INDEX

Sr. No.	Name of Experiment	D.O.P.	D.O.C.	Page No.	Remark
1	To verify the truth table of various logic gates using ICs.				
2	To realize the gates using universal gates.				
3	To realize half adder and full adder.				
4	Study of flip flop IC				
5	To implement ripple carry adder.				
6	To implement carry look ahead adder.				
7	To implement Booth's algorithm.				
8	To implement restoring division algorithm.				
9	To implement non restoring division algorithm.				
10	To implement ALU design.				

Experiment No. 1
Truth table of various logic gates using ICs.
Name: KIranVishnu Dhuri
Roll Number:07
Date of Performance:
Date of Submission:

Aim - To verify the truth table of various logic gates using ICs.

Objective -

1. Understand how to use the breadboard to patch up, test your logic design and debug it.
2. The principal objective of this experiment is to fully understand the function and use of logic gates.

3. Understand how to implement simple circuits based on a schematic diagram using logic gates.

Components required 1.

IC's 7408, 7432, 7404

2. Bread Board.

3. Connecting wires.

Theory -

In digital electronics, a gate is logic circuits with one output and one or more inputs. Logic gates are available as integrated circuits.

AND gate :

AND gate performs logical multiplication, more commonly known as AND operation. The AND gate output will be in high state only when all the inputs are in high state.7408 is a Quad 2 input AND gate.

OR gate:

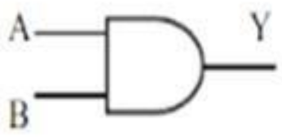
It performs logical addition. Its output become high if any of the inputs is in logic high. 7432 is a Quad 2 input OR gate.

NOT gate:

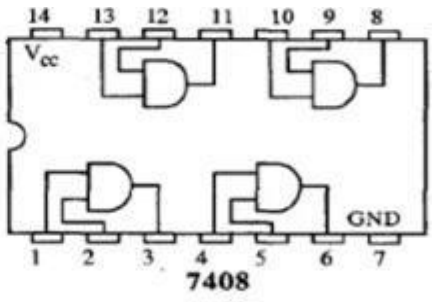
It performs basic logic function for inversion or complementation. The purpose of the inverter is to change one logic level to the opposite level. IC 7404 is a Hex inverter.

Circuit Diagram, Truth Table -

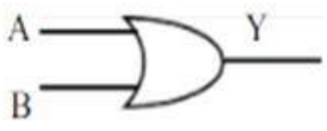
AND Gate -



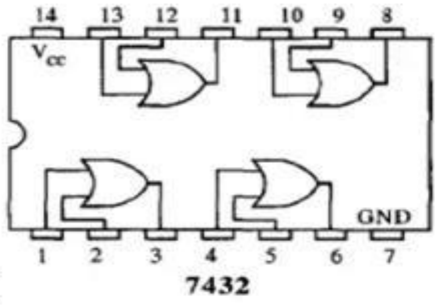
A	B	Y(A.B)
0	0	0
0	1	0
1	0	0
1	1	1



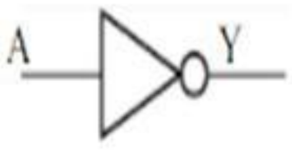
OR Gate -



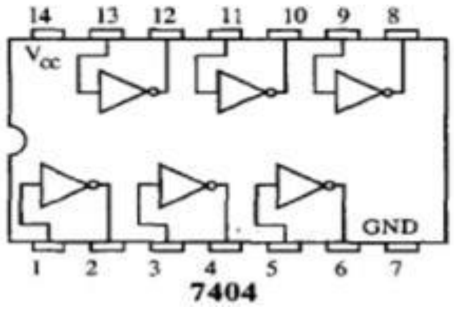
A	B	Y(A+B)
0	0	0
0	1	1
1	0	1
1	1	1



NOT Gate -



A	Y=A'
0	1
1	0



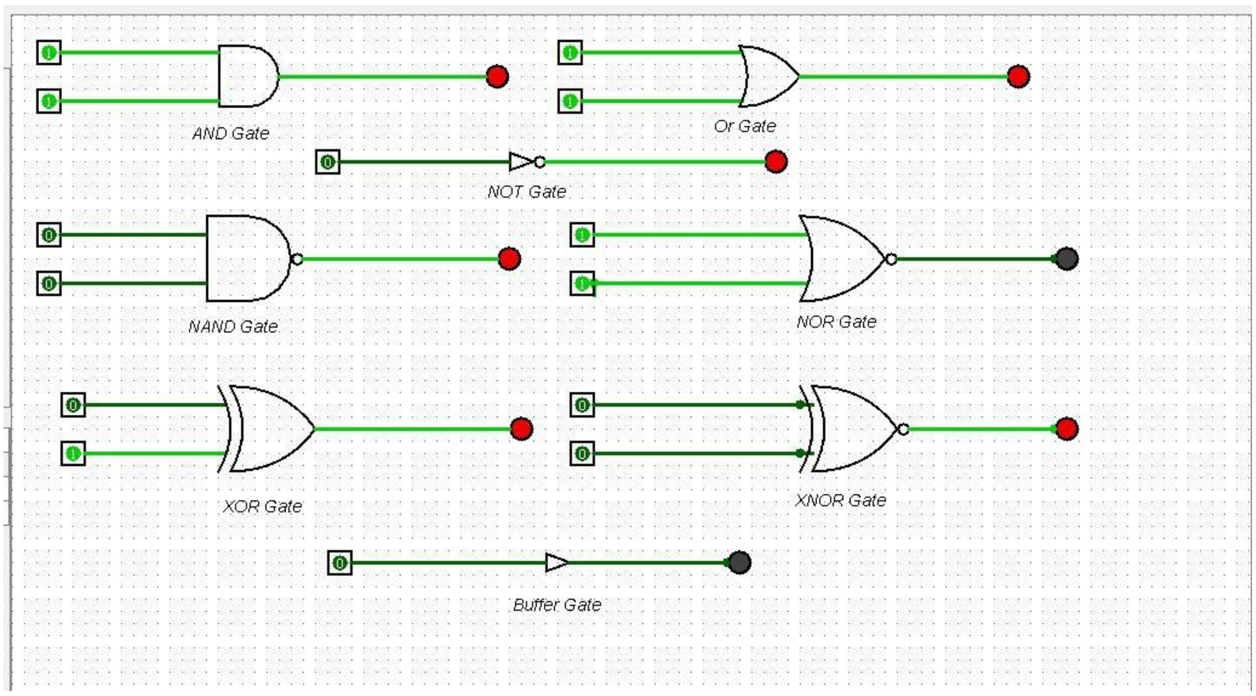
Procedure:

1.Test all the components in the Ic packages using a digital IC tester. Also assure whether all the connecting wires are in good condition by testing for the continuity using a Multimeter or a trainer kit.

2.Verify the dual in line package (DIP) in out of the IC before feeding the inputs.

3.Set up the circuits and observe the outputs.

Output:-



Conclusion -

This experiment with various logic gates in Logisim highlighted the distinct roles of NOT, AND, OR, and XOR gates in digital circuits. It demonstrated the importance of precision in circuit design and showcased how these gates can be combined to perform complex logical operations. This hands-on experience deepened our understanding of digital logic, laying the groundwork for more advanced circuit design and analysis in fields like electrical engineering and computer science.

Experiment No. 2
Basic gates using universal gates.
Name: Kiran Vishnu Dhuri
Roll Number: 07

Date of Performance:
Date of Submission:

Aim - To realize the gates using universal gates.

Objective -

1) To study the realization of basic gates using universal gates.
2) Understanding how to construct any combinational logic function using NAND or NOR gates only.

Theory -

AND, OR, NOT are called basic gates as their logical operation cannot be simplified further.

NAND and NOR are called universal gates as using only NAND or only NOR, any logic function can be implemented.

Components required 1. IC's

7400(NAND) 7402(NOR)

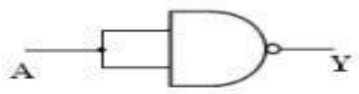
2. Bread Board.

3. Connecting wires.

Circuit Diagram -

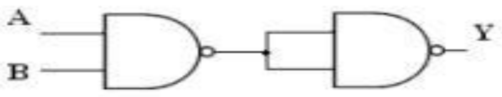
Implementation using NAND gate:

(a) NOT gate: $Y = A'$



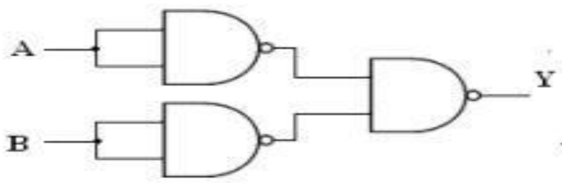
A	Y
0	1
1	0

(b) AND gate: $Y = A \cdot B$



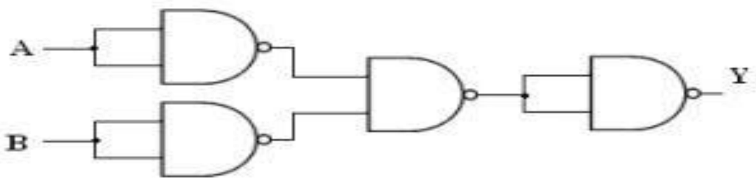
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

(c) OR gate: $Y = A + B$



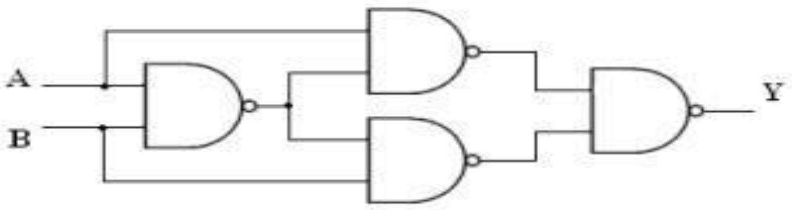
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(d) NOR gate: $Y = (A + B)'$



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

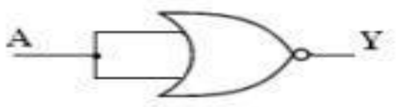
(e) Ex-OR gate: $Y = A \oplus B$



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

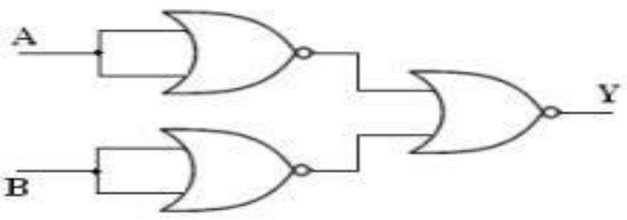
Implementation using NOR gate:

(a) NOT gate: $Y = A'$



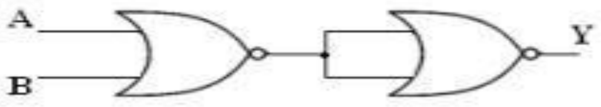
A	Y
0	1
1	0

(b) AND gate: $Y = A \cdot B$



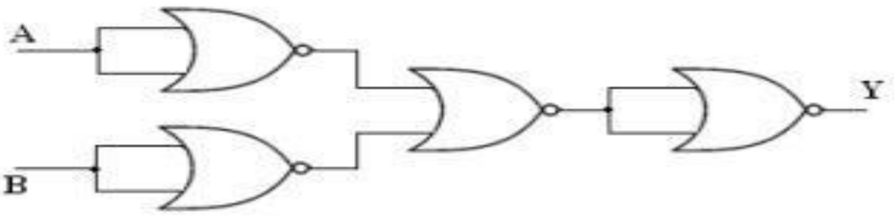
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

