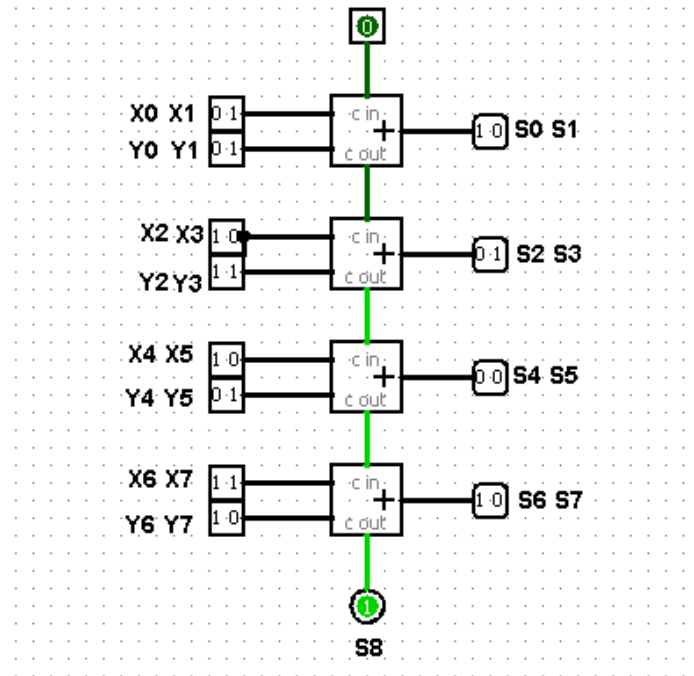


2 BIT ADDER CIRCUIT



8 BIT RIPPLE CARRY ADDER

8 BIT RIPPLE CARRY ADDER USING 2 BIT ADDER



Conclusion:-

Ripple Carry Adder is combination of multiple full adders. The 2-bit adder adds 2-bit data and gives a 2-bit sum and 1-bit carry. The 8-bit ripple carry adder can be made by using 4 2-bit adders.

Experiment No. 07

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number – 21102A0014

Division and Batch – Division A, Batch

Date of Implementation – 13/09/2022

Experiment Title :- To implement Multiplexer and Demultiplexer

Theory:

