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:Ayush Mayekar
Roll Number: 28
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:

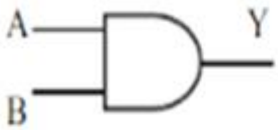
It performs logical addition. Its output become high if any of the inputs is in logic high. 7432 is a Quad 2 input OR gate.

NOT gate:

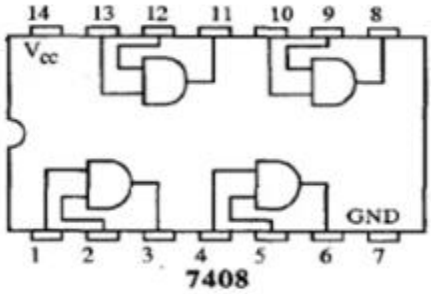
It performs basic logic function for inversion or complementation. The purpose of the inverter is to change one logic level to the opposite level. IC 7404 is a Hex inverter.

Circuit Diagram, Truth Table -

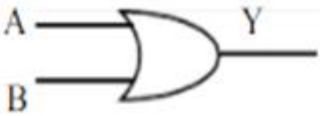
AND Gate -



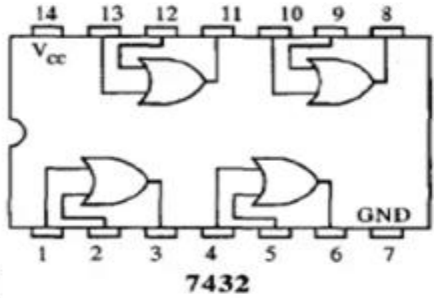
A	B	Y(A.B)
0	0	0
0	1	0
1	0	0
1	1	1



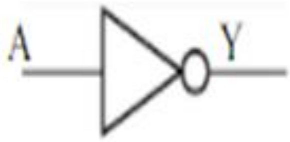
OR Gate -



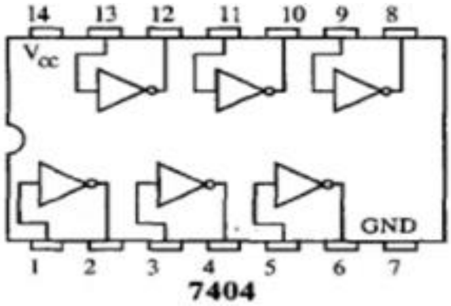
A	B	Y(A+B)
0	0	0
0	1	1
1	0	1
1	1	1



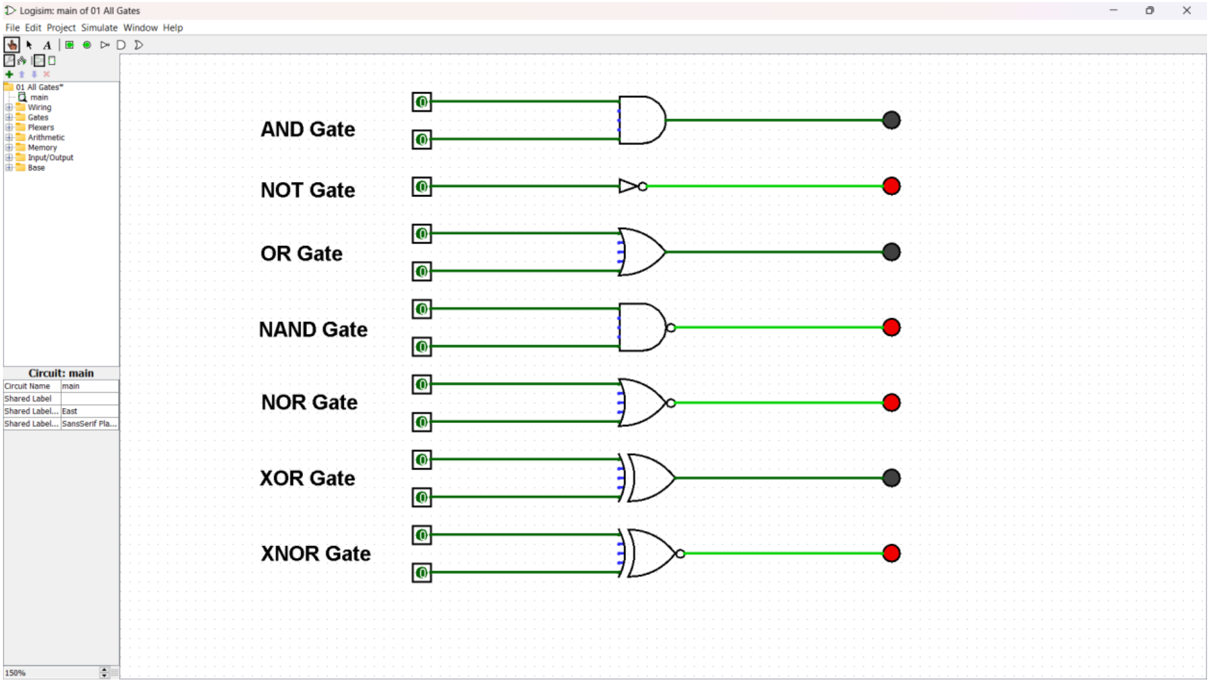
NOT Gate -



A	Y=A'
0	1
1	0



Screenshot:



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.

Conclusion -

In this experiment, our objective was to demonstrate the operation of logic gates in processing binary input signals to generate predefined output states, as defined in their respective truth tables. We accomplished this by harnessing the inherent logical functions of various gate types, such as AND, OR, NOT, NAND, NOR, and XOR gates. This hands-on approach afforded us a