(c) OR gate: $Y = A + B$



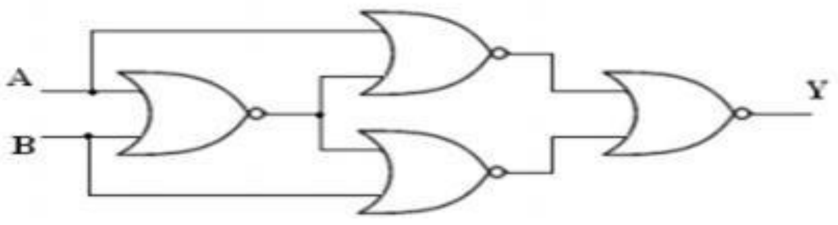
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(d) NAND gate: $Y = (AB)'$



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

(e) Ex-NOR gate: $Y = A \odot B = (A \oplus B)'$



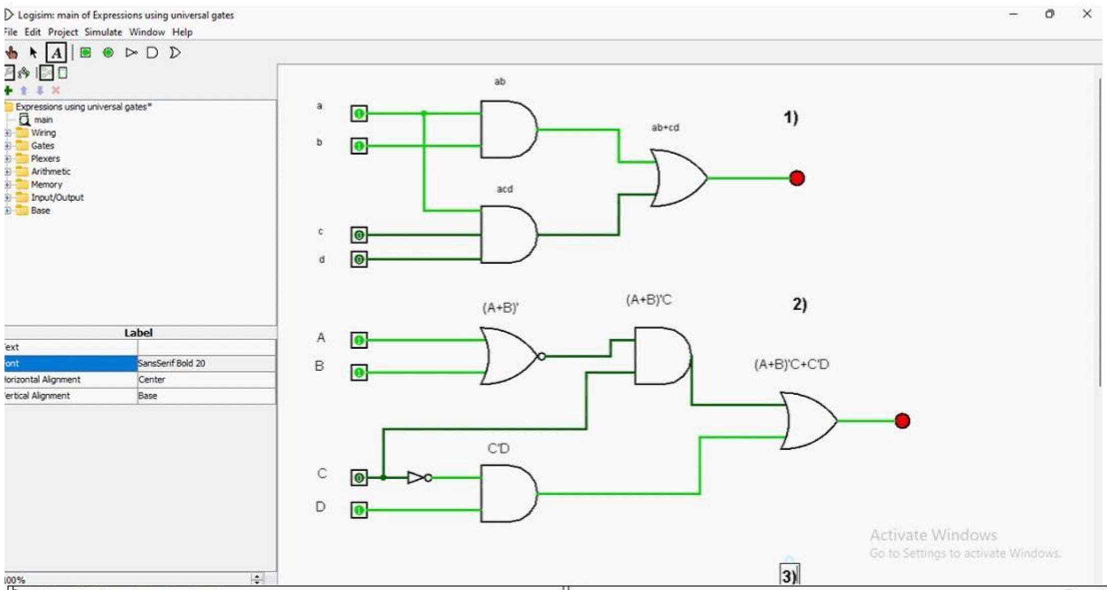
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

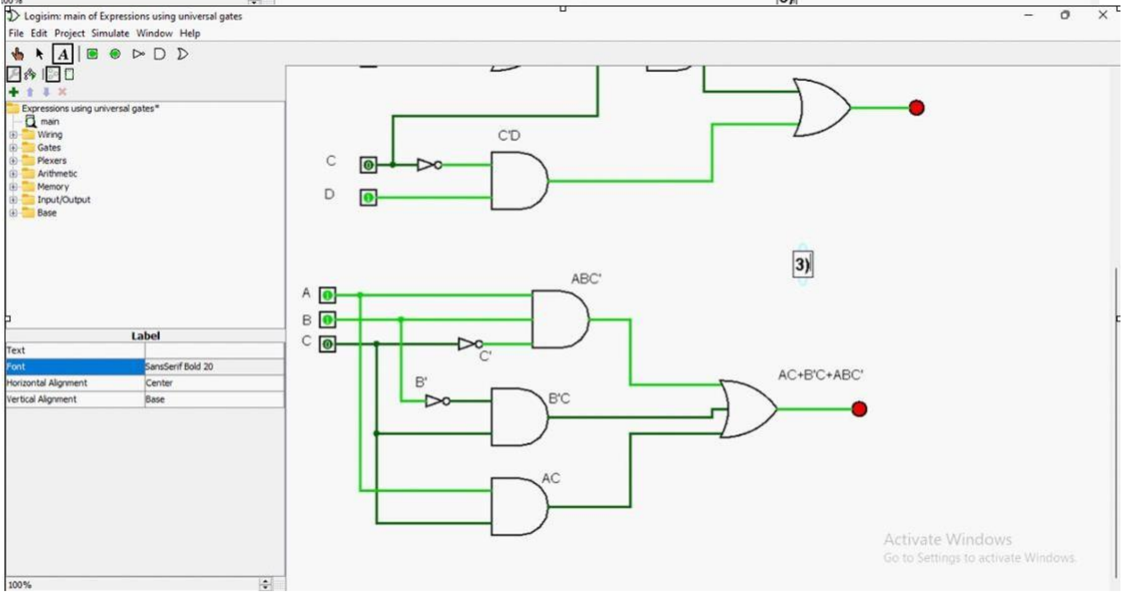
Procedure:

a) Connections are made as per the circuit diagrams.

b) By applying the inputs, the outputs are observed and the operations are verified with the help of truth table.

Output:-





Conclusion –

When exploring the realm of digital circuit design, the concept of implementing basic logic gates through universal gates presents an intriguing and versatile approach. By leveraging the fundamental properties of universal gates such as NAND and NOR gates, it becomes evident that complex logical operations can be accomplished through the arrangement of these basic components. This not only showcases the remarkable adaptability and efficiency of these universal gates but also underscores the significance of their role in the broader landscape of digital logic design.

Experiment No. 3
To realize half adder and full adder.
Name: Kiran Vishnu Dhuri
Roll Number: 07
Date of Performance:
Date of Submission:

Aim - To realize half adder and full adder.

Objective -

1) The objective of this experiment is to understand the function of Half-adder, Fulladder, Half-subtractor and Full-subtractor.
2) Understand how to implement Adder and Subtractor using logic gates.

Components required -

1. IC's - 7486(X-OR), 7432(OR), 7408(AND), 7404 (NOT)

2. Bread Board

3. Connecting wires.

Theory -

Half adder is a combinational logic circuit with two inputs and two outputs. The half adder circuit is designed to add two single bit binary numbers A and B. It is the basic building block for addition of two single bit numbers. This circuit has two outputs CARRY and SUM.

$$\text{Sum} = A \oplus B$$

$$\text{Carry} = A\,B$$

Full adder is a combinational logic circuit with three inputs and two outputs. Full adder is developed to overcome the drawback of HALF ADDER circuit. It can add two one bit umbers A and B. The full adder has three inputs A, B, and CARRY in,the circuit has two outputs CARRY out and SUM.

$$\text{Sum} = (A \oplus B) \oplus \text{Cin}$$

$$\text{Carry} = AB + \text{Cin}\,(A \oplus B)$$

Subtracting a single-bit binary value B from another A (i.e. A -B) produces a difference bit D and a borrow out bit B-out. This operation is called half subtraction and the circuit to realize it is called a half subtractor. The Boolean functions describing the halfSubtractor are

$$\text{Sum} = A \oplus B$$

$$\text{Carry} = A'\,B$$

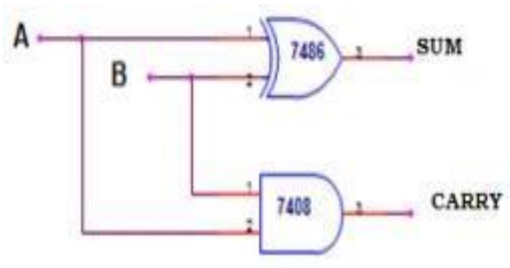
Subtracting two single-bit binary values, B, Cin from a single-bit value A produces a difference bit D and a borrow out Br bit. This is called full subtraction. The Boolean functions describing the full-subtractor are

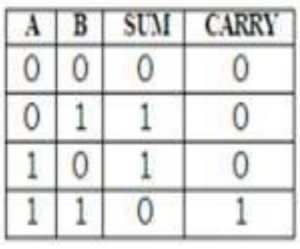
$$\text{Difference} = (A \oplus B) \oplus \text{Cin}$$