Multiplexer

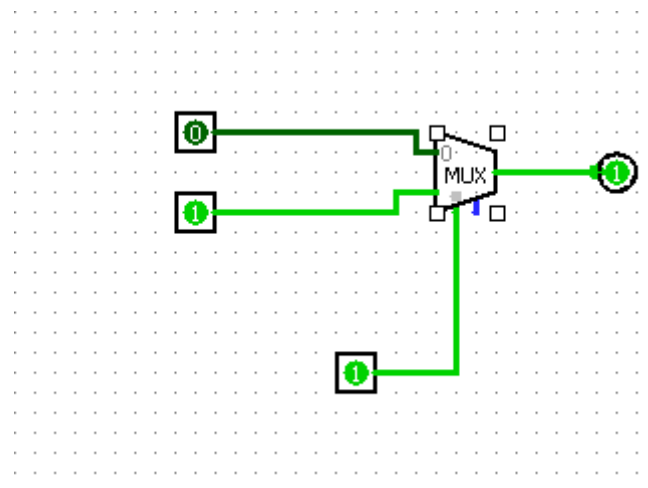
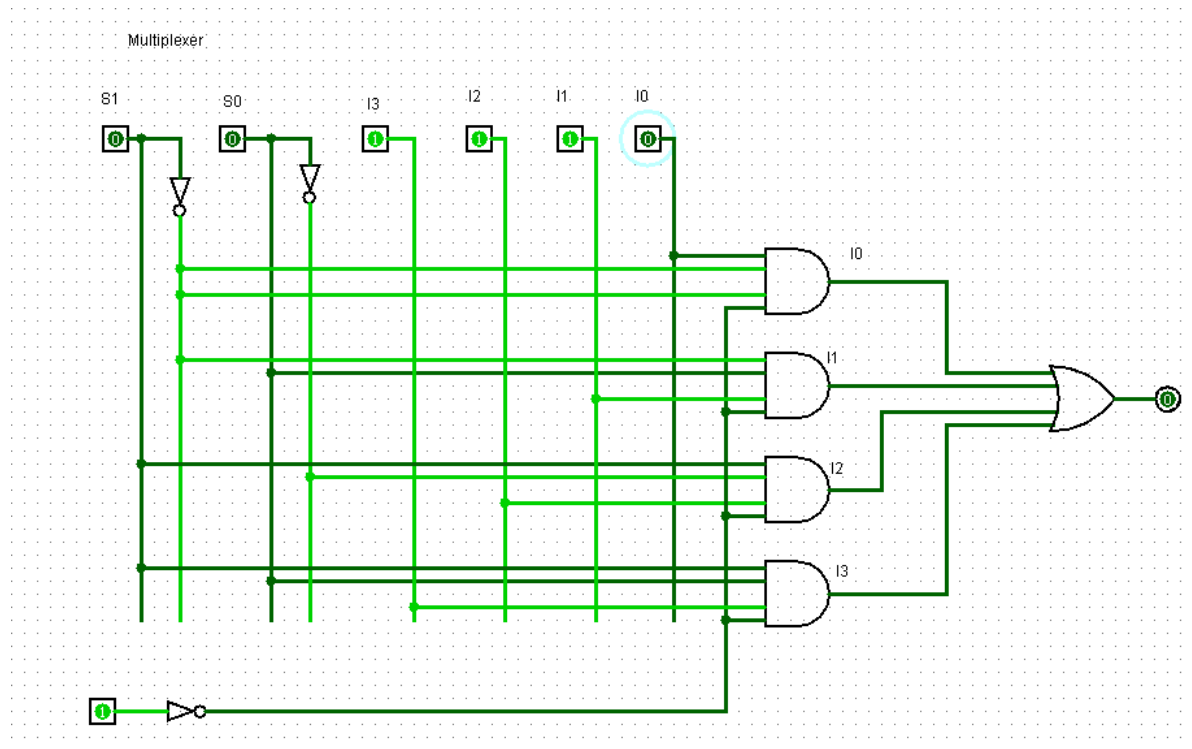
A multiplexer, also known as a data selector, is a device that selects between several analog or digital input signals and forwards the selected input to a single output line. The selection is directed by a separate set of digital inputs known as select lines.

Demultiplexer

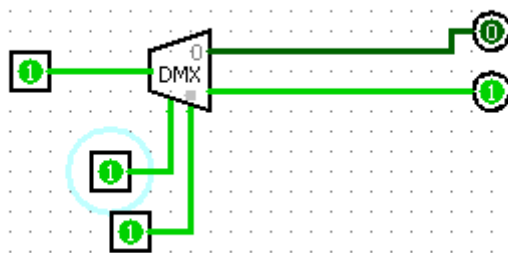
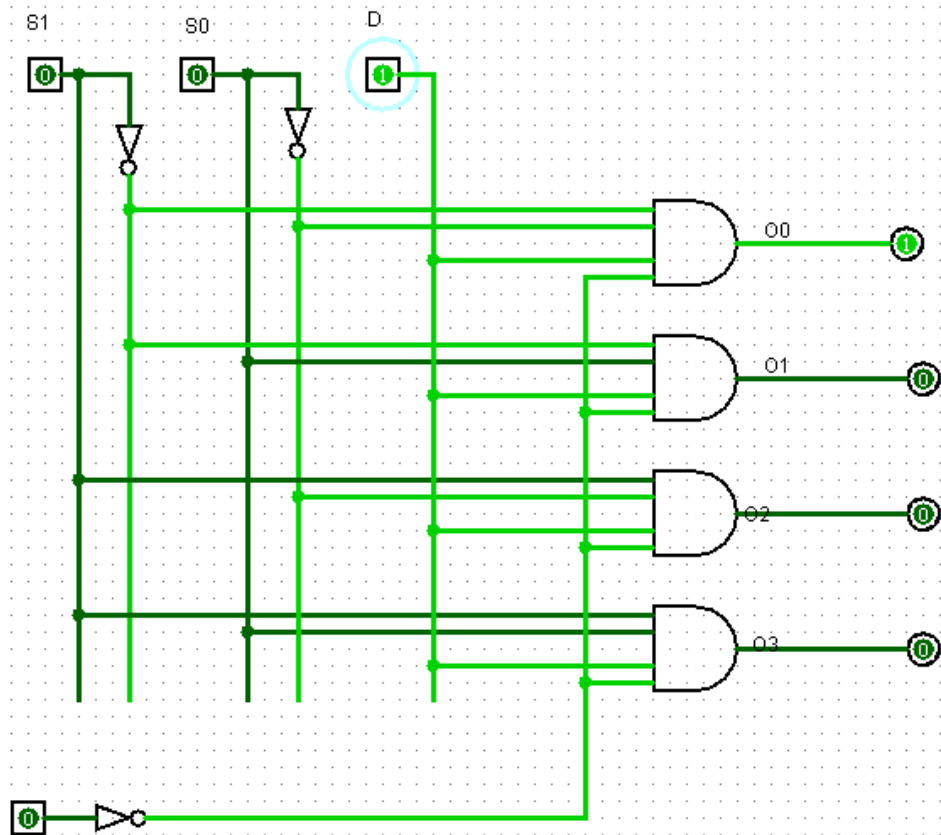
A demultiplexer (also known as a demux or data distributor) is defined as **a circuit that can distribute or deliver multiple outputs from a single input**. A demultiplexer can perform as a single input with many output switches.