tangible comprehension of fundamental digital logic principles and further solidified the connection between logical operations and the resultant outputs. This knowledge is essential for the continued exploration and practical application of digital electronics within the field.

Experiment No. 2
Basic gates using universal gates.
Name: Ayush Mayekar
Roll Number: 28
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.

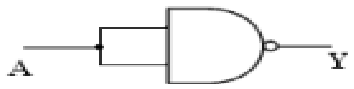
Components required -

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

Circuit Diagram -

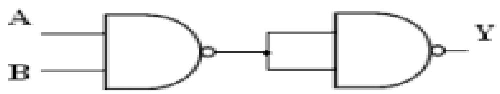
Implementation using NAND gate:

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



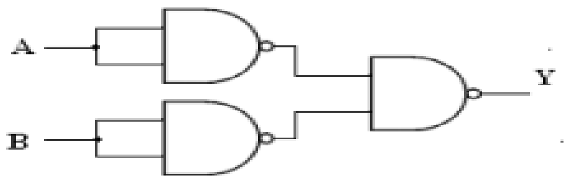
A	Y
0	1
1	0

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



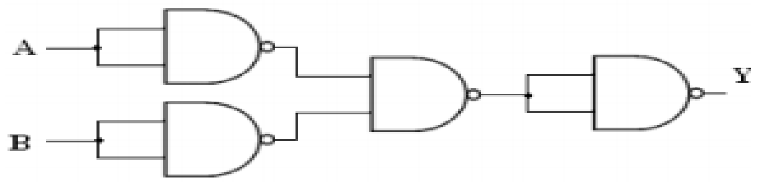
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

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



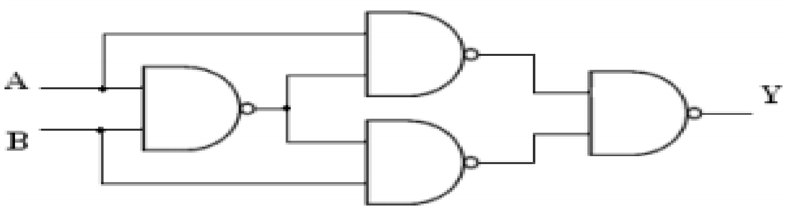
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

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



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

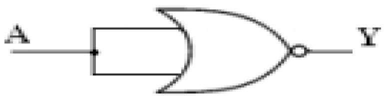
(e) Ex-OR gate: $Y = A \oplus B$



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

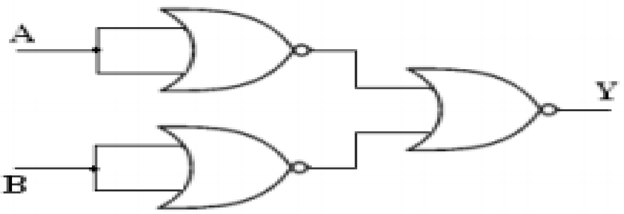
Implementation using NOR gate:

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



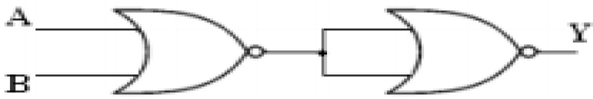
A	Y
0	1
1	0

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



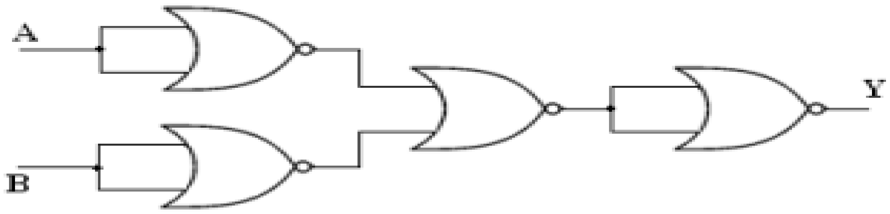
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

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



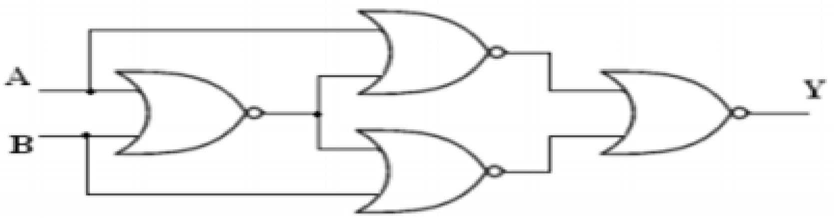
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

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



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

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



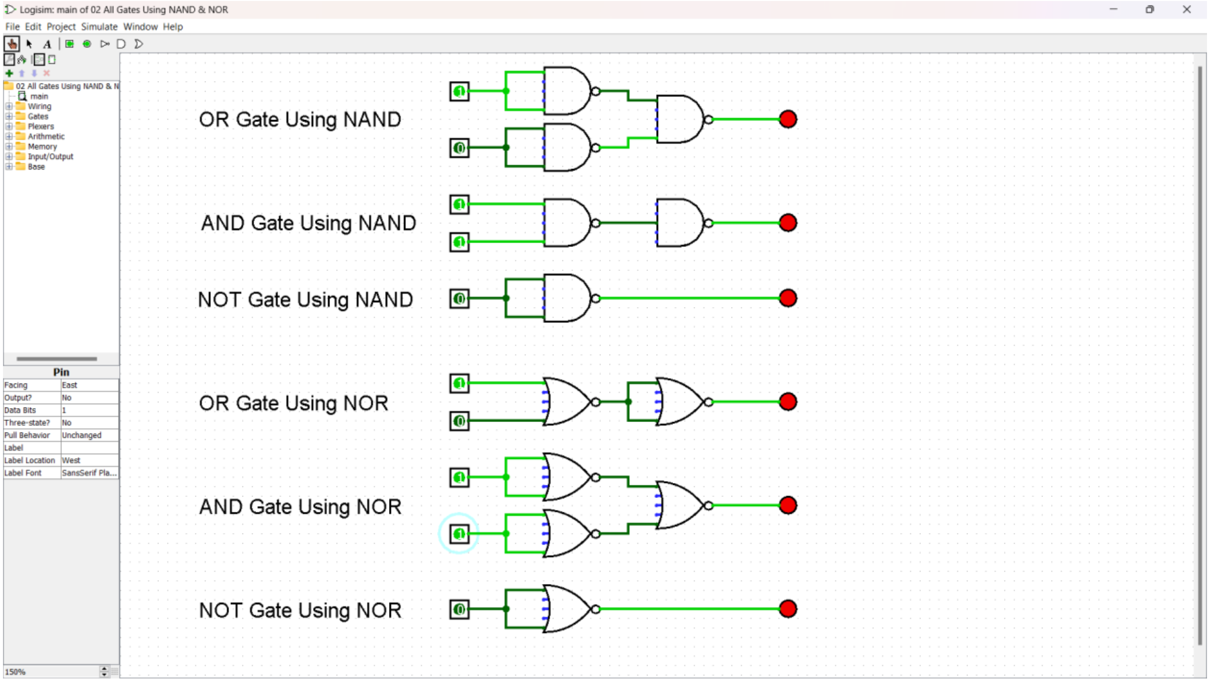
A	B	Y
0	0	1
0	1	0
1	0	0
1	1	1

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.