$$\text{Borrow} = A'B + A'(\text{Cin}) + B(\text{Cin})$$

Circuit Diagram and Truth Table -

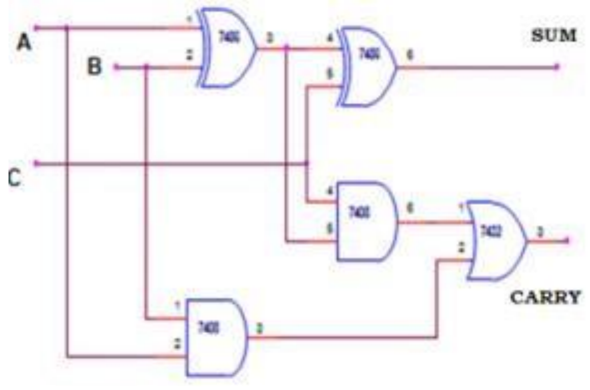
Half-adder

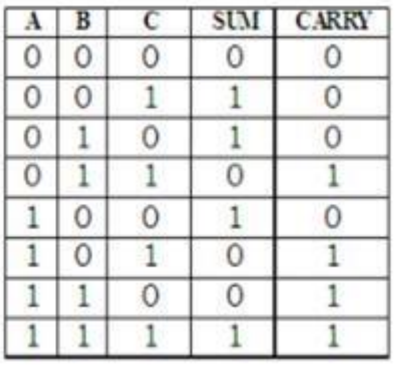




A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full-adder



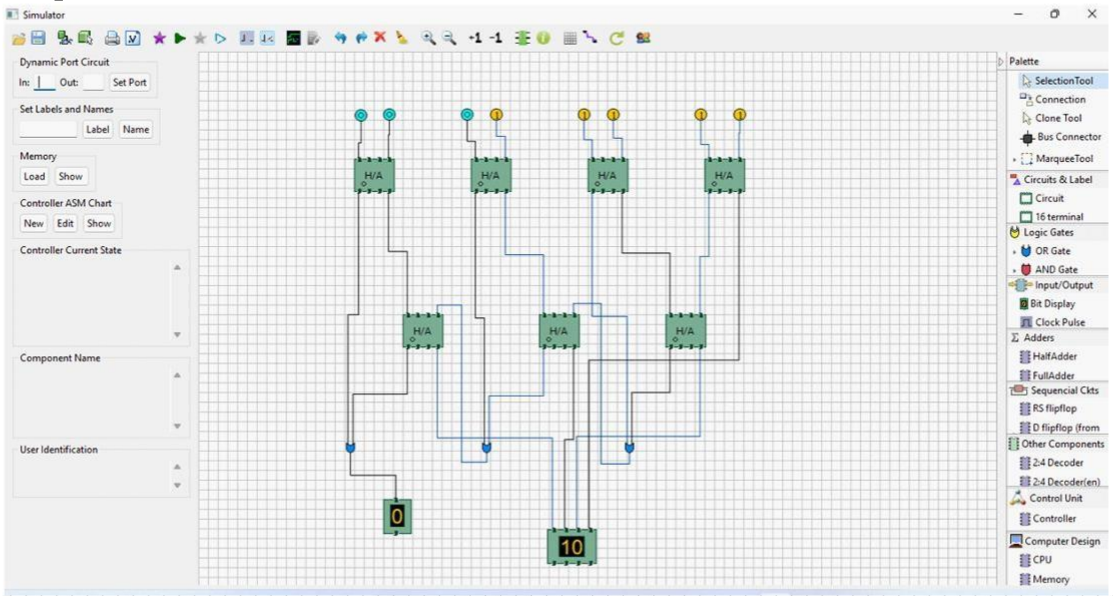


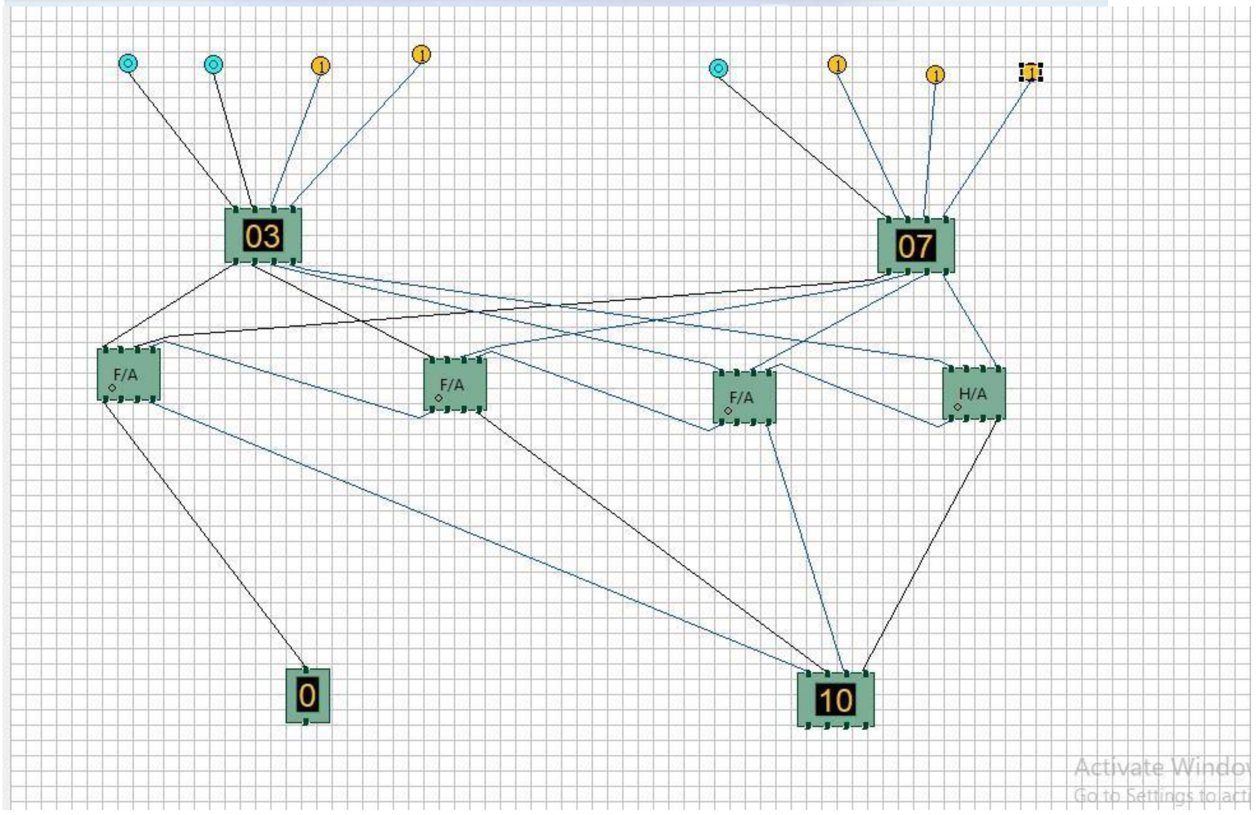
A	B	C	SUM	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Procedure -

1. Verify the gates.
2. Make the connections as per the circuit diagram.
3. Switch on VCC and apply various combinations of input according to truth table.
4. Note down the output readings for half/full adder and half/full subtractor, Sum/difference and the carry/borrow bit for different combinations of inputs verify their truth tables.

Output:-





/ **Conclusion –**

The experiment conducted on the half adder and full adder using Logisim has provided valuable insights into the fundamental principles of digital logic design. We successfully demonstrated the basic operations of addition using half adders and extended our understanding to full adders, which can handle carry inputs. This hands-on experience not only reinforced our knowledge of binary arithmetic but also underscored the importance of these components in building more complex digital circuits.

Experiment No. 4
FLIP-FLOP IC
Name: Kiran Vishnu Dhuri
Roll Number: 07
Date of Performance:
Date of Submission:

FLIP-FLOP IC

AIM:- **To study about Flip-Flop IC's.**

OBJECTIVES:-

Objective 1: Characterization of Flip-Flop ICs

To analyze and compare the fundamental operating principles of various flip-flop ICs, such as D-type, JK-type, and T-type flip-flops.

To measure and record key parameters, including propagation delay, setup time, hold time, and clock-to-output delay, for different flip-flop configurations.

To determine the power consumption of flip-flop ICs under different clock frequencies and input conditions.

Objective 2: Exploration of Flip-Flop Logic Behavior

To examine the behavior of flip-flop ICs under different clocking scenarios, including edge-triggered and level-triggered modes.

To investigate how flip-flop logic state changes based on input signal variations and clocking transitions.

To study the impact of metastability on flip-flop operation and explore methods to mitigate its effects.

Objective 3: Application Analysis of Flip-Flop ICs

To design and implement a binary counter circuit using flip-flop ICs to demonstrate their practical use in digital counting applications.

To construct a frequency divider circuit using flip-flop ICs and evaluate its effectiveness in dividing input clock frequencies.

To explore the role of flip-flop ICs in synchronous sequential circuits, such as shift registers and memory elements.

Objective 4: Flip-Flop IC Performance Under Non-Ideal Conditions

To simulate and analyze the behavior of flip-flop ICs under noisy or distorted clock signals.

To investigate the susceptibility of flip-flop logic to voltage fluctuations and evaluate its impact on circuit reliability.

To explore the limitations of flip-flop ICs in high-speed and lowpower applications and propose potential improvements.

Objective 5: Design and Optimization of Flip-Flop Circuits

To design custom flip-flop circuits with specific functionalities, such as frequency division or data storage, using VHDL or other hardware description languages.

To optimize the design parameters of flip-flop circuits for minimal power consumption, reduced propagation delays, and improved noise immunity.

To evaluate the performance of the designed circuits through simulation and practical implementation on breadboards or FPGA platforms.

THEORY:-

Digital electronic circuit is classified into combinational logic and sequential logic.

Combinational logic output depends on the inputs levels, whereas sequential logic output

Depends on stored levels and also the input levels.

The storage elements (Flip -flops) are devices capable of storing 1-bit binary info. The binary

info stored in the memory elements at any given time defines the state of the Sequential

circuit. The input and the present state of the memory element determines the output. Storage

elements next state is also a function of external inputs and present state.

FLIP FLOP AND THEIR PROPERTIES:-

Flip-flops are synchronous bistable devices. The term synchronous means the output

changes state only when the clock input is triggered. That is, changes in the output occur in

synchronization with the clock. A flip-flop circuit has two outputs, one for the normal value

and one for the complement value of the stored bit. Since memory elements in sequential

circuits are usually flip-flops, it is worth summarizing the behavior of various flip-flop types

before proceeding further. All flip -flops can be divided into four basic types: SR, JK, D and

T. They differ in the number of inputs and in the response invoked by different value of input

signals. The four types of flip -flops are defined in the Table

At its core, a flip-flop IC is a bistable multivibrator, a device that can maintain one of two stable states (0 or 1) until a specific triggering event occurs. This event is often tied to a clock signal or input transitions. The fundamental concept of bi stability is based on positive feedback loops within the circuit.

Common Flip-Flop Types:

Several flip-flop IC types exist, each with unique characteristics suited for various applications:

1)RS (Reset-Set) Flip-Flop:

Basic building block of other flip-flop types.

Utilizes two cross-coupled NOR or NAND gates.

Lacks clock input, sensitive to input changes.

2) D-Type Flip-Flop:

a)Stores a single data bit.

b)Edge-triggered by clock signal.

c)Offers synchronous data transfer.

3) JK-Type Flip-Flop:

a)Combines RS flip-flop functionality with additional logic.

b)Serves as a universal flip-flop with toggling capability.

4) T-Type (Toggle) Flip-Flop:

a)Toggles its output state with each clock pulse.

b)Useful for frequency division and clock signal generation.

5) Master-Slave Flip-Flop:

a)Combines two flip-flops to mitigate input transition issues.

b)Master flip-flop captures input, while slave flip-flop responds to clock edges.

Conclusion:

Flip-flop ICs represent a cornerstone in digital electronics, providing the means to store data, synchronize operations, and enable complex logic functions. Their bistable nature, edge-triggered behavior, and diverse applications make them indispensable components for creating digital systems of varying complexity. Understanding the theory behind flip-flop ICs empowers engineers and designers to develop efficient and reliable digital circuits and systems.

Experiment No. 5
Implement ripple carry adder
Name: kiran vishnu dhuri
Roll Number: 07
Date of Performance:
Date of Submission:

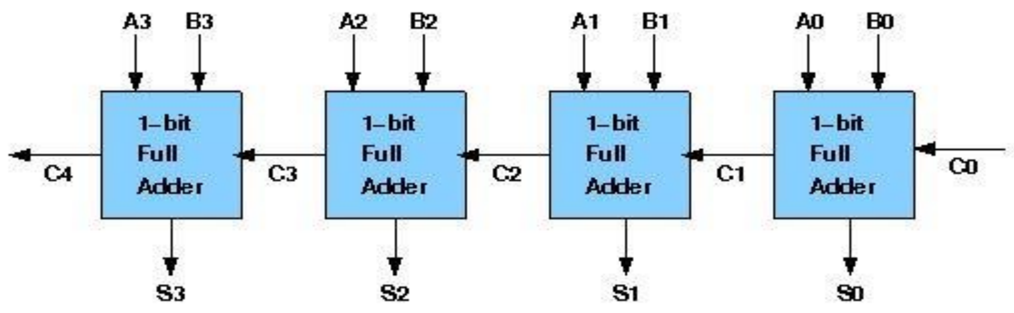
Aim: To implement ripple carry adder.

Objective: To understand the operation of a ripple carry adder, specifically how the carry ripples through the adder.

1. examining the behavior of the working module to understand how the carry ripples through the adder stages
2. to design a ripple carry adder using full adders to mimic the behavior of the working module
3. the adder will add two 4 bit numbers

Theory: Arithmetic operations like addition, subtraction, multiplication, division are basic operations to be implemented in digital computers using basic gates like AND, OR, NOR, NAND etc. Among all the arithmetic operations if we can implement addition then it is easy to perform multiplication (by repeated addition), subtraction (by negating one operand) or division (repeated subtraction).

Half Adders can be used to add two one bit binary numbers. It is also possible to create a logical circuit using multiple full adders to add N-bit binary numbers. Each full adder inputs a Cin, which is the Cout of the previous adder. This kind of adder is a Ripple Carry Adder, since each carry bit "ripples" to the next full adder. The first (and only the first) full adder may be replaced by a half adder. The block diagram of 4-bit Ripple Carry Adder is shown here below -



The layout of ripple carry adder is simple, which allows for fast design time; however, the ripple carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit [ripple carry] adder, there are 32 full adders, so the critical path (worst case) delay is 31 * 2(for carry propagation) + 3(for sum) = 65 gate delays.