Implementation:-

Multiplexer



Demultiplexer



Conclusion:-

In this experiment we understood the working of Multiplexer and Demultiplexer.
It follows the combinational logic type.

Experiment No. 08

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number – 21102A0014

Division and Batch – Division A, Batch

Date of Implementation – 20/09/2022

Experiment Title: To implement Decoder and Encoder

Theory:

Decoder

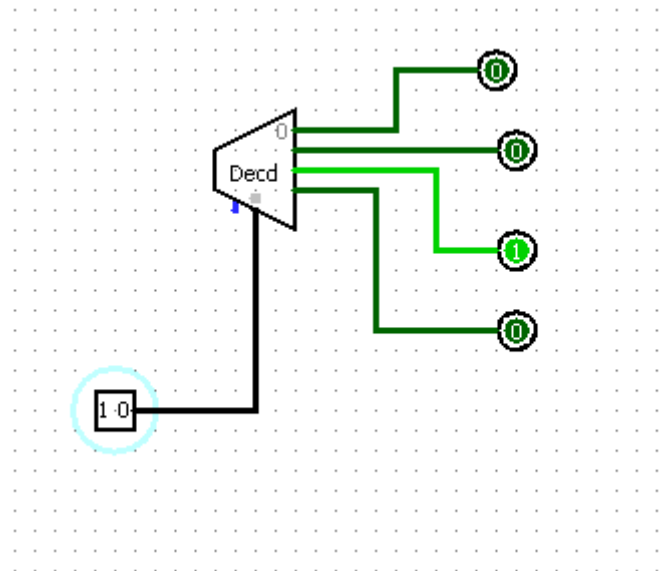
In digital electronics, a binary decoder is a combinational logic circuit that converts binary information from the n coded inputs to a maximum of 2^n unique outputs

Encoder

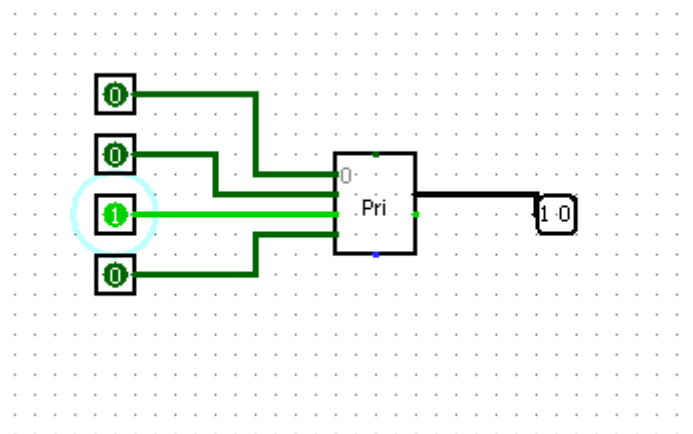
An encoder in digital electronics is a one-hot to binary converter. That is, if there are 2^n input lines, and at most only one of them will ever be high, the binary code of this 'hot' line is produced on the n -bit output lines. A binary encoder is the dual of a binary decoder

Implementation :-

Decoder



Encoder



Conclusion:-

In this experiment we understood the working of encoder & decoder.