Screenshot:



Conclusion -

The experiment carried out on universal gates within the Logisim platform has furnished us with invaluable insights into the adaptability and functionality of these fundamental digital logic components. Through this experiment, we've showcased the remarkable capacity of universal gates to execute a diverse spectrum of logical operations, thus emphasizing their pivotal role in contemporary digital circuit design. This investigation serves as a clear testament to the significance of comprehending and harnessing the potential of universal gates within the realm of digital electronics, offering a path to crafting circuitry that is not only more efficient but also highly versatile.

Experiment No. 3
To realize half adder and full adder.
Name: Ayush Mayekar
Roll Number:28
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, Full-adder, 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.

$$\text{Sum} = (A \oplus B) \oplus C_{in}$$
$$\text{Carry} = AB + C_{in}\ (A \oplus B)$$

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

Sum $= A \oplus B$

Carry $= A' B$

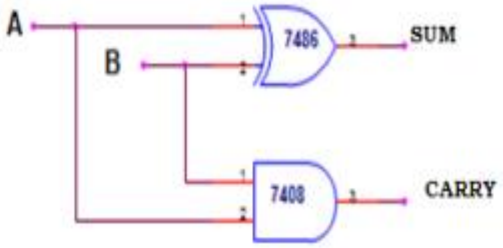
Subtracting two single-bit binary values, B, Cin from a single-bit value A produces a difference bit D and a borrow out Br bit. This is called full subtraction. The Boolean functions describing the full-subtractor are