Design Issues:

The corresponding Boolean expressions are given here to construct a ripple carry adder. In the half adder circuit the sum and carry bits are defined as

sum = $A \oplus B$

carry = AB

In the full adder circuit the the Sum and Carry outpur is defined by inputs A, B and Carryin as

Sum=ABC + ABC + ABC + ABC

Carry=ABC + ABC + ABC + ABC

Having these we could design the circuit. But, we first check to see if there are any logically equivalent statements that would lead to a more structured equivalent circuit.

With a little algebraic manipulation, one can see that

Sum= ABC + ABC + ABC + ABC

$= (AB + AB)\, C + (AB + AB)\, C$

$= (A \oplus B)\, C + (A \oplus B)\, C$

$= A \oplus B \oplus C$

Carry= ABC + ABC + ABC + ABC

$= AB + (AB + AB)\, C$

$= AB + (A \oplus B)\, C$

Procedure:

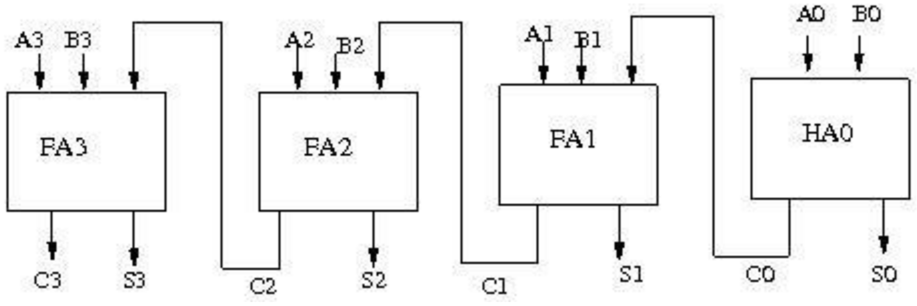
Procedure to perform the experiment: Design of Ripple Carry Adders

1) Start the simulator as directed. This simulator supports 5-valued logic.
2) To design the circuit we need 3 full adder, 1 half adder, 8 Bit switch(to give input), 3 Digital display(2 for seeing input and 1 for seeing output sum), 1 Bit display(to see the carry output), wires.
3) The pin configuration of a component is shown whenever the mouse is hovered on any canned component of the palette or presses the 'show pin config'button. Pin numbering starts from 1 and from the bottom left corner (indicating with the circle) and increases anticlockwise.
4) For half adder input is in pin-5,8 output sum is in pin-4 and carry is pin-1, For full adder input is in pin-5,6,8 output sum is in pin-4 and carry is pin-1
5) Click on the half adder component(in the Adder drawer in the pallet) and then click on the position of the editor window where you want to add the component(no drag and drop, simple click will serve the purpose), likewise add 3 full adders(from the Adder drawer in the pallet), 8 Bit switches, 3 digital display and 1 bit Displays(from Display and Input drawer of the pallet, if it is not seen scroll down in the drawer)
6) To connect any two components select the Connection menu of Palette, and then click on the Source terminal and click on the target terminal. According to the circuit diagram connect all the components, connect 4 bit switches to the 4 terminals of a digital display and another set of 4 bit switches to the 4 terminals of another digital display. connect

the pin-1 of the full adder which will give the final carry output. connet the sum(pin-4) of all the adders to the terminals of the third digital display(according to the circuit diagram shown in screenshot). After the connection is over click the selection tool in the pallet.

7) To see the circuit working, click on the Selection tool in the pallet then give input by double clicking on the bit switch, (let it be 0011(3) and 0111(7)) you will see the output on the output(10) digital display as sum and 0 as carry in bit display.

Circuit diagram of Ripple Carry Adder:



Components required:

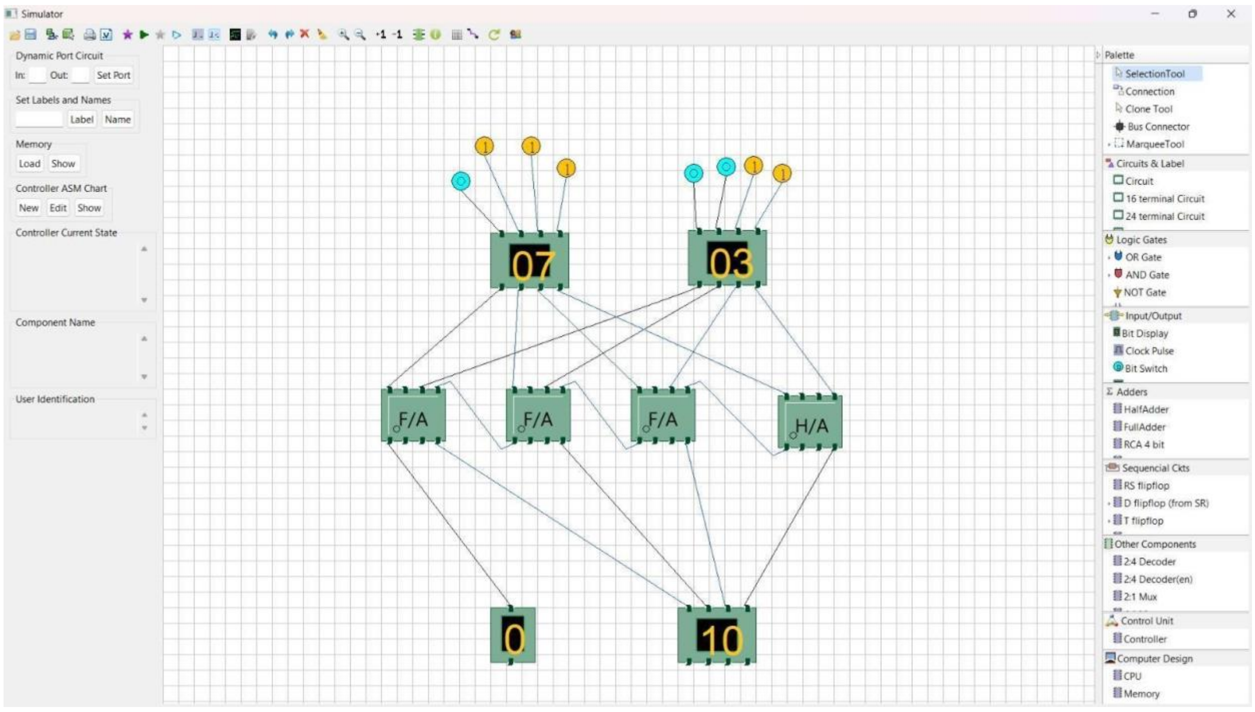
The components needed to create 4 bit ripple carry adder is listed here -

- 4 full-adders
- wires to connect
- LED display to obtain the output OR

we can use

- 3 full-adders
- 1 half adder
- wires to connect
- LED display to obtain the output **Screenshots**

of Ripple Carry Adder:



Conclusion: Ripple carry adder is a simple and low-cost design of multi-bit adder that uses cascaded full adders to perform binary addition. However, it suffers from a high propagation delay, as each full adder has to wait for the carry input from the previous stage. To overcome this limitation, various techniques have been proposed to reduce the carry logic to two-level or even one-level logic, such as carry look-ahead adder, carry skip adder, and prefix adder. These techniques improve the speed of addition, but at the cost of increased area and complexity.

Experiment No.6
Implement Carry Look Ahead Adder.
Name: kiran vishnu dhuri
Roll Number: 07
Date of Performance:
Date of Submission:

Aim: . To implement carry look ahead adder.

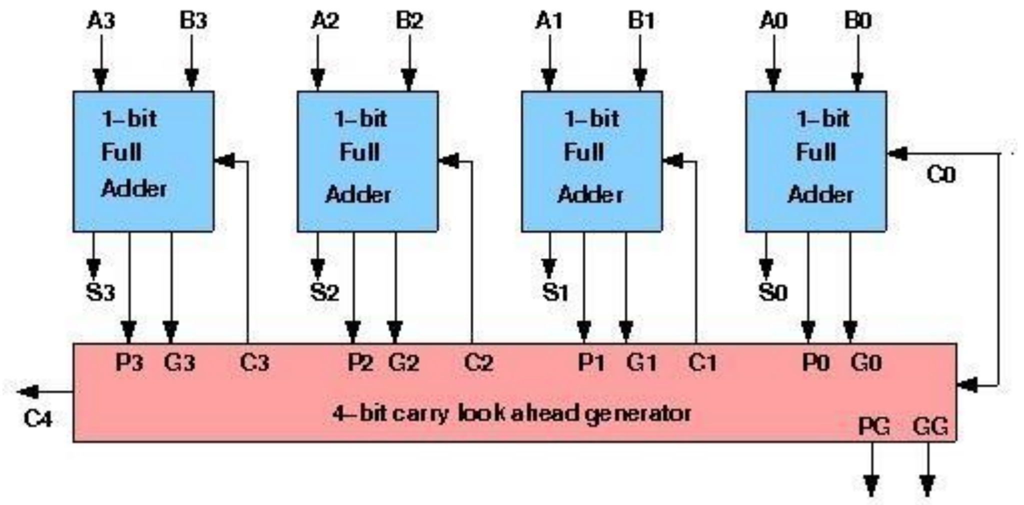
Objective:

It computes the carries parallely thus greatly speeding up the computation.

1. To understanding behaviour of carry lookahead adder from module designed by the student as part of the experiment
2. To understand the concept of reducing computation time with respect of ripple carry adder by using carry generate and propagate functions. 3. The adder will add two 4 bit numbers

Theory:

To reduce the computation time, there are faster ways to add two binary numbers by using carry lookahead adders. They work by creating two signals P and G known to be Carry Propagator and Carry Generator. The carry propagator is propagated to the next level whereas the carry generator is used to generate the output carry ,regardless of input carry. The block diagram of a 4-bit Carry Lookahead Adder is shown here below -



The number of gate levels for the carry propagation can be found from the circuit of full adder. The signal from input carry Cin to output carry Cout requires an AND gate and an OR gate, which constitutes two gate levels. So if there are four full adders in the parallel adder, the output carry C5 would have 2 X 4 = 8 gate levels from C1 to C5. For an n-bit parallel adder, there are 2n gate levels to propagate through.