Decoder is n input and 2^n output device. The input combination decides only one output to be activated out of all.

Experiment No. 09

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number – 21102A0014

Division and Batch – Division A, Batch

Date of Implementation -27/09/22

Experiment Title: To implement Flip Flops- SR, JK, D, T

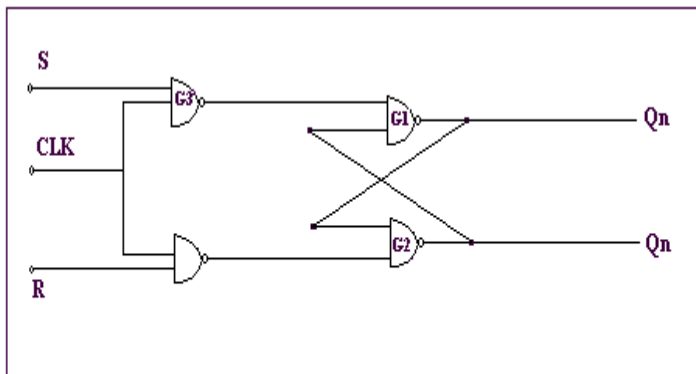
THEORY :-

The first electronic flip-flop was invented in 1918 by William Eccles and F. W. Jordan. It was initially called the Eccles–Jordan trigger circuit and consisted of two active elements (vacuum tubes). Such circuits and their transistorized versions were common in computers even after the introduction of integrated circuits, though flip-flops made from logic gates are also common now. Early flip-flops were known variously as trigger circuits or multivibrators.

Procedure:

- 1) Connect the circuit for SR Flip Flop as shown in the diagram. Verify the truth table of SR flip flop.
- 2) Convert SR Flip flop to D flip flop as shown in the diagram. Verify the Truth table.
- 3) Connect the diagram of JK Flip flop using Nand gates. Verify the truth table.
- 4) Convert JK Flip flop to T flip flop. Verify the truth table.

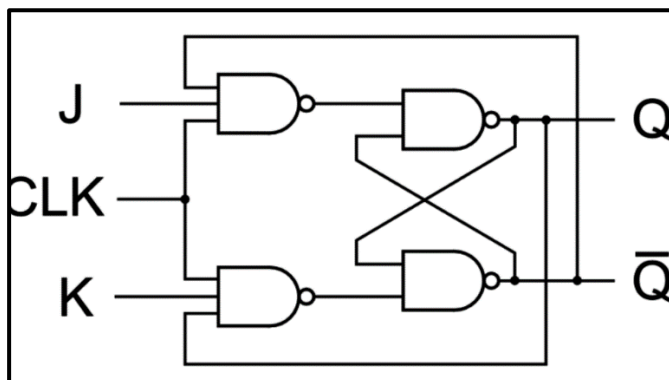
Diagram:



SR Flip Flop:

INPUT		OUTPUT
S _n	R _n	Q _{n+1}
0	0	Q _n
0	1	1
1	0	0
1	1	Forbidden state

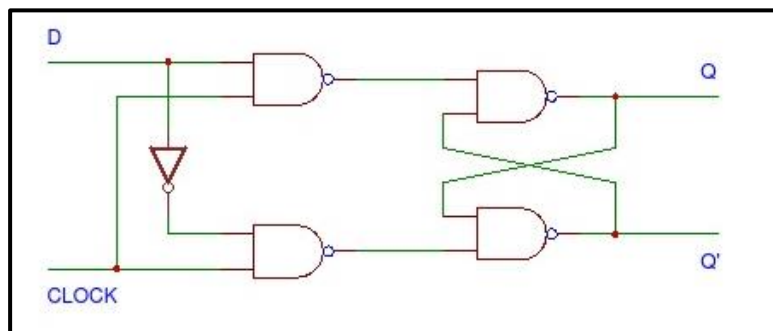
JK Flip Flop:-



JK Flip Flop:-

INPUT		OUTPUT
J _n	K _n	Q _{n+1}
0	0	Q _n
0	1	1
1	0	0
1	1	Q _n

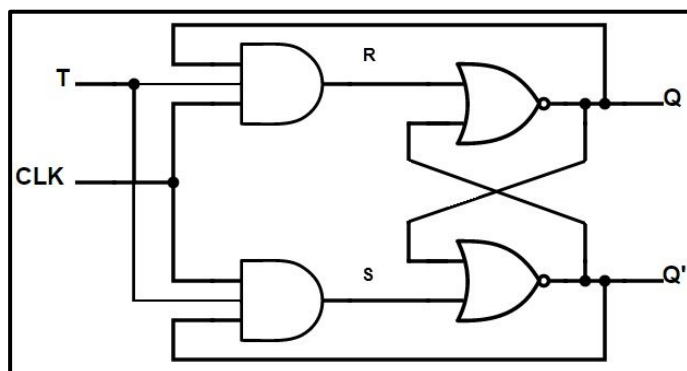
D Flip Flop:-



D Flip Flop:-

D	Q _{n+1}
0	0
1	1

T Flip Flop:-



T Flip Flop:-

T	Q_{n+1}
0	Q_n
1	Q_n^*

Conclusion:-

Working and analysis of various flip flops is done and the results are verified experimentally. Conversion of one flip flop to other is performed and its results are verified.

Experiment No. 10

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number – 21102A0014

Division and Batch – Division A, Batch

Date of Implementation – 11/10/2022

Experiment Title: To implement Booth's algorithm for multiplication

Theory:

Booth's algorithm is a multiplication algorithm that multiplies two signed binary numbers in 2's complement notation.

Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed. Booth's algorithm is of interest in the study of computer architecture. Here's the implementation of the algorithm.

Implementation:-

```
#include <stdio.h>

#include <math.h>

int a = 0, b = 0, c = 0, a1 = 0, b1 = 0, com[5] = { 1, 0, 0, 0, 0};
int anum[5] = {0}, anumcp[5] = {0}, bnum[5] = {0};
int acomp[5] = {0}, bcomp[5] = {0}, pro[5] = {0}, res[5] = {0};

void binary(){
    a1 = fabs(a);
    b1 = fabs(b);
    int r, r2, i, temp;
    for (i = 0; i < 5; i++){
        r = a1 % 2;
        a1 = a1 / 2;
        r2 = b1 % 2;
        b1 = b1 / 2;
        anum[i] = r;
        anumcp[i] = r;
        bnum[i] = r2;
        if(r2 == 0){
            bcomp[i] = 1;
        }
        if(r == 0){
            acomp[i] = 1;
        }
    }
    //part for two's complementing
    c = 0;
    for (i = 0; i < 5; i++){
        res[i] = com[i] + bcomp[i] + c;
        if(res[i] >= 2){
            c = 1;
        }
        else
            c = 0;
        res[i] = res[i] % 2;
    }
    for (i = 4; i >= 0; i--){
        bcomp[i] = res[i];
    }
    //in case of negative inputs
    if (a < 0){
        c = 0;
        for (i = 4; i >= 0; i--){
            res[i] = 0;
        }
    }
}
```

```

        for ( i = 0; i < 5; i++){
            res[i] = com[i] + acomp[i] + c;
            if (res[i] >= 2){
                c = 1;
            }
            else
                c = 0;
            res[i] = res[i]%2;
        }
        for (i = 4; i >= 0; i--){
            anum[i] = res[i];
            anumcp[i] = res[i];
        }
    }
    if(b < 0){
        for (i = 0; i < 5; i++){
            temp = bnum[i];
            bnum[i] = bcomp[i];
            bcomp[i] = temp;
        }
    }
}

void add(int num[]){
    int i;
    c = 0;
    for ( i = 0; i < 5; i++){
        res[i] = pro[i] + num[i] + c;
        if (res[i] >= 2){
            c = 1;
        }
        else{
            c = 0;
        }
        res[i] = res[i]%2;
    }
    for (i = 4; i >= 0; i--){
        pro[i] = res[i];
        printf("%d",pro[i]);
    }
    printf(":");
    for (i = 4; i >= 0; i--){
        printf("%d", anumcp[i]);
    }
}

void arshift(){//for arithmetic shift right
    int temp = pro[4], temp2 = pro[0], i;
    for (i = 1; i < 5 ; i++){//shift the MSB of product

