INDEX

Experiment No. 1
Truth table of various logic gates using ICs.
Name:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

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

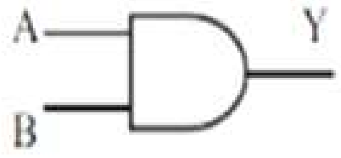
Objective -

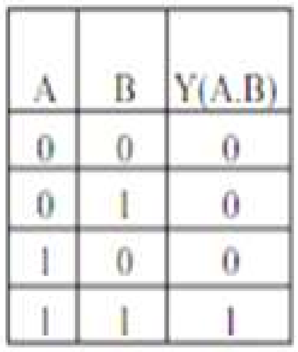
- Understand how to use the breadboard to patch up, test your logic design and debug it.
- The principal objective of this experiment is to fully understand the function and use of logic gates.
- Understand how to implement simple circuits based on a schematic diagram using logic gates.

Components required -
1. IC’s 7408, 7432, 7404
2. Bread Board.
3. Connecting wires.

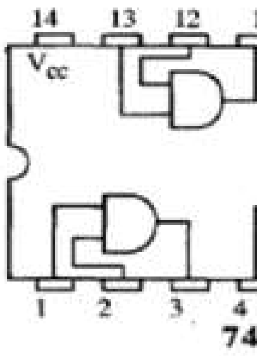
Theory -
In digital electronics, a gate is logic circuits with one output and one or more inputs. Logic gates are available as integrated circuits.
AND gate :
AND gate performs logical multiplication, more commonly known as AND operation. The AND gate output will be in high state only when all the inputs 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 -

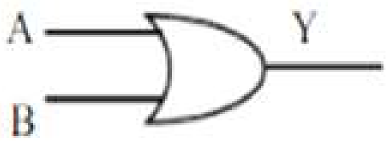


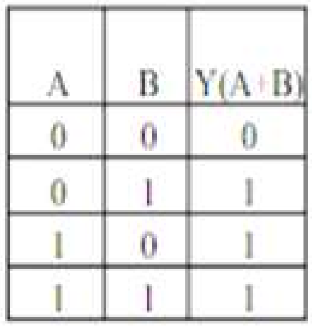


A	B	Y(A.B)
0	0	0
0	1	0
1	0	0
1	1	1

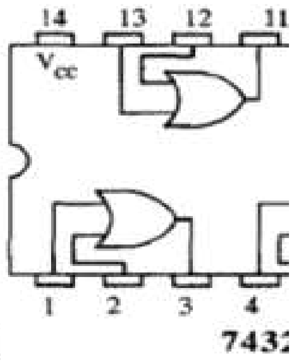


OR Gate -

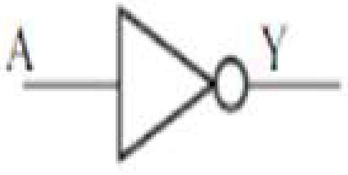


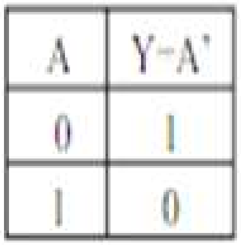


A	B	Y(A+B)
0	0	0
0	1	1
1	0	1
1	1	1

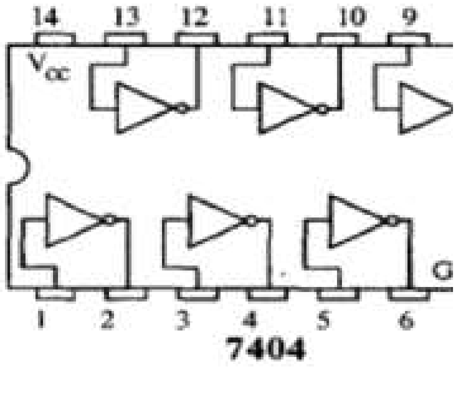


NOT Gate -





A	Y=A'
0	1
1	0

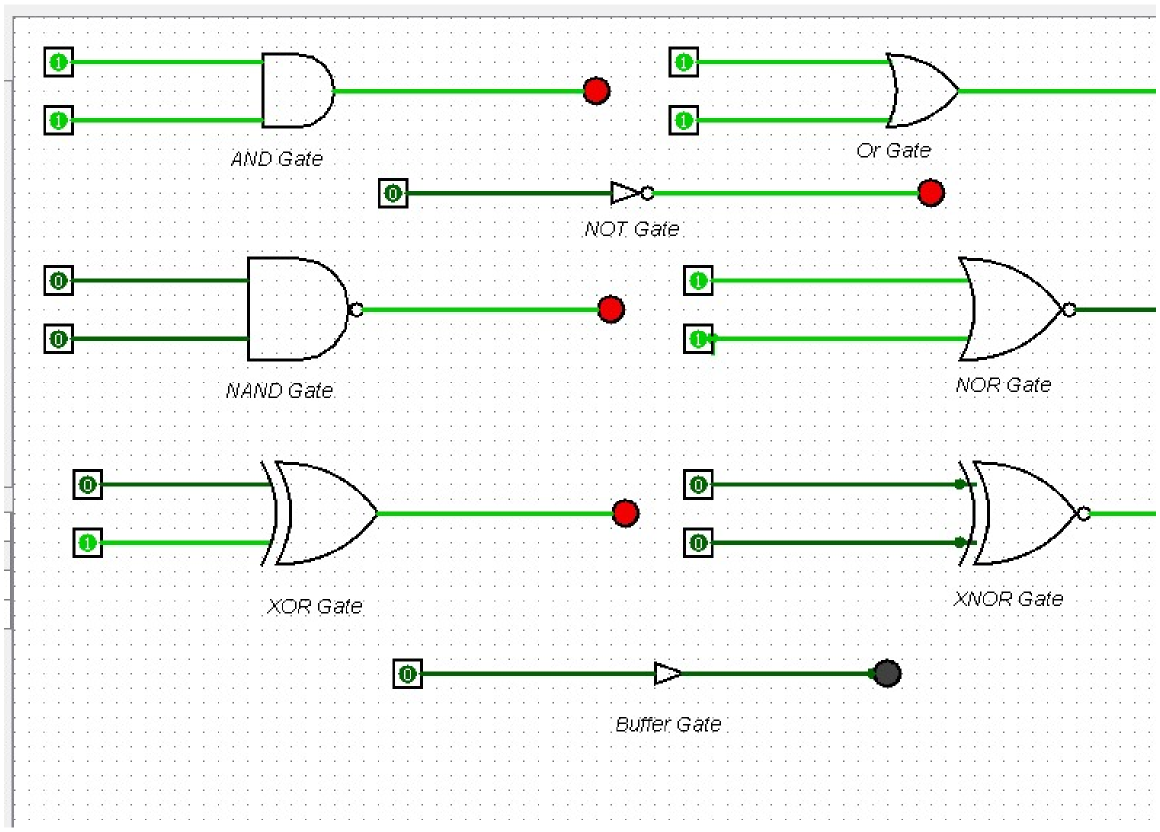


Procedure:

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) inout of the IC before feeding the inputs.
3.Set up the circuits and observe the outputs.

OUTPUT



Conclusion -
I have learned some basic gates like “and” “or” “nand” “nor” “not” “xor” “xnor”.Hence the above experiment is verified and performed.

Experiment No. 2
Basic gates using universal gates.
Name:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

Aim - To realize the gates using universal gates.

Objective -

- To study the realization of basic gates using universal gates.
- 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.

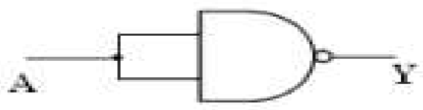
Components required -

1. IC's 7400(NAND) 7402(NOR)
2. Bread Board.
3. Connecting wires.

Circuit Diagram -

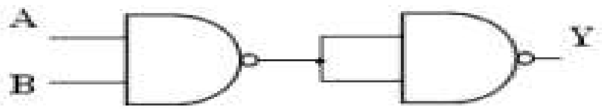
Implementation using NAND gate:

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



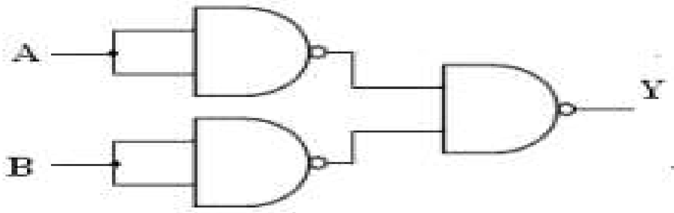
A	Y
0	1
1	0

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



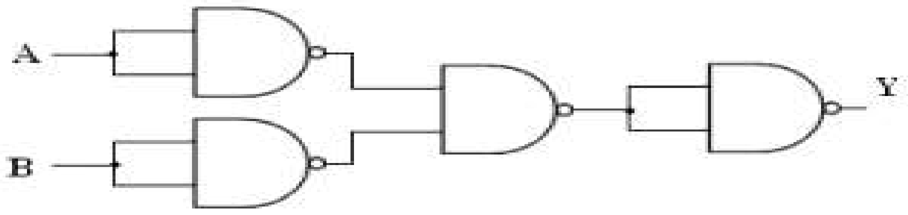
A	B
0	0
0	1
1	0
1	1

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



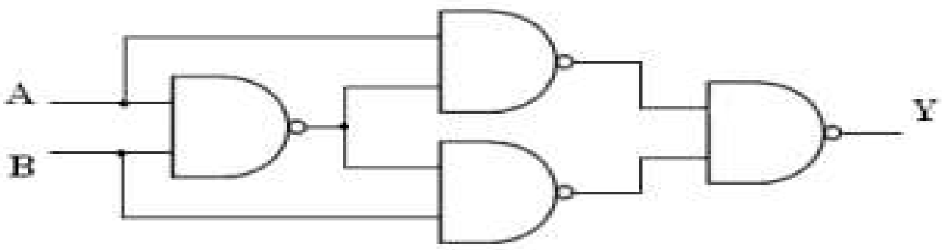
A	B
0	0
0	1
1	0
1	1

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



A	B
0	0
0	1
1	0
1	1

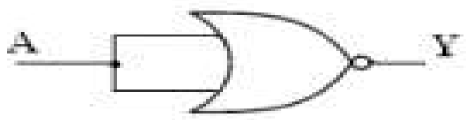
(e) Ex-OR gate: $Y = A \oplus B$



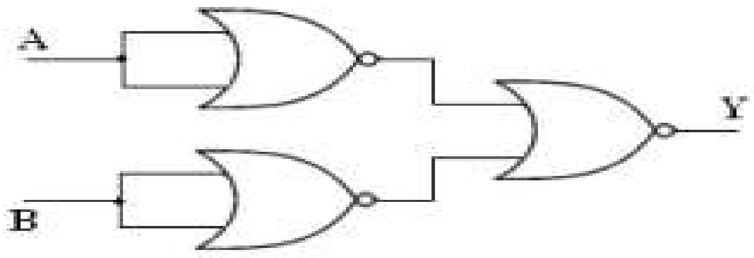
A
0
0
1
1

Implementation using NOR gate:

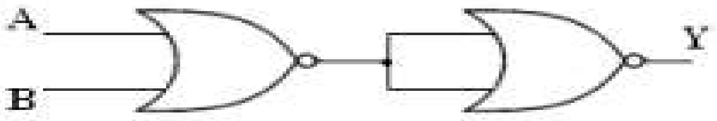
(a) NOT gate: $Y = A'$



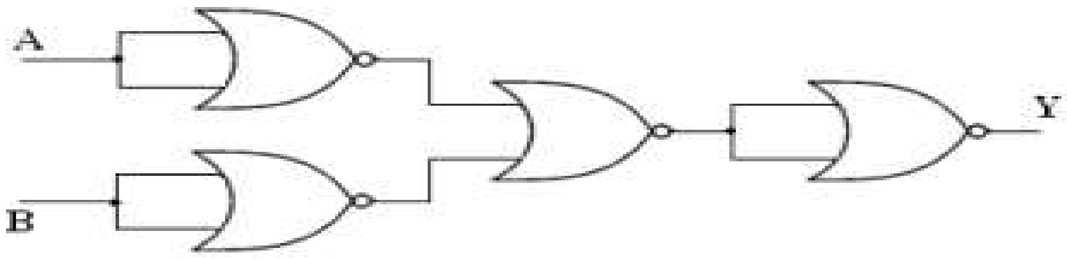
(b) AND gate: $Y = A \cdot B$



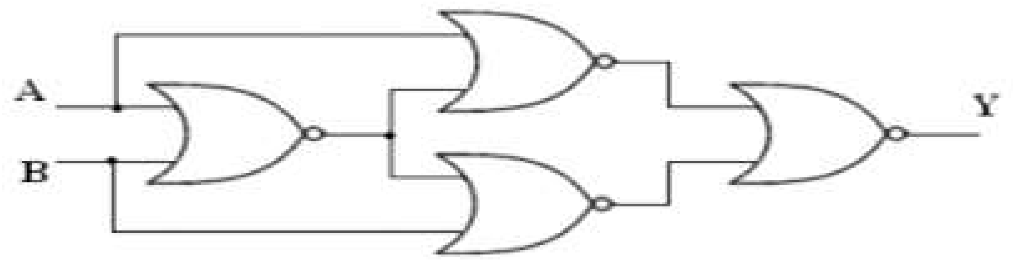
(c) OR gate: $Y = A + B$



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

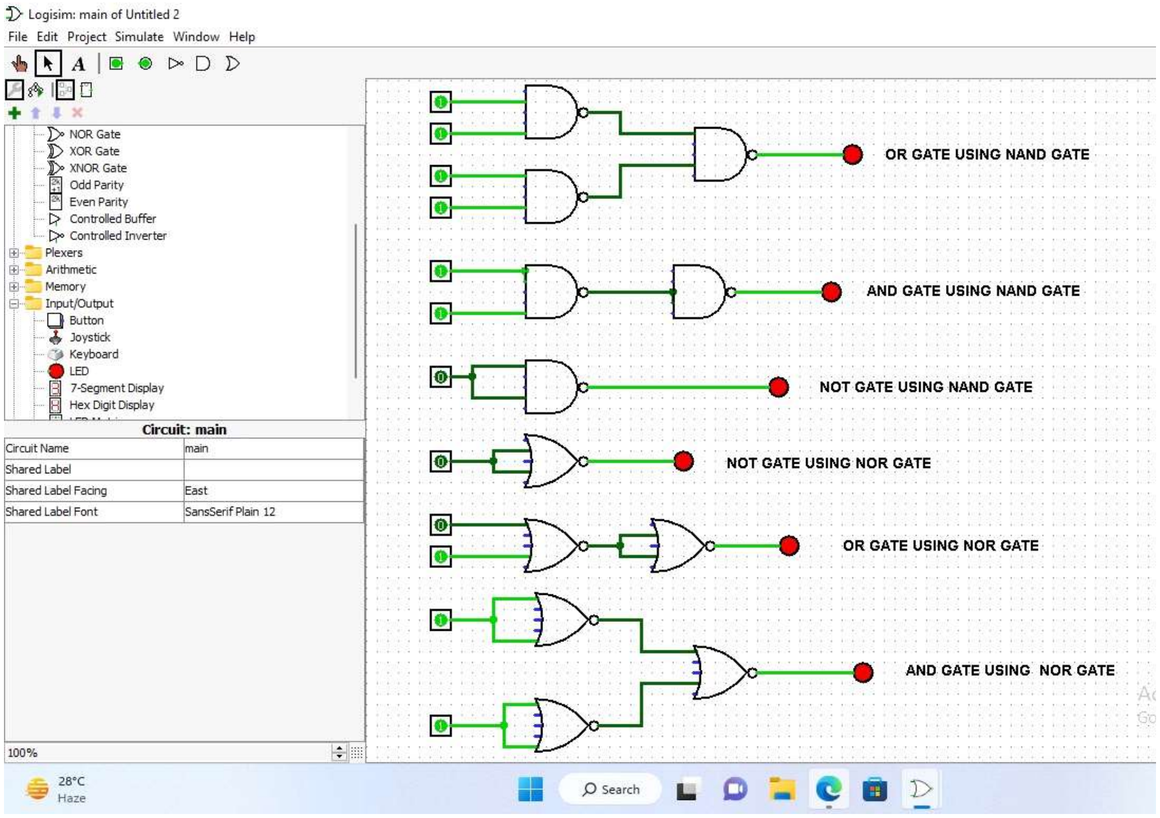


(e) Ex-NOR gate: $Y = A \odot B = (A \oplus B)'$



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



Circuit: main

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

Conclusion -

The experiment conducted on universal gates in Logisim has provided valuable insights into the versatility and functionality of these essential digital logic components. We have demonstrated the ability of universal gates to perform a wide range of logical operations, showcasing their significance in modern digital circuit design. This experiment underscores the importance of understanding and utilizing universal gates in the field of digital electronics, paving the way for more efficient and versatile circuitry.

Experiment No. 3
To realize half adder and full adder.
Name:Singh Rahul Rammialn
Roll Number:56
Date of Performance:
Date of Submission:

Aim - To realize half adder and full adder.

Objective -

- The objective of this experiment is to understand the function of Half-adder, Full-adder, Half-subtractor and Full-subtractor.
- 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.