Design Issues :

The corresponding boolean expressions are given here to construct a carry lookahead adder. In the carry-lookahead circuit we ned to generate the two signals carry propagator(P) and carry generator(G),

$$Pi = Ai \oplus Bi$$

$$Gi = Ai \cdot Bi$$

The output sum and carry can be expressed as

$$Sumi = Pi \oplus Ci$$

$$Ci+1 = Gi + (Pi \cdot Ci)$$

Having these we could design the circuit. We can now write the Boolean function for the carry output of each stage and substitute for each Ci its value from the previous equations:

$C1 = G0 + P0 \cdot C0$

$C2 = G1 + P1 \cdot C1 = G1 + P1 \cdot G0 + P1 \cdot P0 \cdot C0$

$C3 = G2 + P2 \cdot C2 = G2\ P2 \cdot G1 + P2 \cdot P1 \cdot G0 + P2 \cdot P1 \cdot P0 \cdot C0$

$C4 = G3 + P3 \cdot C3 = G3\ P3 \cdot G2\ P3 \cdot P2 \cdot G1 + P3 \cdot P2 \cdot P1 \cdot G0 + P3 \cdot P2 \cdot P1 \cdot P0 \cdot C0$

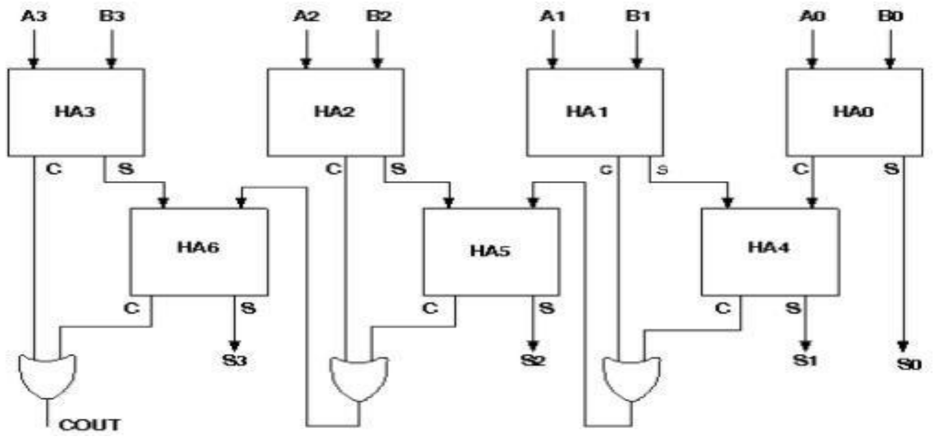
Procedure:

Procedure to perform the experiment: Design of Carry Look ahead Adders

1) Start the simulator as directed. This simulator supports 5-valued logic.
2) To design the circuit we need 7 half adder, 3 OR gate, 1 V+(to give 1 as input), 3 Digital display(2 for seeing input and 1 for seeing output sum), 1 Bit display(to see the carry output), wires.
3) The pin configurations of a component are shown whenever the mouse is hovered on any canned component of the palette or press the 'show pinconfig' button. Pin numbering starts from 1 and from the bottom left corner (indicating with the circle) and increases anticlockwise.
4) For half adder input is in pin-5,8 output sum is in pin-4 and carry is pin-1
5) Click on the half adder component(in the Adder drawer in the pallet) and then click on the position of the editor window where you want to add the component(no drag and drop, simple click will serve the purpose), likewise add 6 more full adders(from the Adder drawer in the pallet), 3 OR gates(from Logic Gates drawer in the pallet), 1 V+, 3 digital display and 1 bit Displays(from Display and Input drawer of the pallet, if it is not seen scroll down in the drawer)
6) To connect any two components select the Connection menu of Palette, and then click on the Source terminal and click on the target terminal. According to the circuit diagram connect all the components; connect V+ to the upper input terminals of 2 digital displays according to you input. Connect the OR gates according to the diagram shown in the screenshot connect the pin-1 of the half adder which will give

   the final carry output. Connect the sum (pin-4) of those adders to the terminals of the third digital display which will give output sum. After the connection is over click the selection tool in the pallet.
7) See the output; in the screenshot diagram we have given the value 0011(3) and 0111(7) so get 10 as sum and 0 as carry. You can also use many bit switches instead of V+ to give input and by double clicking those bit switches can give different values and check the result.

Circuit diagram of Carry Look Ahead Adder:

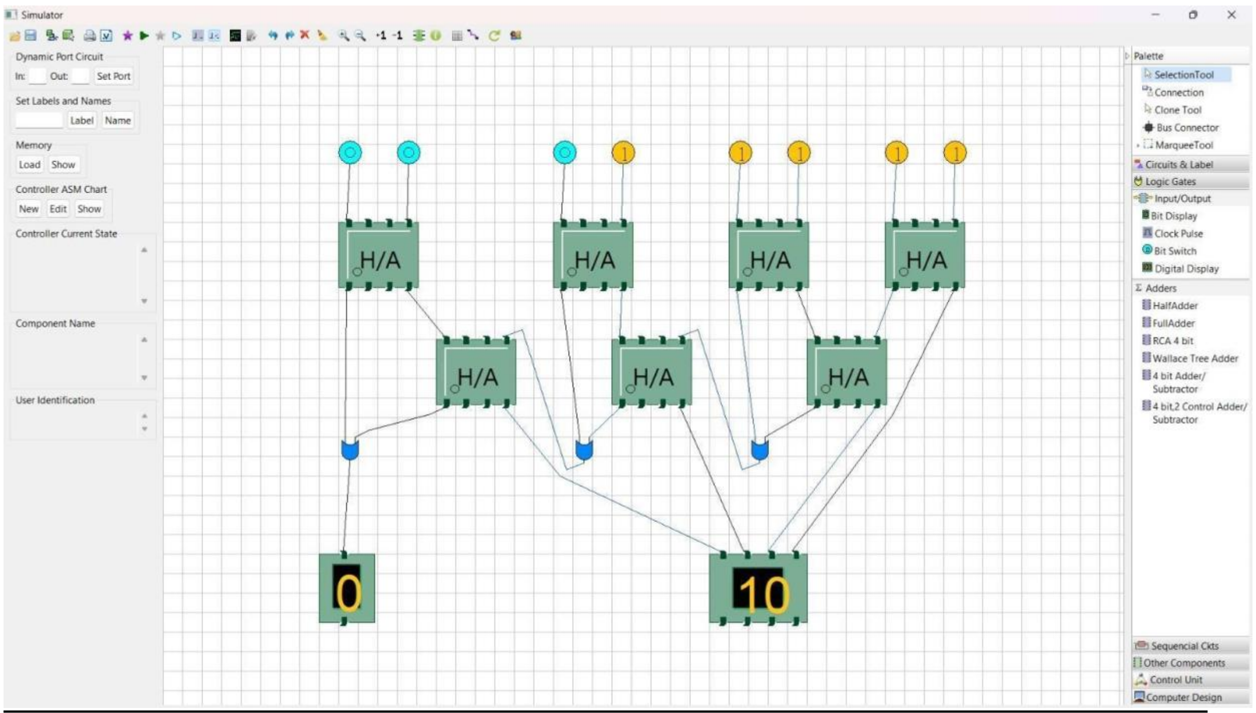


Components required:

The components needed to create 4 bit carry look ahead adder is listed here -

1. 7 half-adders: 4 to create the look adder circuit, and 3 to evaluate Si and Pi · Ci
2. 3 OR gates to generate the next level carry Ci+1
3. wires to connect
4. LED display to obtain the output

Screenshots of Carry Look Ahead Adder:



Conclusion: Carry look-ahead adder is a fast and efficient design of multi-bit adder that uses the concepts of carry generate and carry propagate to reduce the carry propagation delay. It computes the carry signals in advance, based on the input bits, without waiting for the previous stage. It uses a complex hardware circuit that consists of partial full adders and carry look-ahead logic. It can be cascaded to form larger adders by using additional circuitry.

Experiment No. 7
Implement Booth's algorithm using c-programming
Name: Kiran Vishnu Dhuri

Roll Number: 07
Date of Performance:
Date of Submission:

Aim: To implement Booth's algorithm using c-programming.

Objective -

1. To understand the working of Booths algorithm.
2. To understand how to implement Booth's algorithm using c-programming.

Theory:

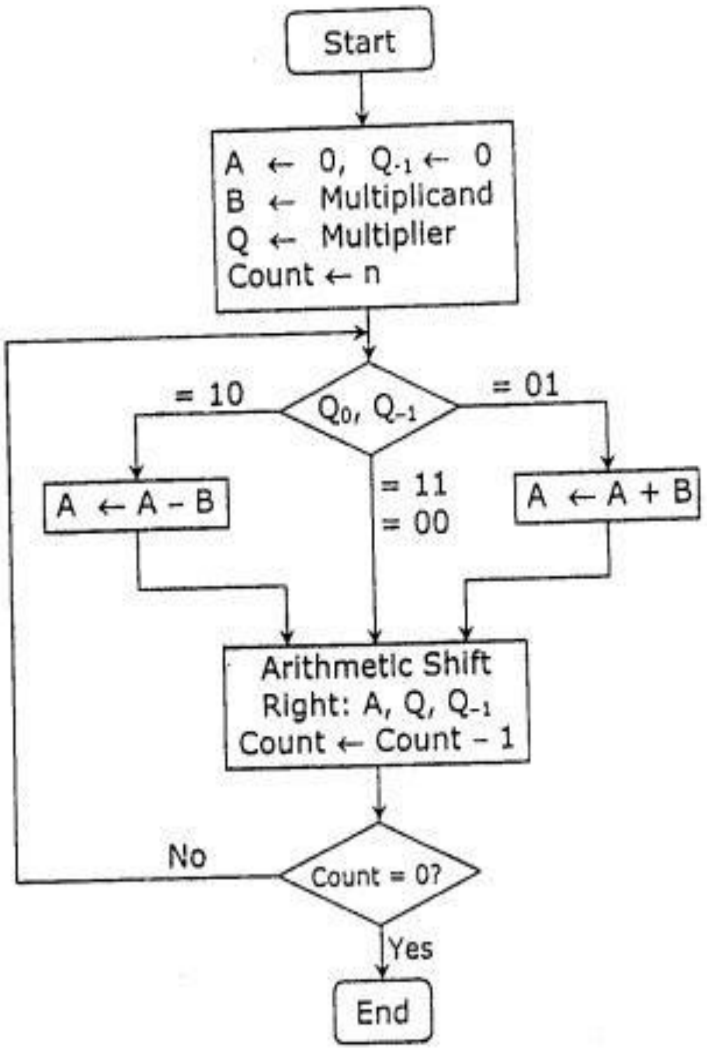
Booth's algorithm is a multiplication algorithm that multiplies two signed binary numbers in 2's complement notation. Booth used desk calculators that were faster at shifting than adding and created the algorithm to increase their speed.