Difference $= (A \oplus B) \oplus Cin$

Borrow $= A'B + A'(Cin) + B(Cin)$

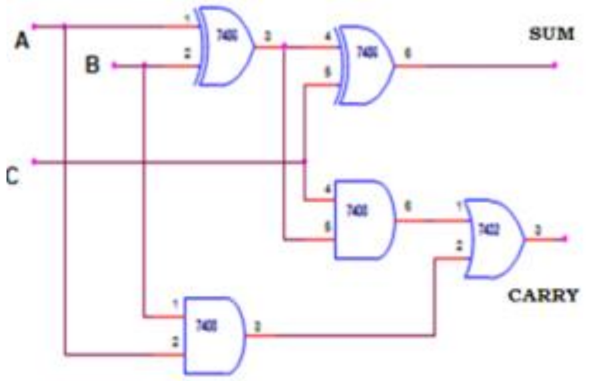
Circuit Diagram and Truth Table -

Half-adder



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

Full-adder

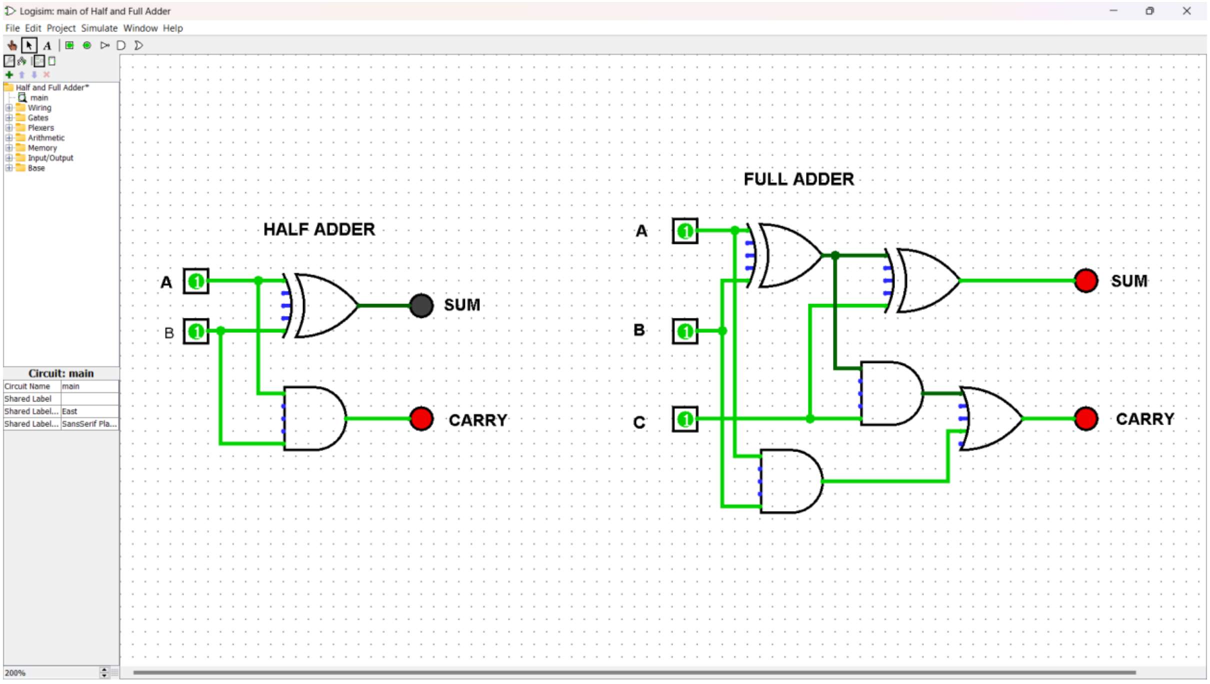


A	B	C	SUM	CARRY
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

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.

Screenshot:



Conclusion -

The experiment conducted with Logisim, focusing on both half adders and full adders, has offered profound insights into the core principles of digital logic design. Our successful demonstration of basic addition operations using half adders expanded our comprehension to include full adders, which are equipped to manage carry inputs. This hands-on experience not only solidified our grasp of binary arithmetic but also emphasized the pivotal role these components play in the construction of intricate digital circuits.

Experiment No. 4
Study of flip flop IC
Name:Ayush Mayekar
Roll Number: 28
Date of Performance:
Date of Submission:

Aim : Study of Flip Flop IC.

Objective : Objective: This experiment aims to understand the functioning of Flip Flop Integrated Circuits (ICs). It involves studying different types of Flip Flops, analyzing signal propagation, examining clock signal effects, observing state transitions, interpreting timing diagrams, comparing Flip Flop types, implementing logic circuits, troubleshooting, and recording/analyzing data.

Theory : Flip Flop ICs (Integrated Circuits) are fundamental building blocks in digital electronics used for storing and manipulating binary information. They are crucial components in digital circuits and are widely used in various applications such as memory units, counters, registers, and more. Flip Flops serve as basic storage elements in digital systems, allowing for the storage and transfer of binary information in the form of 0s and 1s.

There are several types of Flip Flops, each with its unique characteristics and applications. Some common types include D Flip Flop, JK Flip Flop, T Flip Flop, and SR Flip Flop. These Flip Flops can be constructed using various logic gates, such as NAND gates, NOR gates, or a combination of gates.

1. D Flip Flop (Data Flip Flop):

Basic storage element that holds one data bit.

Transfers data to the output on clock signal transition.

Useful for edge-triggered synchronization.

Examples: 74HC74, CD4013.

2. JK Flip Flop:

Combines the features of the SR and D Flip Flops.

Allows toggling of output on certain conditions.

J and K inputs determine the behavior.

Examples: 74HC107, CD4027.

3.T Flip Flop (Toggle Flip Flop):

Toggles its output on each clock signal transition when T input is high.

Useful for frequency division and counters.

Examples: 74HC73, CD4013.

4. SR Flip Flop (Set-Reset Flip Flop):

Has set (S) and reset (R) inputs to control the outputs.

Output depends on the combination of S and R inputs.

Commonly used in asynchronous systems.

Examples: 74HC279, CD4043.

Conclusion :

Flip-flop integrated circuits are essential in digital electronics. They store data, sync operations, and handle complex logic functions. Their dual-state design, triggered by edges, is crucial for a wide range of digital systems. Knowing how flip-flops work empowers engineers to create efficient and dependable circuits and systems.