Sum =$A \oplus B$

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 half- Subtractor are

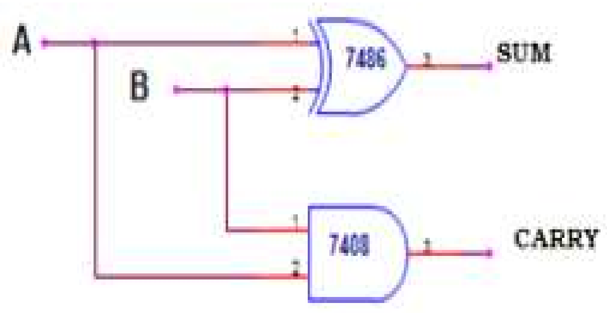
$$\text{Sum} = A \oplus B$$
$$\text{Carry} = A'\,B$$

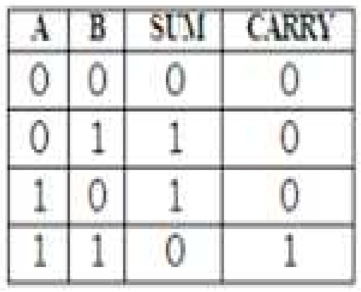
Subtracting two single-bit binary values, B, Cin from a single-bit value A produces a difference bit D and a borrow out Br bit. This is called full subtraction. The Boolean functions describing the full-subtractor are

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

Circuit Diagram and Truth Table -

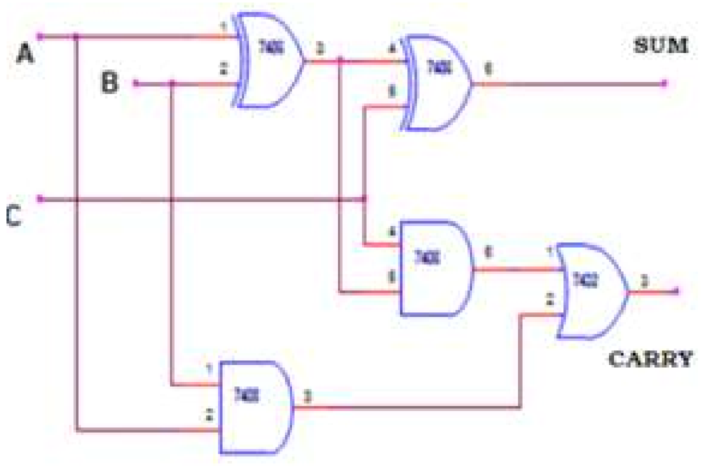
Half-adder

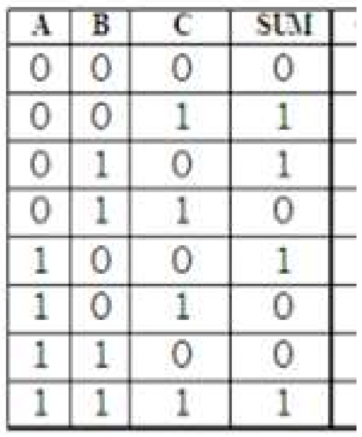




A	B	SUM	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full-adder

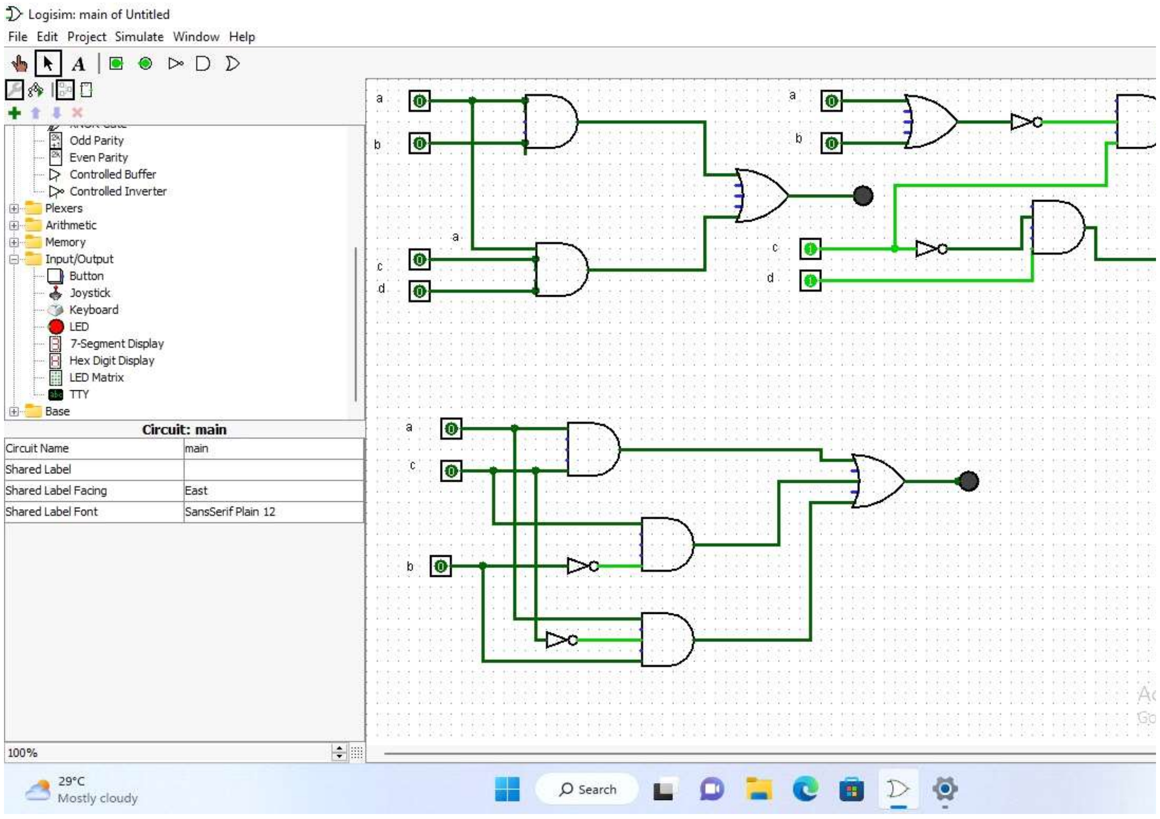




A	B	C	SUM
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

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



Circuit: main

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

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
Study of flip flop IC
Name:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

Aim - Study of flip flop IC
OBJECTIVES:-
1
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.
2
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 low-power 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.
3
Objective 6: Comparative Study of Flip-Flop IC Families
To compare and contrast different families of flip-flop ICs, such as TTL (Transistor-Transistor Logic) and CMOS (Complementary Metal-Oxide-Semiconductor), in terms of speed, power consumption, and noise immunity.
To analyze the trade-offs between various flip-flop architectures and recommend suitable choices based on specific application requirements.
By achieving these objectives, this experiment aims to provide a comprehensive understanding of flip-flop integrated circuits, their behavior, applications, and design considerations, contributing to the advancement of digital electronics and circuit design knowledge.

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
4
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

Experiment No. 5
Implement ripple carry adder

Name:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

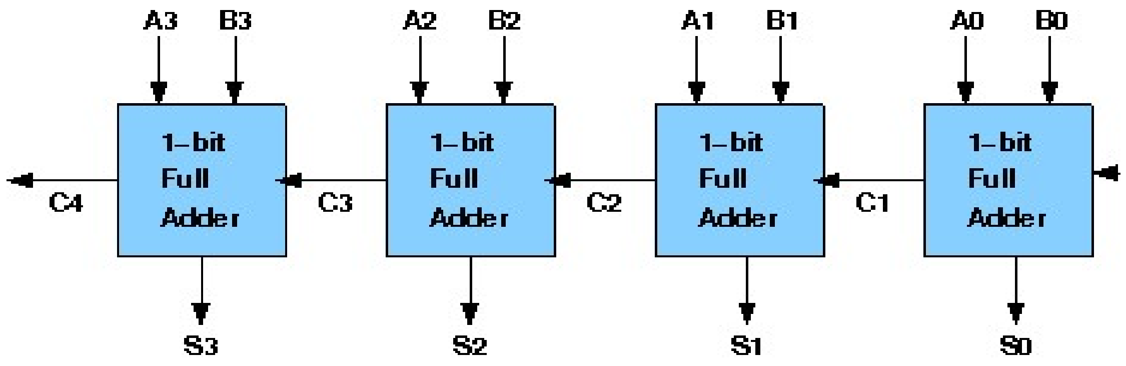
Aim: To implement ripple carry adder.

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

- examining the behavior of the working module to understand how the carry ripples through the adder stages
- to design a ripple carry adder using full adders to mimic the behavior of the working module
- 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

$$\begin{aligned}\text{Sum} &= ABC + ABC + ABC + ABC \\ &= (AB + AB)\,C + (AB + AB)\,C \\ &= (A \oplus B)\,C + (A \oplus B)\,C \\ &= A \oplus B \oplus C\end{aligned}$$

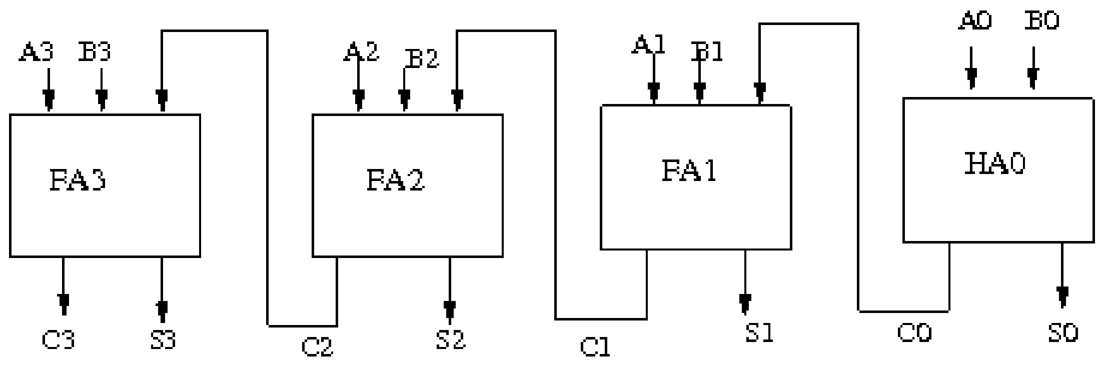
$$\begin{aligned}\text{Carry} &= ABC + ABC + ABC + ABC \\ &= AB + (AB + AB)\,C \\ &= AB + (A \oplus B)\,C\end{aligned}$$