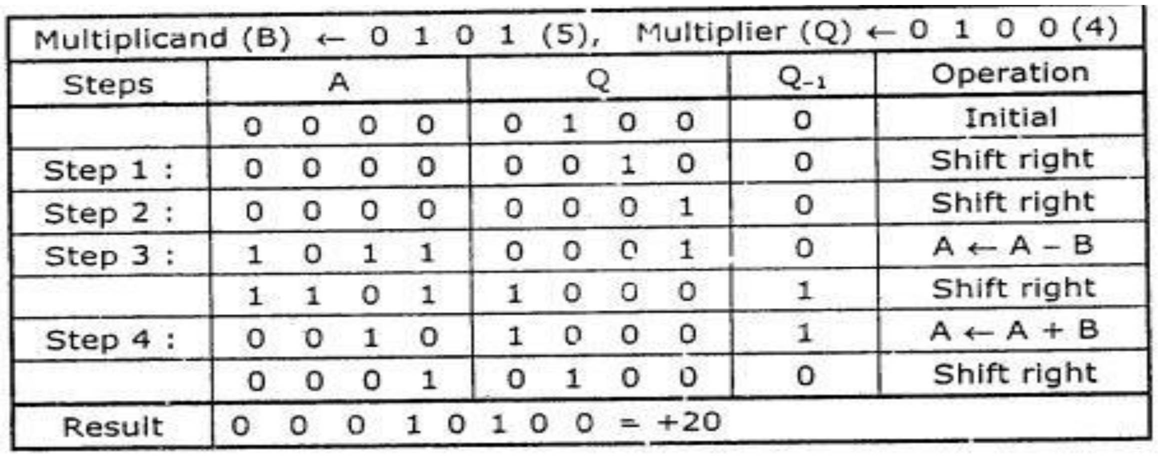
The algorithm works as per the following conditions :

1. If Qn and Q_{-1} are same i.e. 00 or 11 perform arithmetic shift by 1 bit.

2. If Qn Q_{-1} = 10 do A= A - B and perform arithmetic shift by 1 bit.

3. If Qn Q_{-1} = 01 do A= A + B and perform arithmetic shift by 1 bit.





Multiplicand (B) ← 0 1 0 1 (5), Multiplier (Q) ← 0 1 0 0 (4)

Steps	A	Q	Q_{-1}	Operation
	0 0 0 0	0 1 0 0	0	Initial
Step 1 :	0 0 0 0	0 0 1 0	0	Shift right
Step 2 :	0 0 0 0	0 0 0 1	0	Shift right
Step 3 :	1 0 1 1	0 0 0 1	0	A ← A – B
	1 1 0 1	1 0 0 0	1	Shift right
Step 4 :	0 0 1 0	1 0 0 0	1	A ← A + B
	0 0 0 1	0 1 0 0	0	Shift right
Result	0 0 0 1 0 1 0 0 = +20			

Program:

```
#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;

        }

    }


  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];

 }
```

```
  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(){
   int temp = pro[4], temp2 = pro[0], i; for
   (i = 1; i < 5 ; i++){
     pro[i-1] = pro[i]; }
   pro[4] = temp; for (i =
   1; i < 5 ; i++){
       anumcp[i-1] = anumcp[i];
   }     anumcp[4]     =
   temp2;
   printf("\nAR-SHIFT:      ");
   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){
          printf("\n-->");
          arshift();
          q = anum[i]; } else if(anum[i]
      == 1 && q == 0){
         printf("\n-->");
         printf("\nSUB          B:          ");
         add(bcomp);
         arshift();
```

```
            q  =  anum[i];
         }
         else{      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:

OUTPUT:-

BOOTH'S MULTIPLICATION ALGORITHM

```
Enter two numbers to multiply:
Both must be less than 16
Enter A: 10
Enter B: 2
Expected product = 20

Binary Equivalents are:
A = 01010
B = 00010
B'+ 1 = 11110



-->
AR-SHIFT: 00000:00101
-->
SUB B: 11110:00101
AR-SHIFT: 11111:00010
-->
ADD B: 00001:00010
AR-SHIFT: 00000:10001
-->
SUB B: 11110:10001
AR-SHIFT: 11111:01000
-->
ADD B: 00001:01000
AR-SHIFT: 00000:10100
```

Product is = 0000010100

Conclusion -

This experiment with Booth's algorithm has highlighted its significance in optimizing binary multiplication .Booth's algorithm efficiently reduces the number of partial products and minimizes the overall number of operations required for multiplication. This not only enhances computational speed but also reduces hardware complexity. Booth's algorithm is a powerful tool for optimizing multiplication processes and is an essential concept in digital arithmetic. Our experiment has successfully demonstrated its practical applicability in computer architecture and digital circuit design.

Experiment No. 8
Implement Restoring algorithm using c-programming
Name: kiran vishnu dhuri
Roll Number: 07
Date of Performance:
Date of Submission:

Aim: To implement Restoring division algorithm using c-programming.

Objective -

1. To understand the working of Restoring division algorithm.
2. To understand how to implement Restoring division algorithm using c-programming.

Theory:

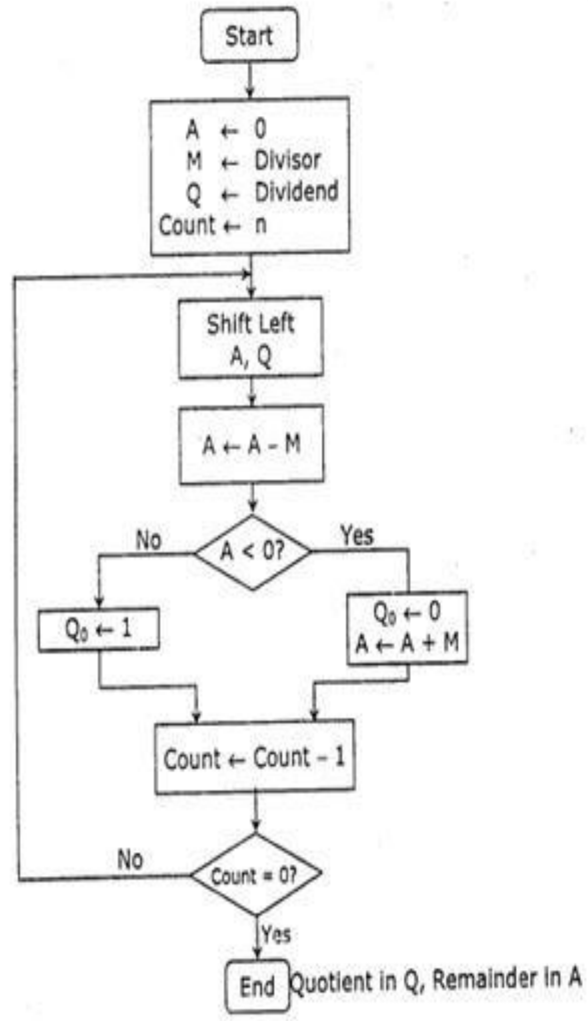
1) The divisor is placed in M register, the dividend placed in Q register.

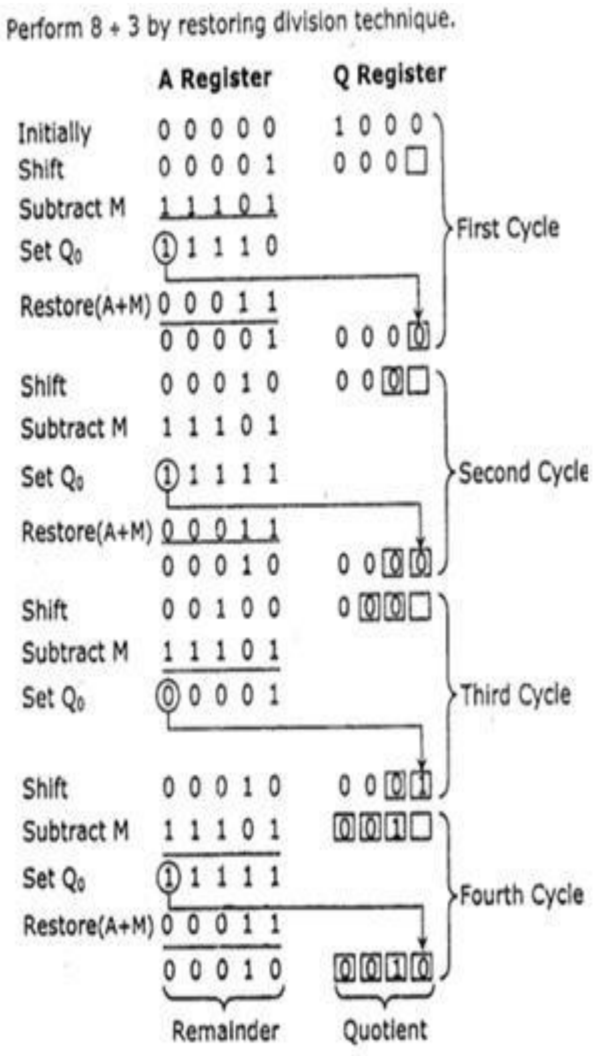
2) At every step, the A and Q registers together are shifted to the left by 1-bit

3) M is subtracted from A to determine whether A divides the partial remainder. If it does, then Q0 set to 1-bit. Otherwise, Q0 gets a 0 bit and M must be added back to A to restore the previous value.

4) The count is then decremented and the process continues for n steps. At the end, the quotient is in the Q register and the remainder is in the A register.

Flowchart




Perform 8 ÷ 3 by restoring division technique.

	A Register	Q Register	
Initially	0 0 0 0 0	1 0 0 0	First Cycle
Shift	0 0 0 0 1	0 0 0 □	
Subtract M	1 1 1 0 1		
Set Q_0	① 1 1 1 0		
Restore(A+M)	0 0 0 1 1		
	0 0 0 0 1	0 0 0 0	
Shift	0 0 0 1 0	0 0 0 □	Second Cycle
Subtract M	1 1 1 0 1		
Set Q_0	① 1 1 1 1		
Restore(A+M)	0 0 0 1 1		
	0 0 0 1 0	0 0 0 0	
Shift	0 0 1 0 0	0 0 0 □	Third Cycle
Subtract M	1 1 1 0 1		
Set Q_0	⓪ 0 0 0 1		
Shift	0 0 0 1 0	0 0 0 1	
Subtract M	1 1 1 0 1	0 0 1 □	Fourth Cycle
Set Q_0	① 1 1 1 1		
Restore(A+M)	0 0 0 1 1		
	0 0 0 1 0	0 0 1 0	
	Remainder	Quotient	

Program#include

<stdio.h> #include

<stdlib.h>

int dec_bin(int, int []);

int twos(int [], int []);

int left(int [], int []);

```
int add(int [], int []);

int main()

{ int a, b, m[4]={0,0,0,0}, q[4]={0,0,0,0}, acc[4]={0,0,0,0}, m2[4], i, n=4;

  printf("Enter the Dividend: "); scanf("%d", &a); printf("Enter the

  Divisor: "); scanf("%d", &b);

  dec_bin(a, q); dec_bin(b, m);

  twos(m, m2);

  printf("\nA\tQ\tComments\n")

  ; for(i=3; i>=0; i--)

  {

    printf("%d", acc[i]);

  } printf("\t");

  for(i=3; i>=0; i--

  )

  {

    printf("%d", q[i]);

  }

  printf("\tStart\n");

  while(n>0)

  {     left(acc,     q);

    for(i=3; i>=0; i--)

    {
```

```
   printf("%d",
acc[i]); } printf("\t");
for(i=3; i>=1; i--)
{
   printf("%d", q[i]);
}
printf("_\tLeft Shift A,Q\n");
add(acc, m2);
for(i=3; i>=0; i--)
{
   printf("%d",
acc[i]); } printf("\t");
for(i=3; i>=1; i--)
{
   printf("%d", q[i]);
}
printf("_\tA=A-M\n");
if(acc[3]==0)
{ q[0]=1; for(i=3;
   i>=0; i--)
   {
      printf("%d",
   acc[i]); } printf("\t");
   for(i=3; i>=0; i--)
```

```
        {
           printf("%d",
    q[i]); }
    printf("\tQo=1\n"); }
    else
    { q[0]=0;
        add(acc,
        m);
        for(i=3; i>=0; i--)
        {
           printf("%d",
        acc[i]); } printf("\t");
        for(i=3; i>=0; i--)
        {
           printf("%d", q[i]);
        }
        printf("\tQo=0; A=A+M\n");
    } n--
    ; }

printf("\nQuotient = ");
for(i=3; i>=0; i--)
{
        printf("%d", q[i]);
}
```

```
    printf("\tRemainder = ");
    for(i=3; i>=0; i--)
    {
           printf("%d", acc[i]);
    } printf("\n");
    return 0;
}

int dec_bin(int d, int m[])
{
    int b=0, i=0;
    for(i=0; i<4; i++)
    {
        m[i]=d%2;
        d=d/2;
    }
    return 0;
}
int twos(int m[], int m2[])
{
    int i, m1[4];
    for(i=0; i<4; i++)
    { if(m[i]==0)
        {
```