Experiment No. 5
Implement ripple carry adder
Name: Ayush Mayekar
Roll Number: 28
Date of Performance:
Date of Submission:

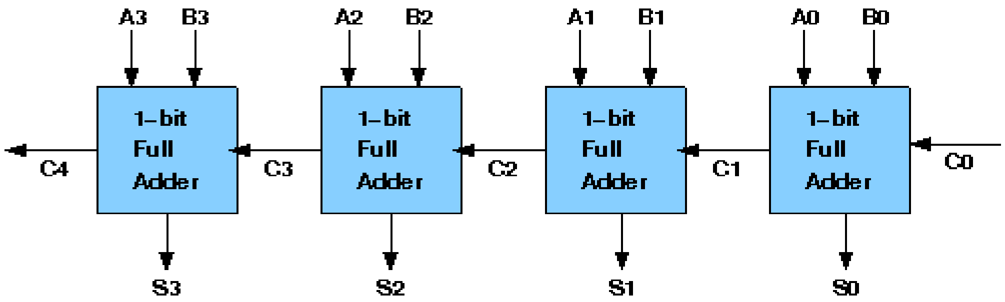
Aim: To implement ripple carry adder.

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 ⊕ 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 ⊕ B) C + (A ⊕ B) C

=A ⊕ B ⊕ C

Carry= ABC + ABC + ABC + ABC

= AB + (AB + AB) C

= AB + (A ⊕ B) C

Procedure:

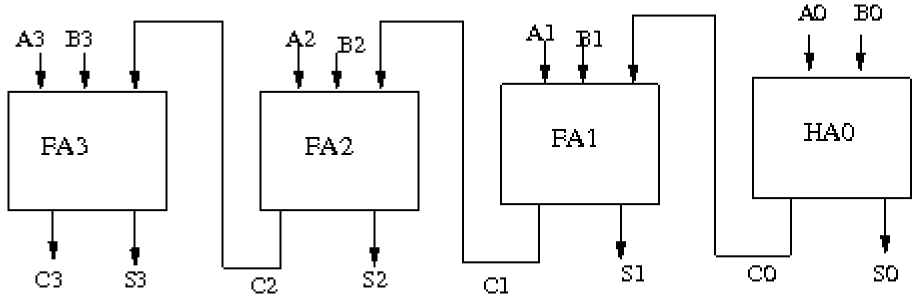
Procedure to perform the experiment: Design of Ripple Carry Adders

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:

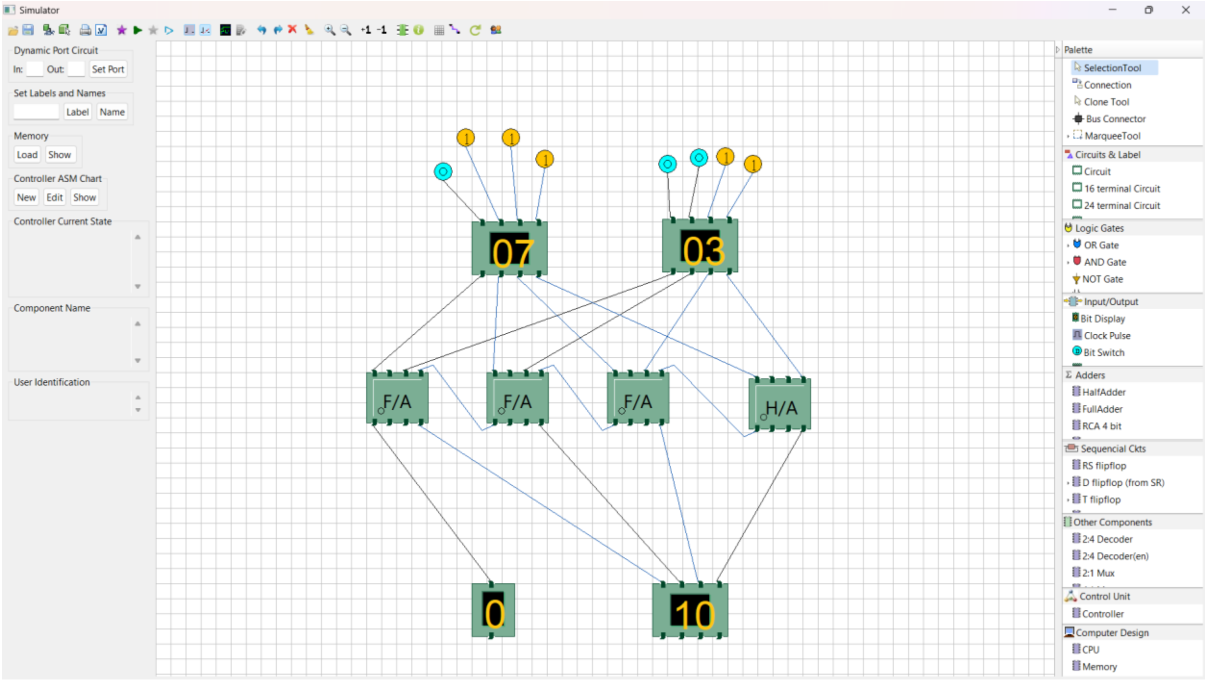
The components needed to create 4 bit ripple carry adder is listed here -