Procedure:
Procedure to perform the experiment: Design of Ripple Carry Adders

- Start the simulator as directed. This simulator supports 5-valued logic.
- 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.
- 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.
- 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

- 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)
- 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.
- 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 -

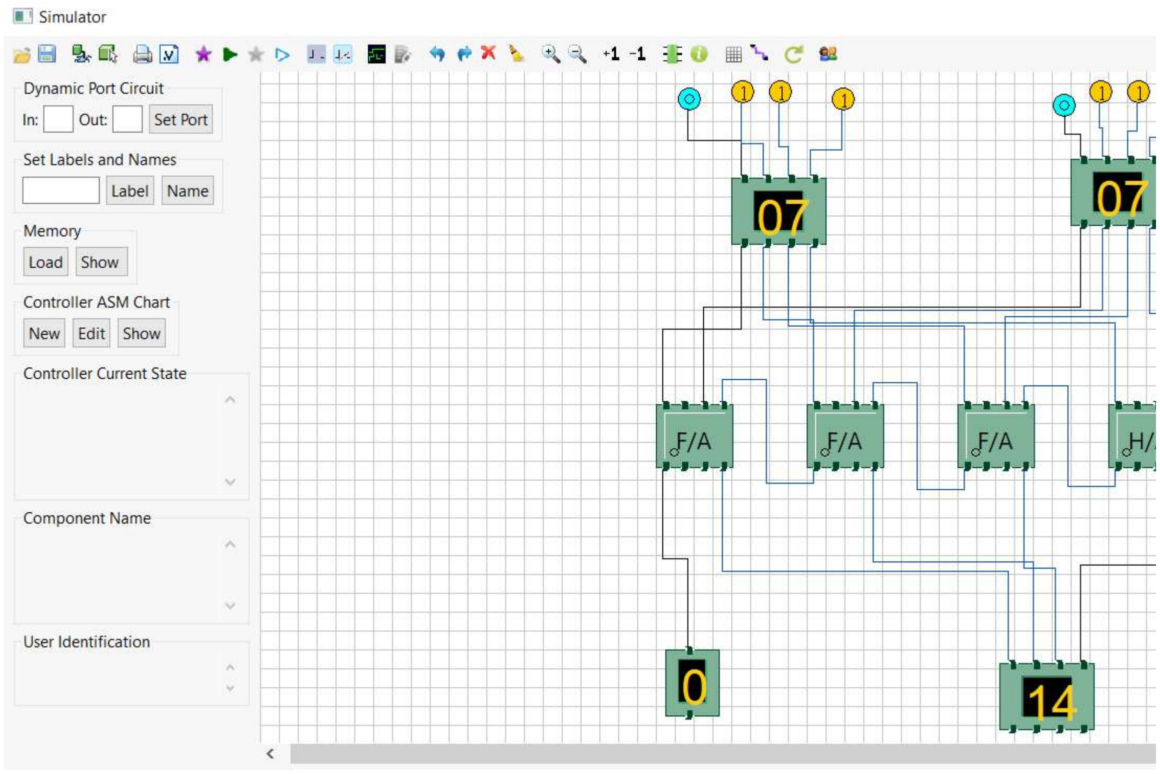
- 4 full-adders
- wires to connect
- LED display to obtain the output

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:

Our experiment on the Ripple Carry Adder in Logisim provided valuable insights into the fundamental principles of binary addition and digital circuitry. We observed how this basic adder architecture sequentially propagates carry bits, which can lead to increased latency for larger inputs.

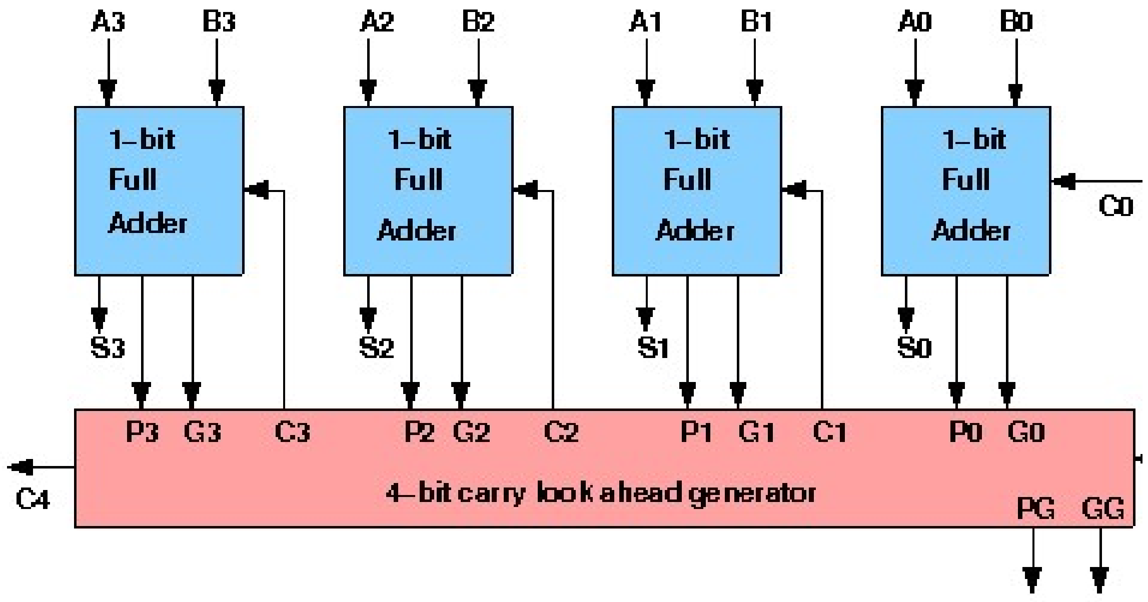
Experiment No.6
Implement Carry Look Ahead Adder.
Name:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

Aim: . To implement carry look ahead adder.

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

- To understanding behaviour of carry lookahead adder from module designed by the student as part of the experiment
- To understand the concept of reducing computation time with respect of ripple carry adder by using carry generate and propagate functions.
- 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:

$C1 = G0 + P0 \cdot C0$

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

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

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

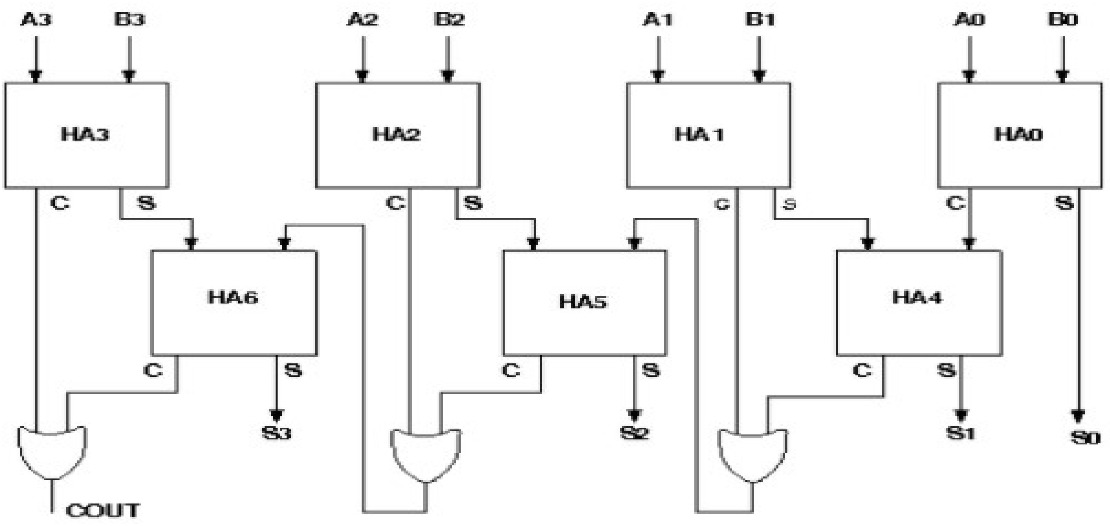
C0

Procedure:

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

- Start the simulator as directed. This simulator supports 5-valued logic.
- 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.
- 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.
- For half adder input is in pin-5,8 output sum is in pin-4 and carry is pin-1
- 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)
- 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.
- 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:

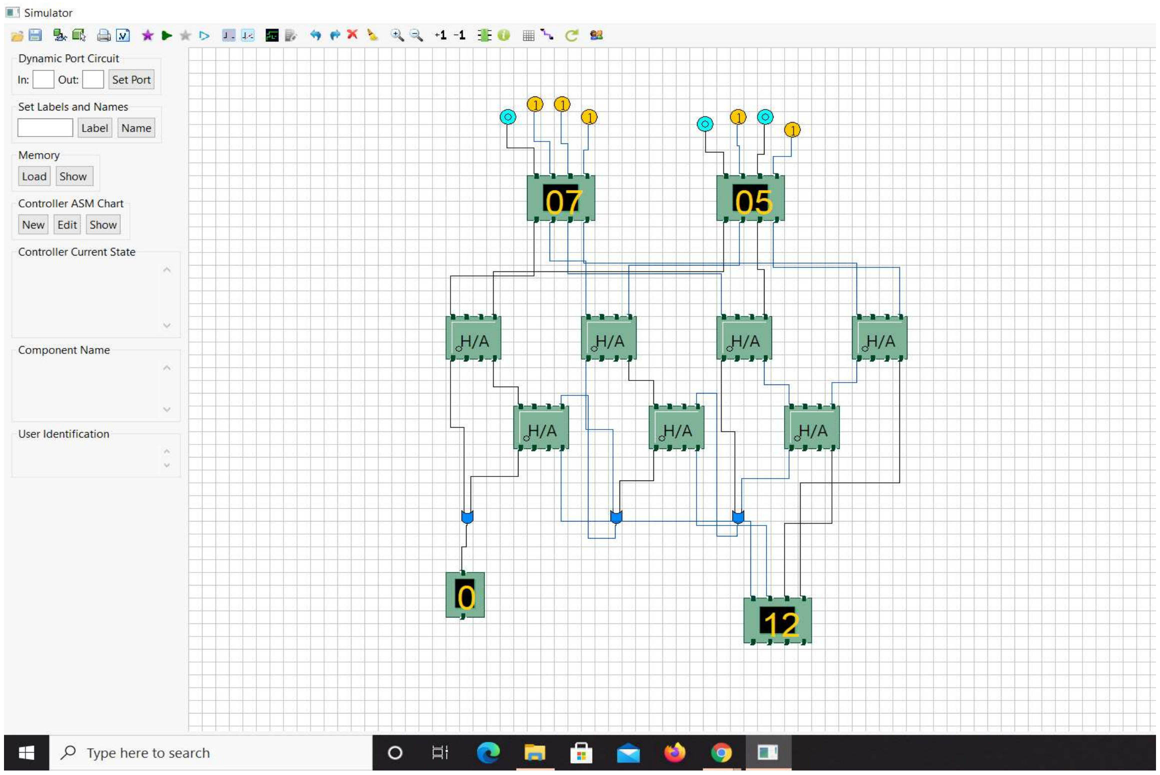


Components required:

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

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

Screenshots of Carry Look Ahead Adder:



Conclusion:

This experiment conducted on the Carry Look-Ahead Adder in Logisim has provided
valuable insights into the efficiency and functionality of this advanced digital circuit. We
have demonstrated that the Carry Look-Ahead Adder is a highly effective approach for
achieving fast addition of binary numbers. Its ability to minimize carry propagation delay,
resulting in reduced computation time, makes it a vital component in modern computer
architecture and arithmetic circuits. This experiment has not only reaffirmed the
importance of efficient adder designs but has also illustrated the practical
implementation of complex digital circuits within the realm of digital logic simulation.

Experiment No. 7
Implement Booth’s algorithm using c-programming
Name:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

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

Objective -

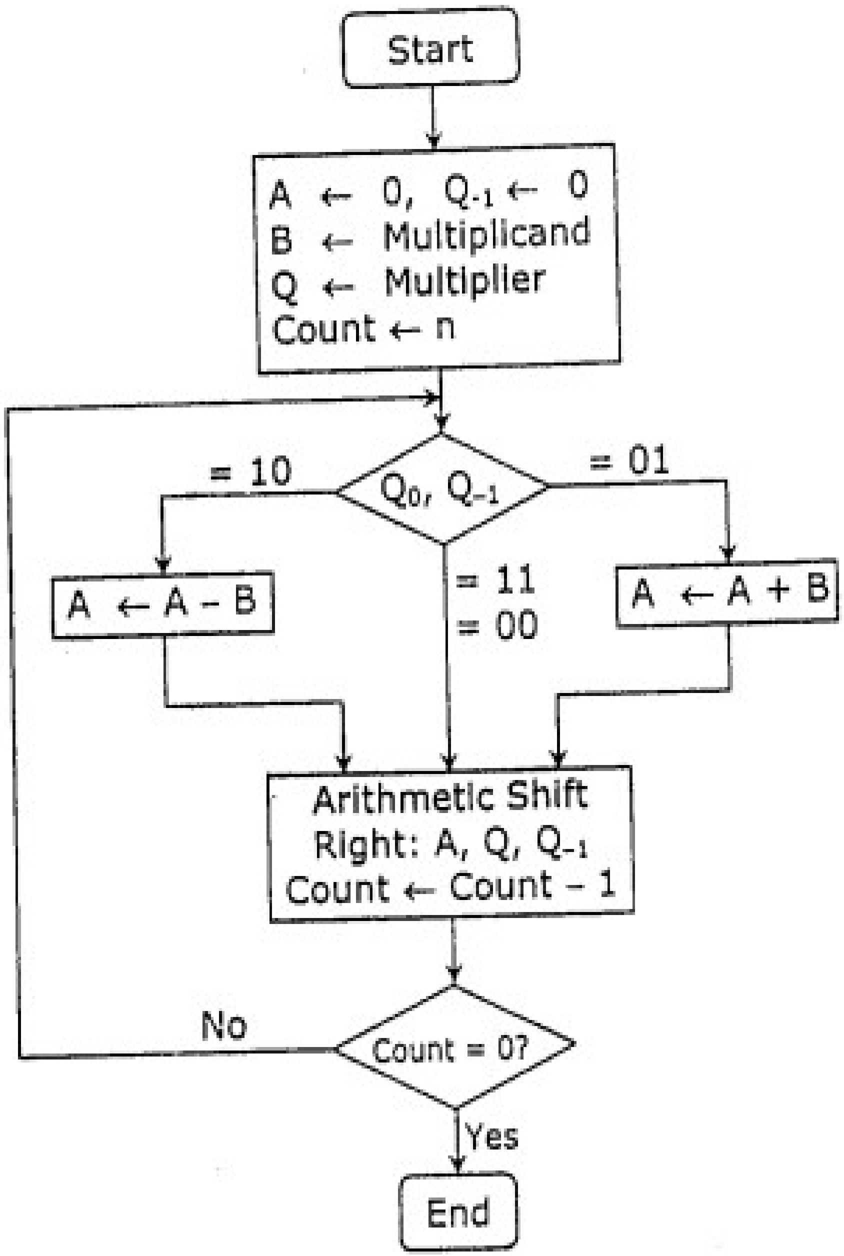
- To understand the working of Booths algorithm.
- To understand how to implement Booth’s algorithm using c-programming.

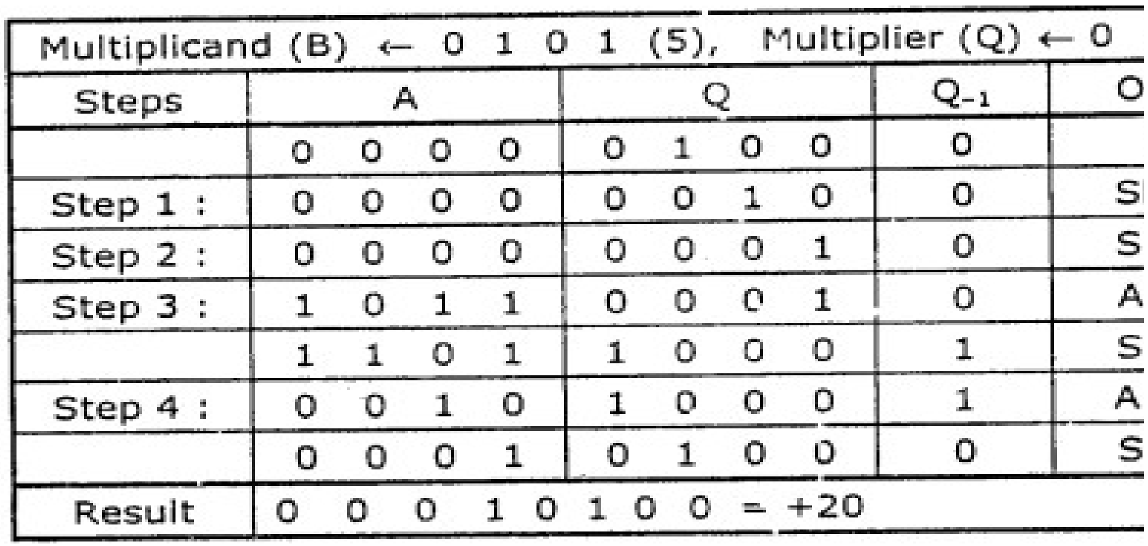
Theory:

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.