```
        m1[i]=1
      ; } else
      {
        m1[i]=0;
      }
    }
    for(i=0; i<4; i++)
    {
      m2[i]=m1[i];
    }
    if(m2[0]==0)
    {
      m2[0]=1
    ; } else {

      m2[0]=0;
      if(m2[1]==0)
      {
        m2[1]=1
      ; } else
      {
        m2[1]=0;
        if(m2[2]==0)
        {
            m2[2]=1
          ; } else
          {
            m2[2]=0;
            if(m2[3]==0)
            {
              m2[3]=1
            ; } else
            {
              m2[3]=0;
            }
          }
        }
      }
   return 0;
}

int left(int acc[], int q[])
{ int i; for(i=3;
   i>0; i--)
   { acc[i]=acc[i-1];
   } acc[0]=q[3];
   for(i=3; i>0; i--)
   { q[i]=q[i-1];
```

```
    }
}


int add(int acc[], int m[])
{
  int i, carry=0;
  for(i=0; i<4; i++)
  {
    if(acc[i]+m[i]+carry==0
    )
    {
      acc[i]=0; carry=0; } else
    if(acc[i]+m[i]+carry==1)
    { acc[i]=1; carry=0; } else
    if(acc[i]+m[i]+carry==2)
    { acc[i]=0; carry=1; } else
    if(acc[i]+m[i]+carry==3)
    {
      acc[i]=1
      ;
      carry=1
      ;
    }
  }
  return 0;
}
```

Output -

```
Enter the Dividend: 12

Enter the Divisor: 2
A       Q       Comments

0000    1100    Start

0001    100_    Left Shift A,Q

1111    100_    A=A-M

0001    1000    Qo=0; A=A+M

0011    000_    Left Shift A,Q

0001    000_    A=A-M

0001    0001    Qo=1

0010    001_    Left Shift A,Q

0000    001_    A=A-M

0000    0011    Qo=1

0000    011_    Left Shift A,Q

1110    011_    A=A-M

0000    0110    Qo=0; A=A+M
Quotient = 0110        Remainder = 0000
```

Conclusion -

In conclusion, the implementation of the Restoring Division Algorithm using the C programming language offers a powerful method for efficient and precise division operations. Through this project, we have explored the intricacies of this algorithm, emphasizing its significance in modern computing. By leveraging C's robust features and versatility, we have successfully translated the theoretical framework of the Restoring Division Algorithm into a functional codebase. As a result, we have not only deepened our understanding of division algorithms but also honed our programming skills.

Experiment No. 9
Implement Non-Restoring algorithm using c-programming
Name: kiran vishnu dhuri
Roll Number:07
Date of Performance:
Date of Submission:

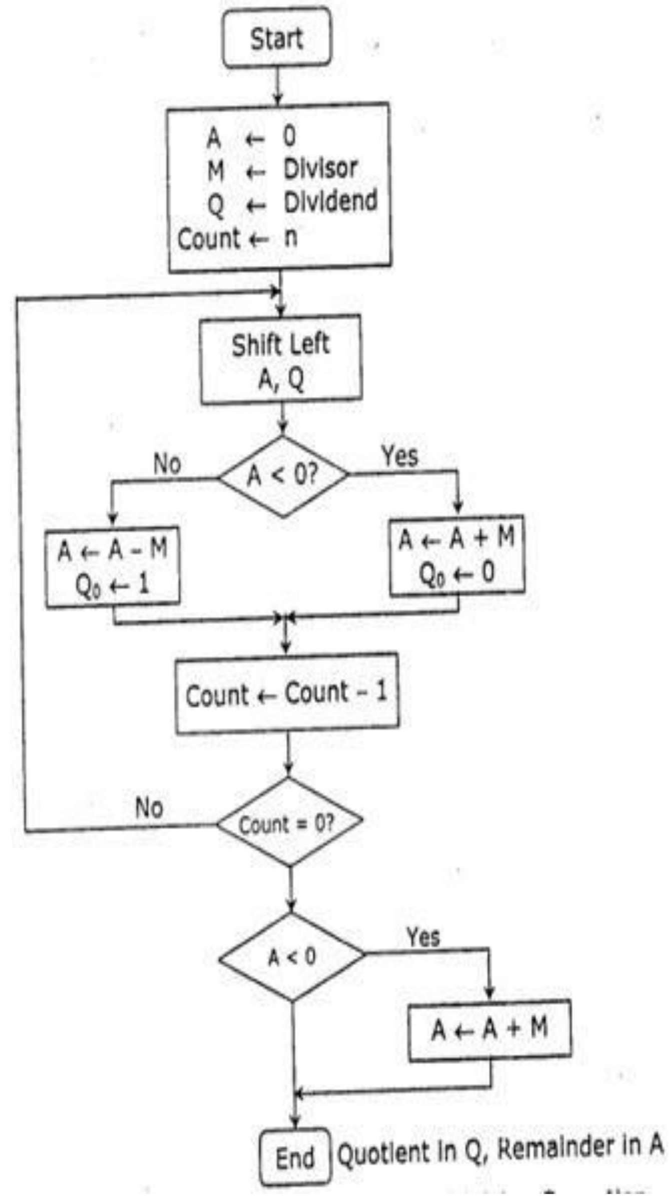
Aim - To implement Non-Restoring division algorithm using c-programming.

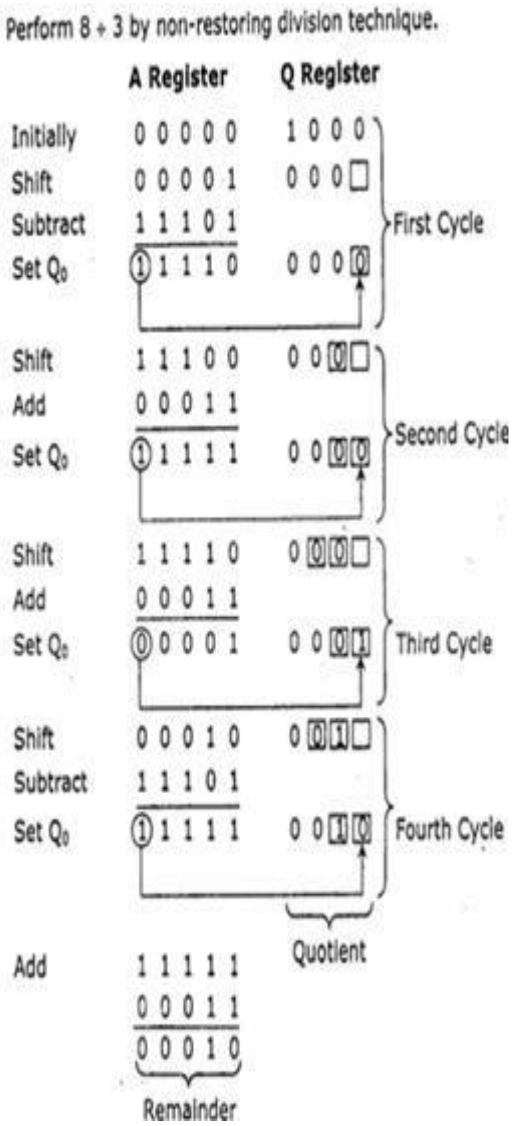
Objective -

1. To understand the working of Non-Restoring division algorithm.
2. To understand how to implement Non-Restoringdivision algorithm using cprogramming.

Theory:

In each cycle content of the register, A is first shifted and then the divisor is added or subtracted with the content of register A depending upon the sign of A. In this, there is no need of restoring, but if the remainder is negative then there is a need of restoring the remainder. This is the faster algorithm of division.





Perform 8 ÷ 3 by non-restoring division technique.

	A Register	Q Register	
Initially	0 0 0 0 0	1 0 0 0	First Cycle
Shift	0 0 0 0 1	0 0 0 □	
Subtract	1 1 1 0 1		
Set Q_0	1 1 1 1 0	0 0 0 0	
Shift	1 1 1 0 0	0 0 0 □	Second Cycle
Add	0 0 0 1 1		
Set Q_0	1 1 1 1 1	0 0 0 0	
Shift	1 1 1 1 0	0 0 0 □	Third Cycle
Add	0 0 0 1 1		
Set Q_0	0 0 0 0 1	0 0 0 1	
Shift	0 0 0 1 0	0 0 1 □	Fourth Cycle
Subtract	1 1 1 0 1		
Set Q_0	1 1 1 1 1	0 0 1 0	
		Quotient	
Add	1 1 1 1 1		
	0 0 0 1 1		
	0 0 0 1 0		
	Remainder		

Program -

```
#include        <stdio.h>
#include <stdlib.h>


int dec_bin(int, int []);
int twos(int [], int []);
int left(int [], int []);

int add(int [], int []);


int main()

{ int a, b, m[4]={0,0,0,0}, q[4]={0,0,0,0}, acc[4]={0,0,0,0}, m2[4], i, n=4;

   printf("Enter the Dividend: "); scanf("%d", &a); printf("Enter the

   Divisor: "); scanf("%d", &b); dec_bin(a, q); dec_bin(b, m); twos(m,

   m2);

   printf("\nA\tQ\tComments\n");

   for(i=3; i>=0; i--)

   {

      printf("%d", acc[i]);

   }

   printf("\t");

   for(i=3; i>=0; i--)

   {

      printf("%d", q[i]);

   }

   printf("\tStart\n");

   while(n>0)

   {

      left(acc,          q);

      for(i=3; i>=0; i--)

      {
```

```
    printf("%d", acc[i]);
  }
  printf("\t");
  for(i=3; i>=1; i--)
  {
    printf("%d",   q[i]);
  }
  printf("_\tLeft       Shift       A,Q\n");
  add(acc, m2);

  for(i=3; i>=0; i--)
  {
    printf("%d", acc[i]);
  }
  printf("\t");
  for(i=3; i>=1; i--)
  {
    printf("%d", q[i]); }
  printf("_\tA=A-
  M\n"); if(acc[3]==0)
  { q[0]=1; for(i=3;
    i>=0; i--)
    {
      printf("%d", acc[i]);
    }
```

```
        printf("\t");
        for(i=3; i>=0; i--)
        {
           printf("%d",   q[i]);
        }
        printf("\tQo=1\n");
     }
     else
     { q[0]=0; add(acc,
        m); for(i=3;
        i>=0; i--)
        {
           printf("%d", acc[i]);
        }
        printf("\t");
        for(i=3; i>=0; i--)
        {
           printf("%d",      q[i]);      }
     printf("\tQo=0; A=A+M\n"); }
     n--;
  }
  printf("\nQuotient       =       ");
  for(i=3; i>=0; i--)
  {
        printf("%d", q[i]); }
   printf("\tRemainder = ");
   for(i=3; i>=0; i--)
   {
        printf("%d", acc[i]);
   }
   printf("\n");
   return   0;
}

int dec_bin(int d, int m[])
{ int b=0, i=0;
   for(i=0; i<4;
   i++)
   { m[i]=d%2;
   d=d/2; }
   return   0;
}

int twos(int m[], int m2[])
{ int i, m1[4];
   for(i=0; i<4;
   i++)
   {
      if(m[i]==0)
      { m1[i]=1;
      }
      else
      { m1[i]=0;
      } } for(i=0;
   i<4; i++)
   {  m2[i]=m1[i];
   } if(m2[0]==0)
   { m2[0]=1;
   }
   else
   { m2[0]=0;
      if(m2[1]==0)
      { m2[1]=1;
      }
      else
      { m2[1]=0;
        if(m2[2]==0
        ) {
           m2[2]=1;
        }
        else
```