```

```

        pro[i-1] = pro[i];
    }
    pro[4] = temp;
    for (i = 1; i < 5 ; i++){//shift the LSB of product
        anumcp[i-1] = anumcp[i];
    }
    anumcp[4] = temp2;
    printf("\nAR-SHIFT: ");//display together
    for (i = 4; i >= 0; i--){
        printf("%d",pro[i]);
    }
    printf(":");
    for(i = 4; i >= 0; i--){
        printf("%d", anumcp[i]);
    }
}

void main(){
    int i, q = 0;
    printf("\t\tBOOTH'S MULTIPLICATION ALGORITHM");
    printf("\nEnter two numbers to multiply: ");
    printf("\nBoth must be less than 16");
    //simulating for two numbers each below 16
    do{
        printf("\nEnter A: ");
        scanf("%d",&a);
        printf("Enter B: ");
        scanf("%d", &b);
    }while(a >=16 || b >=16);

    printf("\nExpected product = %d", a * b);
    binary();
    printf("\n\nBinary Equivalents are: ");
    printf("\nA = ");
    for (i = 4; i >= 0; i--){
        printf("%d", anum[i]);
    }
    printf("\nB = ");
    for (i = 4; i >= 0; i--){
        printf("%d", bnum[i]);
    }
    printf("\nB'+ 1 = ");
    for (i = 4; i >= 0; i--){
        printf("%d", bcomp[i]);
    }
    printf("\n\n");
    for (i = 0; i < 5; i++){
        if (anum[i] == q){//just shift for 00 or 11

```

```

        printf("\n-->");
        arshift();
        q = anum[i];
    }
    else if(anum[i] == 1 && q == 0){//subtract and shift for 10
        printf("\n-->");
        printf("\nSUB B: ");
        add(bcomp);//add two's complement to implement subtraction
        arshift();
        q = anum[i];
    }
    else{//add ans shift for 01
        printf("\n-->");
        printf("\nADD B: ");
        add(bnum);
        arshift();
        q = anum[i];
    }
}

printf("\nProduct is = ");
for (i = 4; i >= 0; i--){
    printf("%d", pro[i]);
}
for (i = 4; i >= 0; i--){
    printf("%d", anumcp[i]);
}
}

```


Output:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS E:\GIT> cd "e:\GIT\" ; if ($?) { gcc tempCodeRunnerFile.c -o tempCodeRunnerFile } ; if ($?) { .\tempCodeRunnerFile }
      BOOTH'S MULTIPLICATION ALGORITHM
Enter two numbers to multiply:
Both must be less than 16
Enter A: 4
Enter B: 5