Start
A ← 0, Q_{-1} ← 0
B ← Multiplicand
Q ← Multiplier
Count ← n
Q_0, Q_{-1}
= 10
= 01
= 11
= 00
A ← A – B
A ← A + B
Arithmetic Shift Right: A, Q, Q_{-1}
Count ← Count – 1
Count = 0?
No
Yes
End



Multiplicand (B) ← 0 1 0 1 (5), Multiplier (Q) ← 0				
Steps	A	Q	Q-1	O
	0 0 0 0	0 1 0 0	0	
Step 1 :	0 0 0 0	0 0 1 0	0	S
Step 2 :	0 0 0 0	0 0 0 1	0	S
Step 3 :	1 0 1 1	0 0 0 1	0	A
	1 1 0 1	1 0 0 0	1	S
Step 4 :	0 0 1 0	1 0 0 0	1	A
	0 0 0 1	0 1 0 0	0	S
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;
 }
3
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){
4
 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: ");
5
 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){
6
 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:

```
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
7
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:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

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

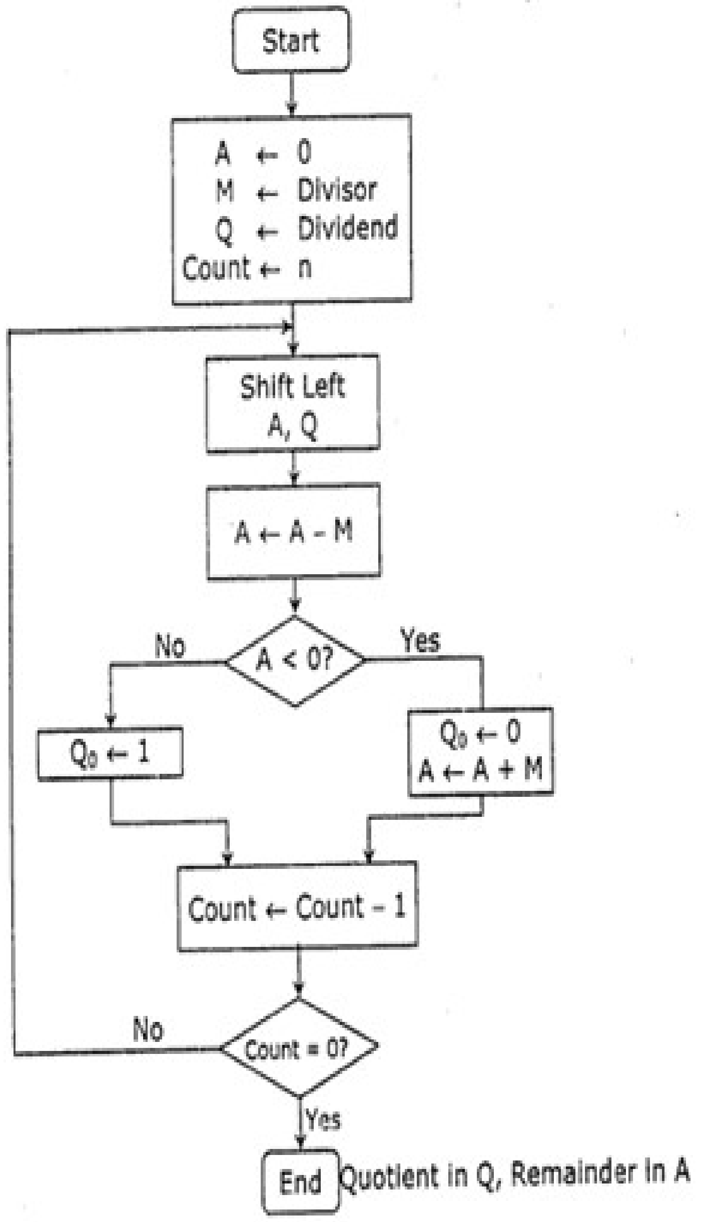
Objective -

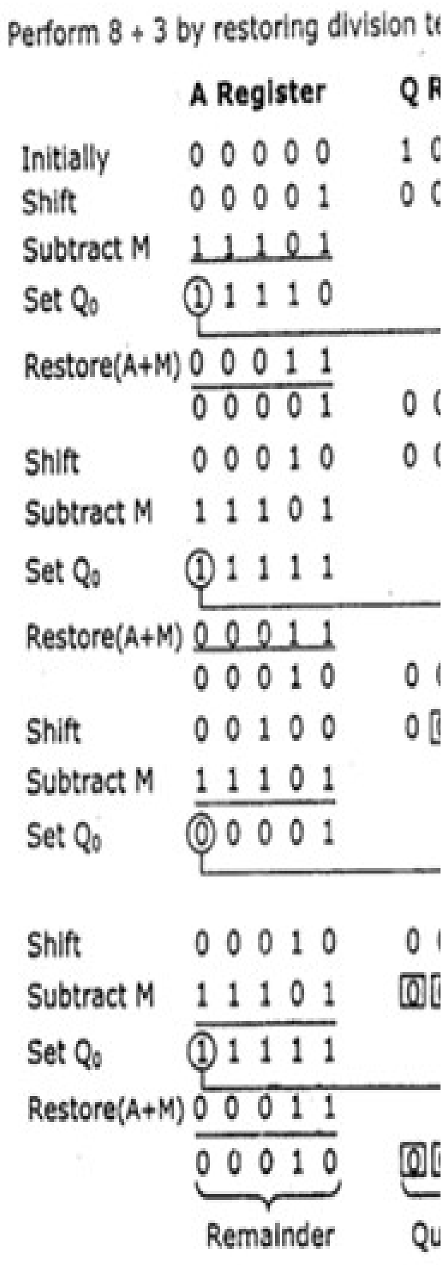
- To understand the working of Restoring division algorithm.
- To understand how to implement Restoring division algorithm using c-programming.

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 t

	A Register	Q R
Initially	0 0 0 0 0	1
Shift	0 0 0 0 1	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
Shift	0 0 0 1 0	0
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
Shift	0 0 1 0 0	0
Subtract M	1 1 1 0 1	
Set Q_0	0 0 0 0 1	
Shift	0 0 0 1 0	0
Subtract M	1 1 1 0 1	0
Set Q_0	1 1 1 1 1	
Restore(A+M)	0 0 0 1 1	
	0 0 0 1 0	0
	Remainder	Qu

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]);
```

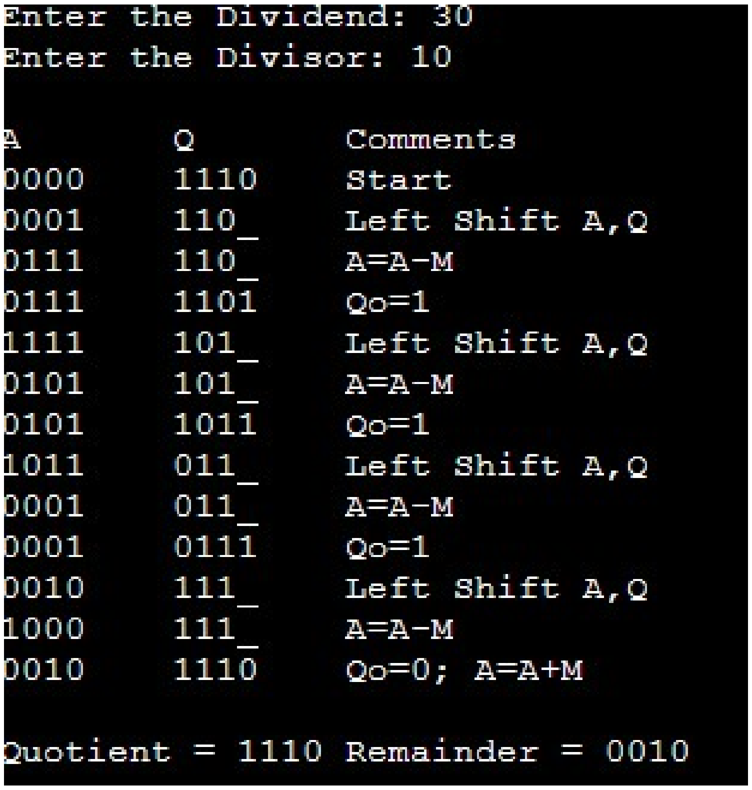
```
}
printf("_\tLeft Shift A,Q\n");
add(acc, m2);
for(i=3; i>=0; i--)
{
   printf("%d", acc[i]);
}
printf("\t");
for(i=3; i>=1; i--)
{
   printf("%d", q[i]);
}
printf("_\tA=A-M\n");
if(acc[3]==0)
{
   q[0]=1;
   for(i=3; i>=0; i--)
   {
      printf("%d", acc[i]);
   }
   printf("\t");
   for(i=3; i>=0; 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: 30
Enter the Divisor: 10

A       Q       Comments
0000    1110    Start
0001    110_    Left Shift A,Q
0111    110_    A=A-M
0111    1101    Qo=1
1111    101_    Left Shift A,Q
0101    101_    A=A-M
0101    1011    Qo=1
1011    011_    Left Shift A,Q
0001    011_    A=A-M
0001    0111    Qo=1
0010    111_    Left Shift A,Q
1000    111_    A=A-M
0010    1110    Qo=0; A=A+M

Quotient = 1110 Remainder = 0010
```

Conclusion -

This experiment involving the Restoring Division Algorithm has provided a comprehensive understanding of this fundamental technique for binary division. The algorithm's step-by-step restoration process allows for precise quotient calculation, making it a valuable tool in computer arithmetic. This experiment has not only reinforced the importance of understanding and implementing division algorithms but has also demonstrated its practical application in various computer systems and data processing tasks

Experiment No. 9
Implement Non-Restoring algorithm using c-programming
Date of Performance:
Date of Submission:

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

Objective -

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

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.