```
        { m2[2]=0;
          if(m2[3]==0
          ) { m2[3]=1;
          }
          else {
          m2[3]=0;
          }
        }
      }
   }
   return   0;
}

int left(int acc[], int q[])
{
   int   i;   for(i=3;
   i>0; i--)
   {
      acc[i]=acc[i-1];
   }
   acc[0]=q[3];
   for(i=3; i>0; i--)
   {
      q[i]=q[i-1];
   }
}

int add(int acc[], int m[])
{
  int     i,     carry=0;
  for(i=0; i<4; i++)
  {
   if(acc[i]+m[i]+carry==0)
   {
     acc[i]=0;
   carry=0; }
   else if(acc[i]+m[i]+carry==1)
   {
     acc[i]=1;
   carry=0; }
   else if(acc[i]+m[i]+carry==2)
   {
     acc[i]=0;
   carry=1; }
   else if(acc[i]+m[i]+carry==3)
   {
     acc[i]=1;
     carry=1;
```

```
    }
  }
  return 0;
}
```

Output:

```
Enter the Dividend: 10
Enter the Divisor: 2
A       Q       Comments
0000    1010    Start
0001    010_    Left Shift A,Q
1111    010_    A=A-M
0001    0100    Qo=0; A=A+M
0010    100_    Left Shift A,Q
0000    100_    A=A-M
0000    1001    Qo=1
0001    001_    Left Shift A,Q
1111    001_    A=A-M
0001    0010    Qo=0; A=A+M
0010    010_    Left Shift A,Q
0000    010_    A=A-M
0000    0101    Qo=1
Quotient = 0101         Remainder = 0000
```

Conclusion -

In conclusion, the Non-Restoring division algorithm serves as an efficient method for performing division operations in computer programming. Through the implementation of this algorithm using the C programming language, we have explored a robust approach to dividing numbers that minimizes the number of iterations required and optimizes computational efficiency. By carefully designing the algorithm and writing a C program that incorporates it, we have demonstrated the practicality and effectiveness of this approach in performing complex mathematical operations. The successful execution of this project underscores the significance of algorithmic efficiency in enhancing the performance of computational tasks, paving the way for further advancements and applications in the field of computer science and software development.

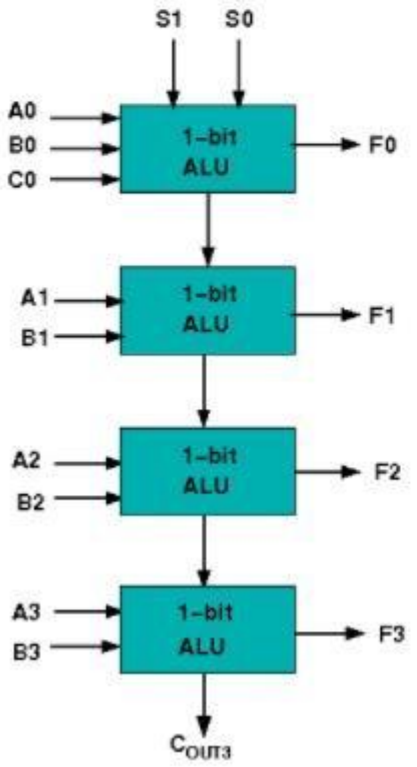
Experiment No.10
Implement ALU design.
Name: kiran vishnu dhuri
Roll Number:07
Date of Performance:
Date of Submission:

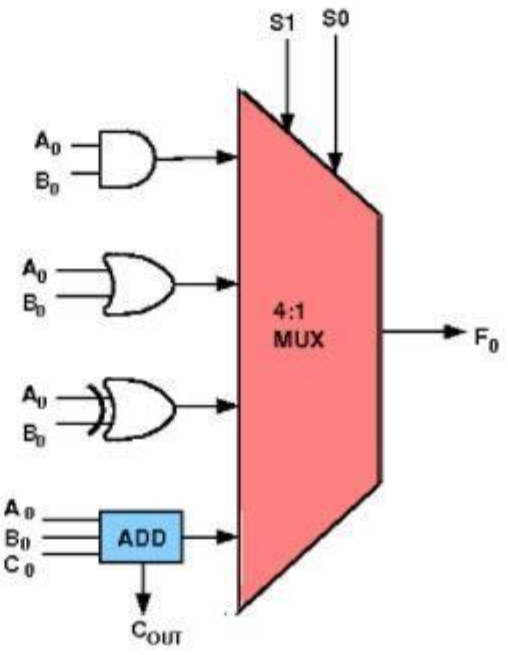
Objective : Objective of 4 bit arithmetic logic unit (with AND, OR, XOR, ADD operation):

1. To understand behaviour of arithmetic logic unit from working module.
2. To Design an arithmetic logic unit for given parameter.

Theory:

ALU or Arithmetic Logical Unit is a digital circuit to do arithmetic operations like addition, subtraction,division, multiplication and logical oparations 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 S1 and S0 the circuit operates as follows:

for Control signal S1 = 0 , S0 = 0, the output is A And B,

for Control signal S1 = 0 , S0 = 1, the output is A Or B,

for Control signal S1 = 1 , S0 = 0, the output is A Xor B,

for Control signal S1 = 1 , S0 = 1, the output is A Add B.

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

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

MODE SELECT	F_N FOR ACTIVE HIGH OPERANDS	
INPUTS	LOGIC	ARITHMETIC (NOTE 2)

S3	S2	S1	S0	(M = H)	(M = L) (Cn=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⊕ 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⊕ 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

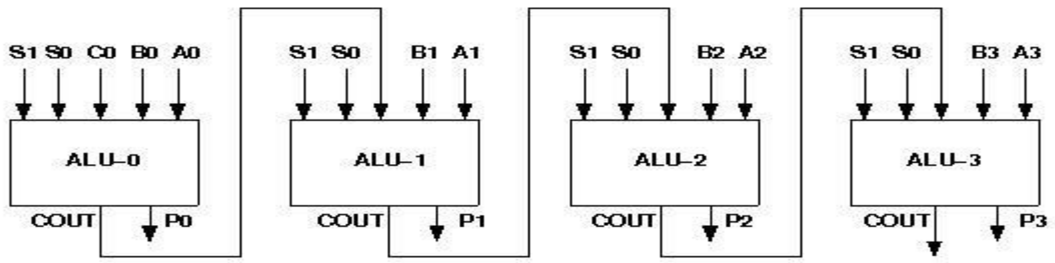
Procedure

1) Start the simulator as directed.This simulator supports 5-valued logic.
2) To design the circuit we need 4 1-bit ALU, 11 Bit switch (to give input,which will toggle its value with a double click), 5 Bit displays (for seeing output), wires.
3) The pin configuration of a component is shown whenever the mouse is hovered on any canned component of the palette. Pin numbering starts from 1 and from the bottom left corner (indicating with the circle) and increases anticlockwise.
4) For 1-bit ALU input A0 is in pin-9,B0 is in pin-10, C0 is in pin-11 (this is input carry), for selection of operation, S0 is in pin-12, S1 is in pin-13, output F is in pin-8 and output carry is pin-7

5) Click on the 1-bit ALU component (in the Other Component drawer in the pallet) and then click on the position of the editor window where you want to add the component

   (no drag and drop, simple click will serve the purpose), likewise add 3 more 1-bit ALU (from the Other Component drawer in the pallet), 11 Bit switches and 5 Bit Displays (from Display and Input drawer of the pallet,if it is not seen scroll down in the drawer), 3 digital display and 1 bit Displays (from Display and Input drawer of the pallet,if it is not seen scroll down in the drawer)
6) To connect any two components select the Connection menu of Palette, and then click on the Source terminal and click on the target terminal. According to the circuit diagram connect all the components. Connect the Bit switches with the inputs and Bit displays component with the outputs. After the connection is over click the selection tool in the pallete.
7) See the output, in the screenshot diagram we have given the value of S1 S0=11 which will perform add operation and two number input as A0 A1 A2 A3=0010 and B0 B1 B2 B3=0100 so get output F0 F1 F2 F3=0110 as sum and 0 as carry which is indeed an add operation.you can also use many other combination of different values and check the result. The operations are implemented using the truth table for 4 bit ALU given in the theory.

Circuit diagram of 4 bit ALU:



Components required :

To build any 4 bit ALU, we need :

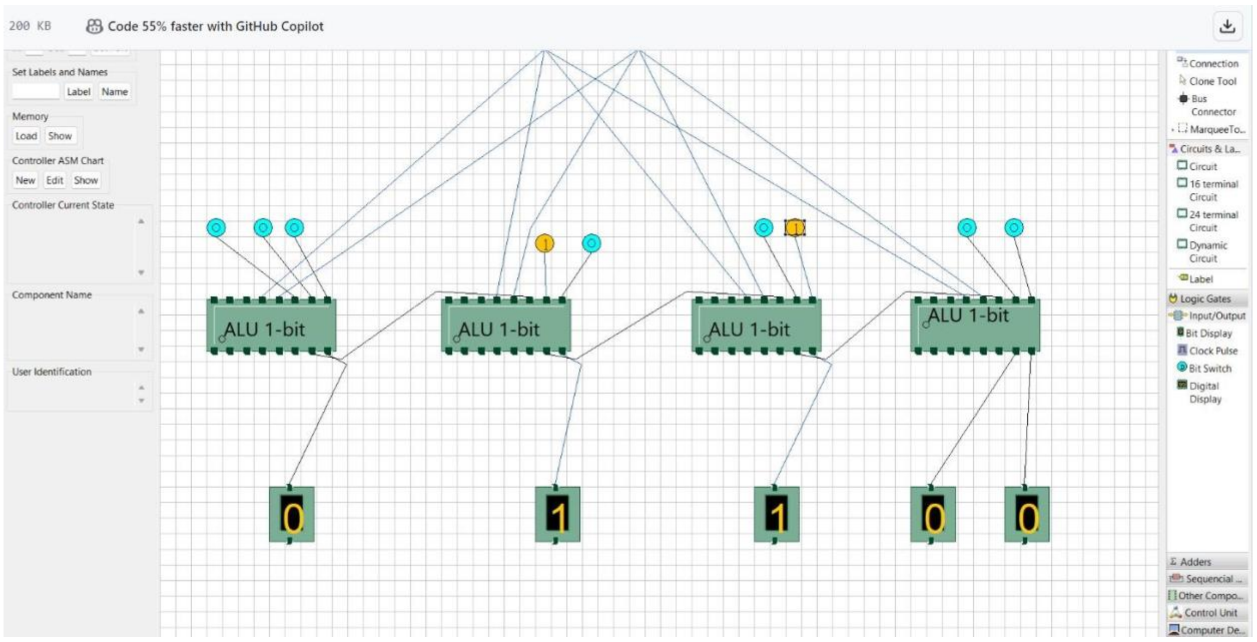
➢ AND gate, OR gate, XOR gate

➢ Full Adder,

➢ 4-to-1 MUX ➢

Wires to connect.

Screenshots of ALU design:



Conclusion:

In conclusion, the implementation of an Arithmetic Logic Unit (ALU) design is a critical step towards enhancing computational capabilities and efficiency within a digital system. Through careful planning, meticulous design, and rigorous testing, engineers can ensure the successful integration of ALU, thereby enabling the processing of complex arithmetic and logical operations with remarkable speed and accuracy. By leveraging the advancements in hardware and software technologies, the ALU design not only facilitates the execution of diverse computing tasks but also serves as a fundamental building block for the development of more sophisticated and powerful computing system