Expected product = 20

-->
AR-SHIFT: 0000:00010
-->
AR-SHIFT: 0000:00001
-->
SUB B: 11011:00001
AR-SHIFT: 11101:10000
-->
ADD B: 00010:10000
AR-SHIFT: 00001:01000
-->
AR-SHIFT: 00000:10100
Product is = 0000010100
```

Conclusion:-

Booth's Multiplier can handle

1. It can handle signed integers in 2's complement notion
2. It decreases the number of addition and subtraction
3. It requires less hardware than combinational multiplier
4. It is faster than straightforward sequential multiplier

Hence making it very versatile

Experiment No. 11

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivas
Assisting Teachers	Prof. Avinash Shrivas

Student Name – **Deep Salunkhe**

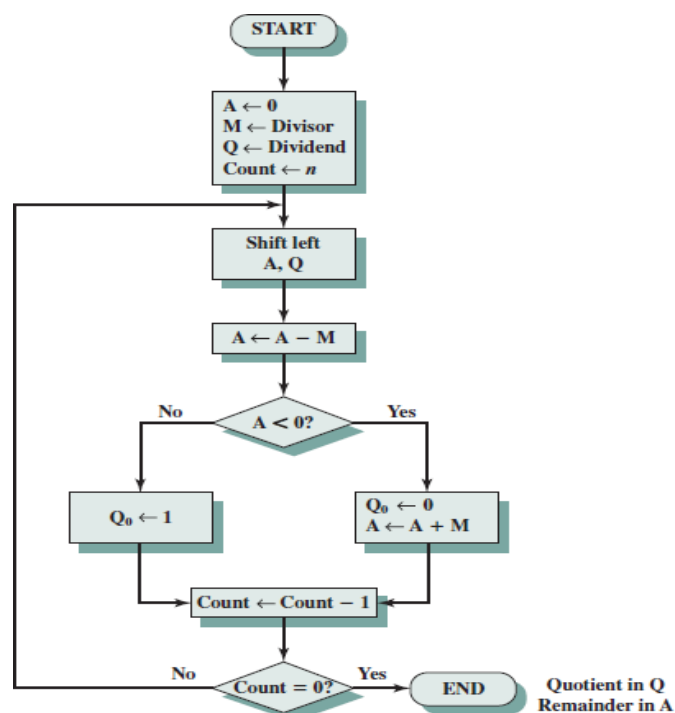
Roll Number – 21102A0014

Division and Batch – Division A, Batch

Date of Implementation – 18/10/2022

Experiment Title: To implement Restoring algorithm for division

Theory: Restoring Division Algorithm is used to divide two unsigned integers. This algorithm is used in Computer Organization and Architecture. This algorithm is called restoring because it restores the value of Accumulator(A) after each or some iterations.



Implementation :-

```
#include<stdlib.h>
#include<stdio.h>
int acum[100]={0}    ;
void add(int acum[],int b[],int n);
int q[100],b[100];
int main()
{
    int x,y;
    printf("Enter the Number :");
    scanf("%d%d",&x,&y);
    int i=0;
    while(x>0 || y>0)
    {
        if(x>0)
        {
            q[i]=x%2;
            x=x/2;
        }
        else
        {
            q[i]=0;
        }
        if(y>0)
        {
            b[i]=y%2;
            y=y/2;
        }
        else
        {
            b[i]=0;
        }
        i++;
    }

    int n=i;
    int bc[50];
    printf("\n");
    for(i=0;i<n;i++)
    {
        if(b[i]==0)
        {
            bc[i]=1;
        }
        else
        {

```

```

bc[i]=0;
}
}
bc[n]=1;
for(i=0;i<=n;i++)
{
if(bc[i]==0)
{
bc[i]=1;
i=n+2;
}
else
{
bc[i]=0;
}
}
int l;
    b[n]=0;
int k=n;
int n1=n+n-1;
int j,mi=n-1;
for(i=n;i!=0;i--)
{
for(j=n;j>0;j--)
{
acum[j]=acum[j-1];

}
acum[0]=q[n-1];
for(j=n-1;j>0;j--)
{
q[j]=q[j-1];
}

add(acum,bc,n+1);
if(acum[n]==1)
{
q[0]=0;
add(acum,b,n+1);
}
else
{
q[0]=1;
}
}
printf("\nQuoient    : ");

for(    l=n-1;l>=0;l--)

```

```
{
printf("%d",q[l]);
}
printf("\nRemainder : ");
for( l=n;l>=0;l--)
{
printf("%d",acum[l]);
}
return 0;
}
void add(int acum[],int bo[],int n)
{
int i=0,temp=0,sum=0;
for(i=0;i<n;i++)
{
sum=0;
sum=acum[i]+bo[i]+temp;
if(sum==0)
{
acum[i]=0;
temp=0;
}
else if (sum==2)
{
acum[i]=0;
temp=1;
}
else if(sum==1)
{
acum[i]=1;
temp=0;
}
else if(sum==3)
{
acum[i]=1;
temp=1;
}
}
}
```

Output:

```
PS E:\GIT> cd "e:\GIT\  
Enter the Number :10  
5  
  
Quoient      : 0010  
Remainder    : 00000
```

Conclusion:-

Thus we successfully implemented Restoring Division Algorithm.

Experiment No. 12

Semester	S.E-Semester III – Computer Engineering
Subject	Digital Logic and Computer Architecture
Subject Professor In-charge	Prof. Avinash Shrivastava
Assisting Teachers	Prof. Avinash Shrivastava

Student Name – **Deep Salunkhe**

Roll Number – 21102A0014

Division and Batch – Division A, Batch

Date of Implementation – 2/11/2022

Experiment Title: -To implement Non-Restoring algorithm for division

Theory: Non-restoring division algorithm is used to divide two unsigned integers. The other form of this algorithm is Restoring Division. This algorithm is different from the other algorithm because here, there is no concept of restoration and this algorithm is less complex than the restoring division algorithm.

Implementation :-

```
#include<stdio.h>
#include<stdlib.h>
int acum[100]={0};
void add(int acum[],int b[],int n);
int q[100],b[100],l;
int main()
{
    int x,y;
    printf("Enter the Number  : ");
    scanf("%d%d",&x,&y);
    int i=0;
    while(x>0||y>0)
    {
        if(x>0)
        {
            q[i]=x%2;
            x=x/2;
        }
        else
        {
            q[i]=0;
        }
        if(y>0)
        {
            b[i]=y%2;
            y=y/2;
        }
        else
        {
            b[i]=0;
        }
        i++;
    }
    int n=i;
    int bc[50];
    printf("\n");
    for(i=0;i<n;i++)
    {
        if(b[i]==0)
        {
            bc[i]=1;
        }
        else
        {
            bc[i]=0;
        }
    }
}
```



```
}  
bc[n]=1;  
for(i=0;i<=n;i++)  
{  
if(bc[i]==0)  
{  
bc[i]=1;  
i=n+2;  
}  
else  
{  
bc[i]=0;  
}  
}  
b[n]=0;  
int j;  
for(i=n;i!=0;i--)  
{  
if(acum[n]==0)  
{  
for(j=n;j>0;j--)  
{  
acum[j]=acum[j-1];  
}  
acum[0]=q[n-1];  
for(j=n-1;j>0;j--)  
{  
q[j]=q[j-1];  
}  
add(acum,bc,n+1);  
}  
else  
{  
for(j=n;j>0;j--)  
{  
acum[j]=acum[j-1];  
}  
acum[0]=q[n-1];  
for(j=n-1;j>0;j--)  
{  
q[j]=q[j-1];  
}  
add(acum,b,n+1);  
}  
if(acum[n]==1)  
{  
q[0]=0;  
}  
}
```

```

else
{
q[0]=1;
}
}
if(acum[n]==1)
{
add(acum,b,n+1);
}
printf("\nQuoient   : ");
for( l=n-1;l>=0;l--)
{
printf("%d",q[l]);
}
printf("\nRemainder : ");
for( l=n;l>=0;l--)
{
printf("%d",acum[l]);
}
return 0;
}
void add(int acum[],int bo[],int n)
{
int i=0,temp=0,sum=0;
for(i=0;i<n;i++)
{
sum=0;
sum=acum[i]+bo[i]+temp;
if(sum==0)
{
acum[i]=0;
temp=0;
}
else if(sum==2)
{
acum[i]=0;
temp=1;
}
else if(sum==1)
{
acum[i]=1;
temp=0;
}
else if(sum==3)
{
acum[i]=1;
temp=1;
}
}

```

```
}  
}
```

Output:-

```
PS E:\GIT> cd "e:\GIT\" ; if ($?)  
Enter the Number : 50 25  
  
Quoient : 000010  
Remainder : 0000000
```

Conclusion:-

Thus, We successfully implemented Non-Restoring Division Algorithm.