Program -

```
#include <math.h>
#include <stdio.h>
//NON RESTORING DIVISION
int main()
{
```

```
int a[50],a1[50],b[50],d=0,i,j;
 int n1,n2, c, k1,k2,n,k,quo=0,rem=0;
  printf("Enter the number of bits\n");
  scanf("%d",&n);
 printf("Enter the divisor and dividend\n");
 scanf("%d %d", &n1,&n2);

 for (c = n-1; c >= 0; c--)//converting the 2 nos to binary
 {
  k1 = n1 >> c;

  if (k1 & 1)
    a[n-1-c]=1;// M
  else
   a[n-1-c]=0;

   k2 = n2 >> c;

  if (k2 & 1)
    b[2*n-1-c]=1;// Q
  else
   b[2*n-1-c]=0;

 }

 for(i=0;i<n;i++)//making complement
 {
    if(a[i]==0)
      a1[i]=1;
    else
      a1[i]=0;
 }

 a1[n-1]+=1;//twos complement ie -M

 if(a1[n-1]==2)
 {
        for(i=n-1;i>0;i--)
    {
          if(a1[i]==2)
        {
           a1[i-1]+=1;
           a1[i]=0;
        }
    }
 }
 if(a1[0]==2)
  a1[0]=0;

 for( i=0;i<n;i++)// putting A in the same array as Q
 {
    b[i]=0;

 }

printf("A\tQ\tPROCESS\n");

 for(i=0;i<2*n;i++)
{
  if(i==n)
      printf("\t");

  printf("%d",b[i]);
}
 printf("\n");

 for(k=0;k<n;k++)//n iterations
 {
    for(j=0;j<2*n-1;j++)//left shift
      {
        b[j]=b[j+1];

      }

      for(i=0;i<2*n -1;i++)
      {
         if(i==n)
            printf("\t");
         printf("%d",b[i]);
      }printf("_");
```

```
printf("\tLEFT SHIFT\n");

   if(b[0]==0)
   {
            for(i=n-1;i>=0;i--)//A=A-M
            {
               b[i]+=a1[i];

                  if(i!=0)
              {
                  if(b[i]==2)
                        {
                           b[i-1]+=1;
                           b[i]=0;
                        }
                  if(b[i]==3)
                        {
                           b[i-1]+=1;
                           b[i]=1;
                        }
                       // printf("%d",b[i]);
               }
            }
                  if(b[0]==2)
                     b[0]=0;

                  if(b[0]==3)
                     b[0]=1;

            for(i=0;i<2*n -1;i++)
            {
               if(i==n)
                  printf("\t");



               printf("%d",b[i]);
            }printf("_");
```

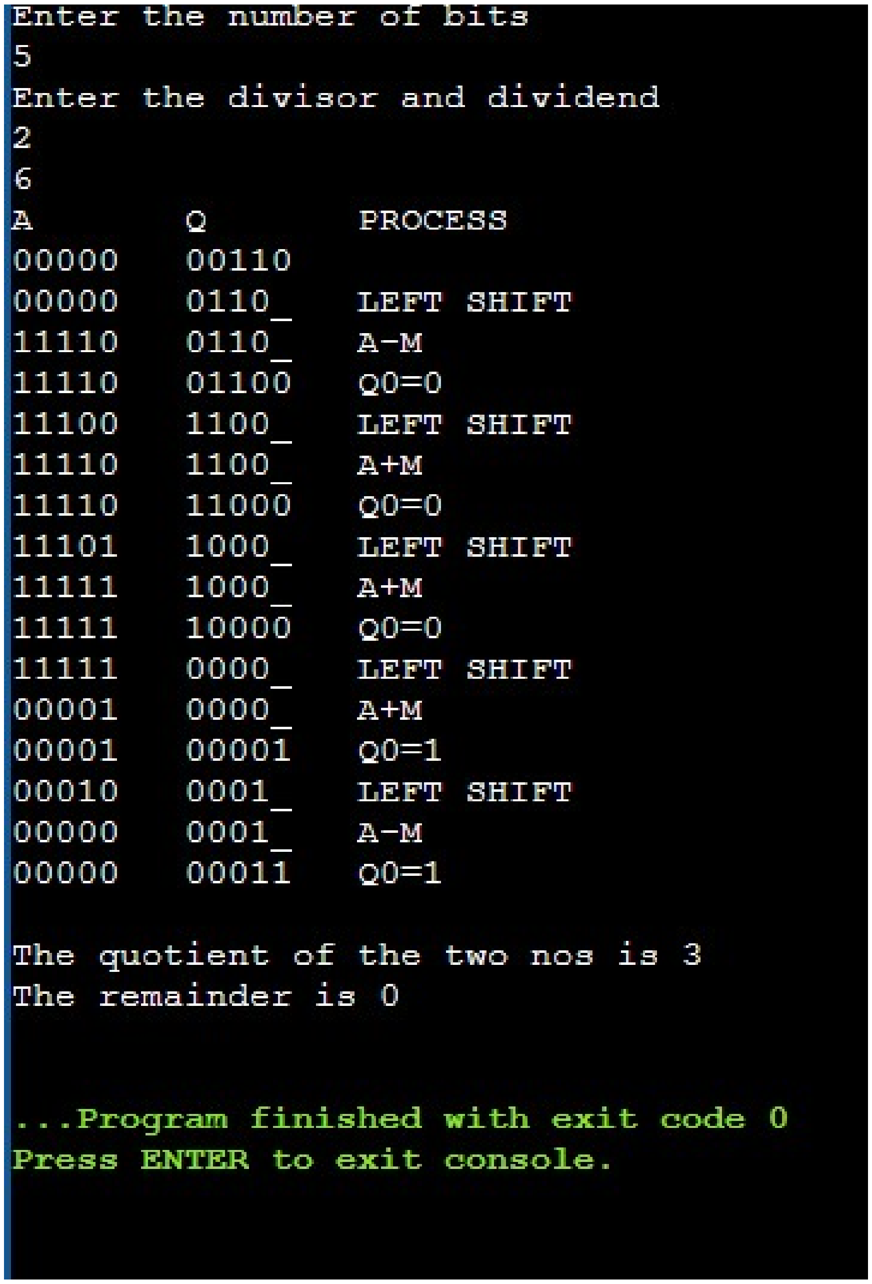
```
          printf("\tA-M\n");
}

else
{
          for(j=n-1;j>=0;j--)//A=A+M
             {
                b[j]+=a[j];

                if(j!=0)
            {
                if(b[j]==2)
                      {
                         b[j-1]+=1;
                         b[j]=0;
                      }
                if(b[j]==3)
                      {
                         b[j-1]+=1;
                         b[j]=1;
                      }
             }

                if(b[0]==2)
                   b[0]=0;

                if(b[0]==3)
                   b[0]=1;
             }

             for(i=0;i<2*n -1;i++)
          {
             if(i==n)
                printf("\t");



             printf("%d",b[i]);
          }printf("_");
```

```
            printf("\tA+M\n");

     }



       if(b[0]==0)//A==0?
       {
          b[2*n-1]=1;
          for(i=0;i<2*n ;i++)
            {
              if(i==n)
                 printf("\t");



              printf("%d",b[i]);
            }

            printf("\tQ0=1\n");
       }



       if(b[0]==1)//A==1?
       {
          b[2*n-1]=0;
          for(i=0;i<2*n ;i++)
            {
              if(i==n)
                 printf("\t");



              printf("%d",b[i]);
            }

            printf("\tQ0=0\n");

       }
```

```
 }

if(b[0]==1)
{
              for(j=n-1;j>=0;j--)//A=A+M
                    {
                       b[j]+=a[j];

                       if(j!=0)
                    {
                       if(b[j]==2)
                             {
                                b[j-1]+=1;
                                b[j]=0;
                             }
                       if(b[j]==3)
                             {
                                b[j-1]+=1;
                                b[j]=1;
                             }
                    }

                       if(b[0]==2)
                          b[0]=0;

                       if(b[0]==3)
                          b[0]=1;
                    }

                    for(i=0;i<2*n;i++)
                 {
                    if(i==n)
                       printf("\t");



                    printf("%d",b[i]);
                 }
```

```
                    printf("\tA+M\n");
}
 printf("\n");
for(i=n;i<2*n;i++)
{
   quo+= b[i]*pow(2,2*n-1-i);
}
for(i=0;i<n;i++)
{
   rem+= b[i]*pow(2,n-1-i);
}
printf("The quotient of the two nos is %d\nThe remainder is %d",quo,rem);

 printf("\n");
  return 0;
}
```

Output:



```
Enter the number of bits
5
Enter the divisor and dividend
2
6
A       Q       PROCESS
00000   00110
00000   0110_   LEFT SHIFT
11110   0110_   A-M
11110   01100   Q0=0
11100   1100_   LEFT SHIFT
11110   1100_   A+M
11110   11000   Q0=0
11101   1000_   LEFT SHIFT
11111   1000_   A+M
11111   10000   Q0=0
11111   0000_   LEFT SHIFT
00001   0000_   A+M
00001   00001   Q0=1
00010   0001_   LEFT SHIFT
00000   0001_   A-M
00000   00011   Q0=1

The quotient of the two nos is 3
The remainder is 0


...Program finished with exit code 0
Press ENTER to exit console.
```

Conclusion -

This experiment and code implementation of the Non-Restoring Division Algorithm have

provided valuable insights into the world of binary division.We have demonstrated the
algorithm's effectiveness in dividing binary numbers without the need for restoring operations, making it suitable for hardware implementations where efficiency is critical.
This experiment has not only showcased the power of algorithmic optimization in digital
computation but has also illustrated the practical application of non-restoring division as
a reliable method for achieving precise binary division in a hardware context.

Experiment No.10
Implement ALU design.
Name:Singh Rahul Rammilan
Roll Number:56
Date of Performance:
Date of Submission:

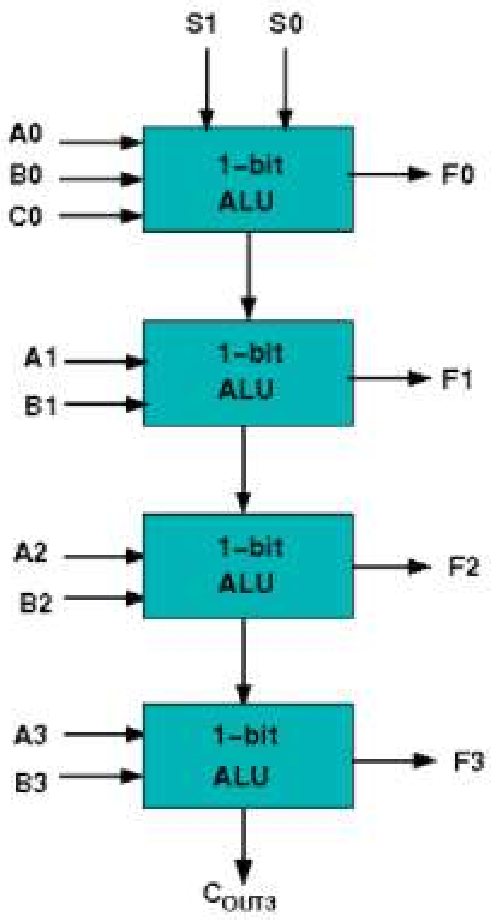
Aim: To implement ALU design

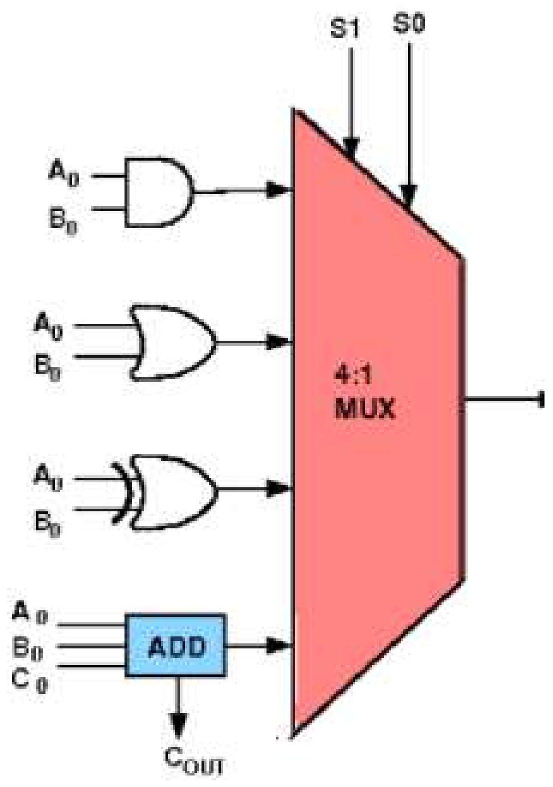
Objective : Objective of 4 bit arithmetic logic unit (with AND, OR, XOR, ADD operation):

- To understand behaviour of arithmetic logic unit from working module.
- 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,

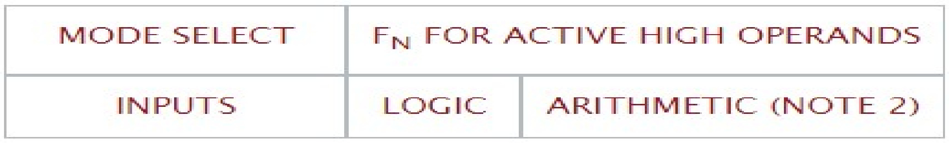
for Control signal S1 = 0 , S0 = 1, the output is A Or B,

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

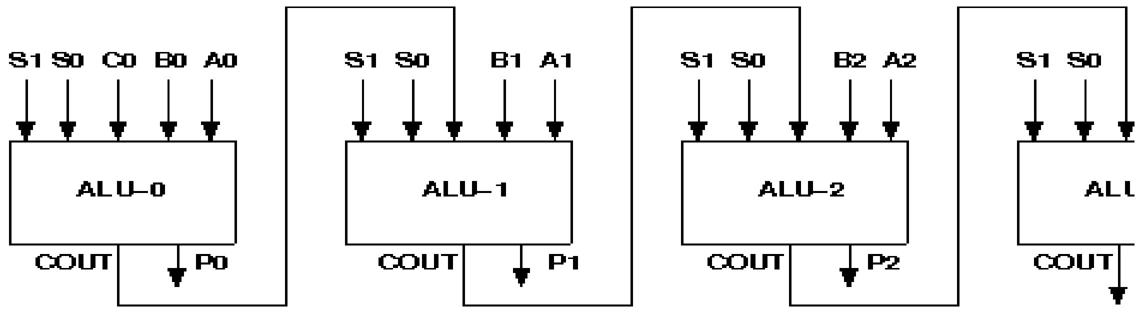
Procedure

- Start the simulator as directed.This simulator supports 5-valued logic.
- 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.
- 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.
- 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
- 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)

- 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.
- 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.

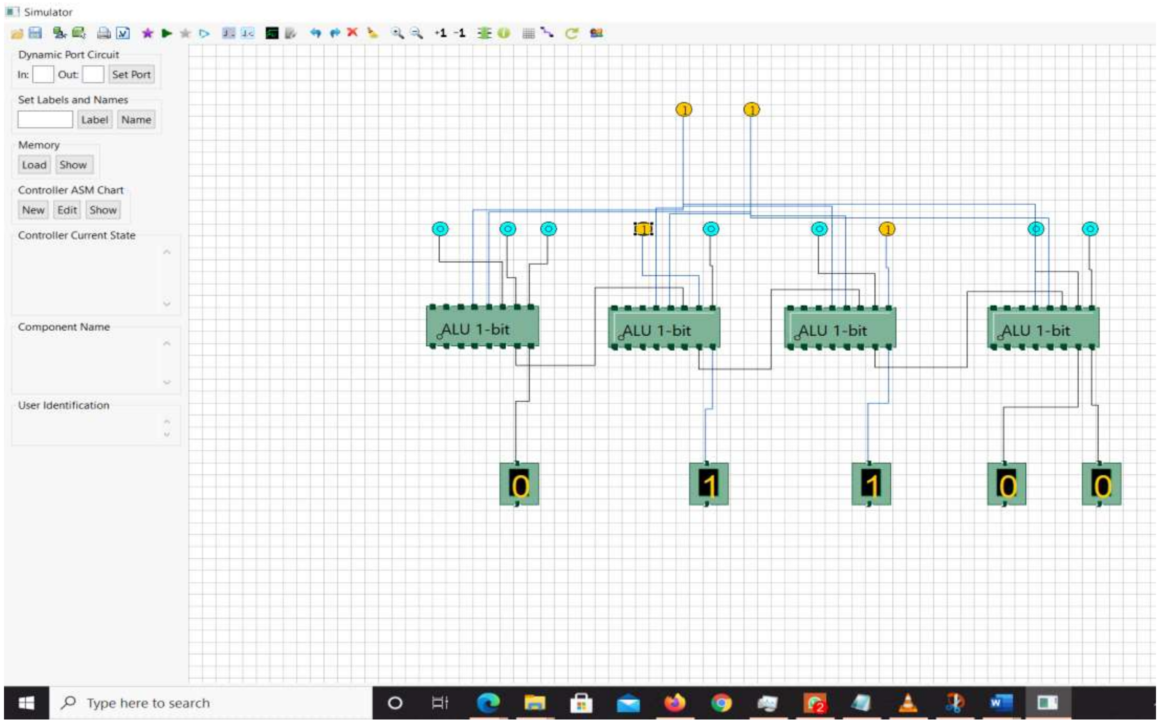
Circuit diagram of 4 bit ALU:



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 experiment conducted on ALU design in Logisim has provided valuable insights into the fundamental aspects of arithmetic and logic unit functionality.
Through rigorous testing and analysis, we have successfully designed and implemented
an ALU that demonstrates efficient operation and accuracy in performing various arithmetic and logical operations. This experiment has not only enhanced our understanding of digital logic design but also highlighted the importance of meticulous
planning and testing in creating reliable computing components. These findings underscore the significance of ALUs in modern computer architecture and their pivotal
role in processing and executing instructions.