- 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:

Our experiment with the Ripple Carry Adder in Logisim offered valuable insights into the fundamental aspects of binary addition and digital circuitry. We observed how this foundational adder design sequentially carries and propagates bits, which can result in greater latency, especially with larger input values.

Experiment No.6
Implement Carry Look Ahead Adder.
Name: Ayush Mayekar
Roll Number: 28
Date of Performance:
Date of Submission:

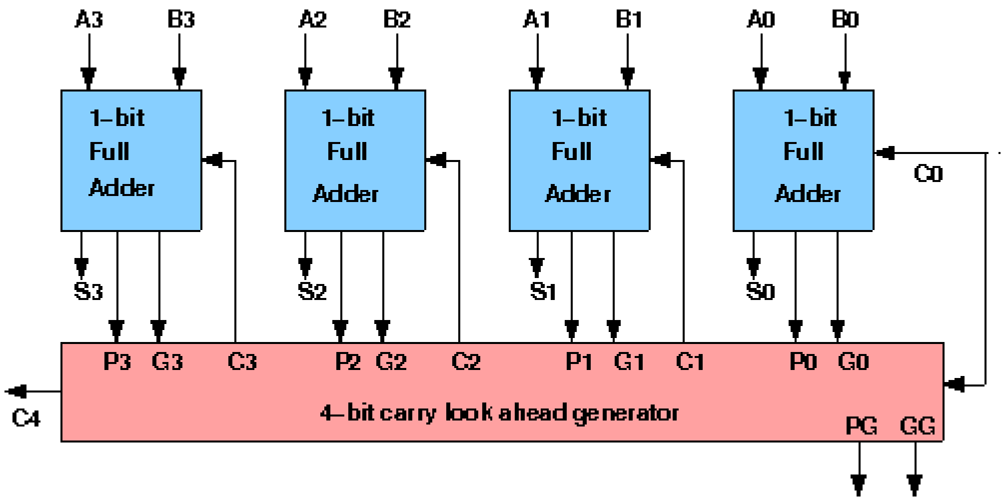
Aim: . To implement carry look ahead adder.

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:

$C1 = G0 + P0 \cdot C0$

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

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

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

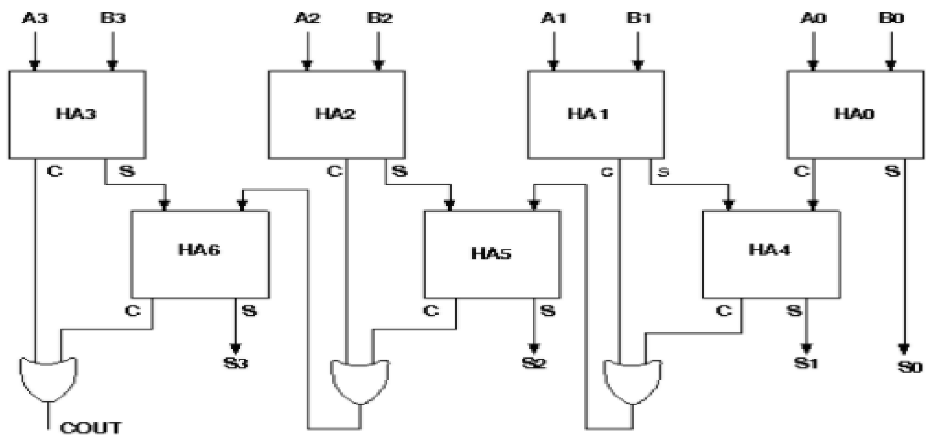
Procedure:

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:

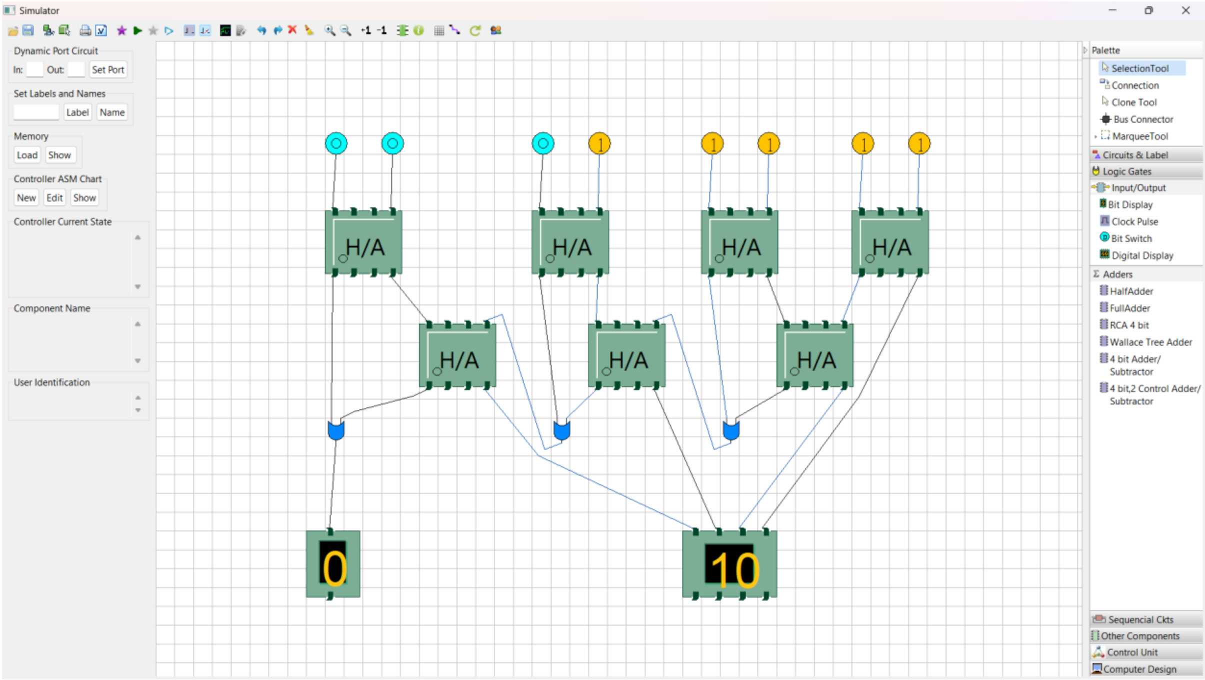


Components required:

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:

The experiment carried out with the Carry Look-Ahead Adder in Logisim has furnished us with significant insights into the efficiency and operation of this advanced digital circuit. We've illustrated the remarkable speed at which the Carry Look-Ahead Adder can perform binary number addition, thanks to its ability to minimize carry propagation delays and reduce computation time. This attribute underscores its pivotal role in contemporary computer architecture and arithmetic circuits. Beyond that, this experiment reaffirms the significance of streamlined adder designs and demonstrates how complex digital circuits can be practically implemented within the realm of digital logic simulation.

Experiment No. 7
Implement Booth's algorithm using c-programming
Name: Ayush Mayekar
Roll Number: 28
Date of Performance:
Date of Submission:

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

Objective -

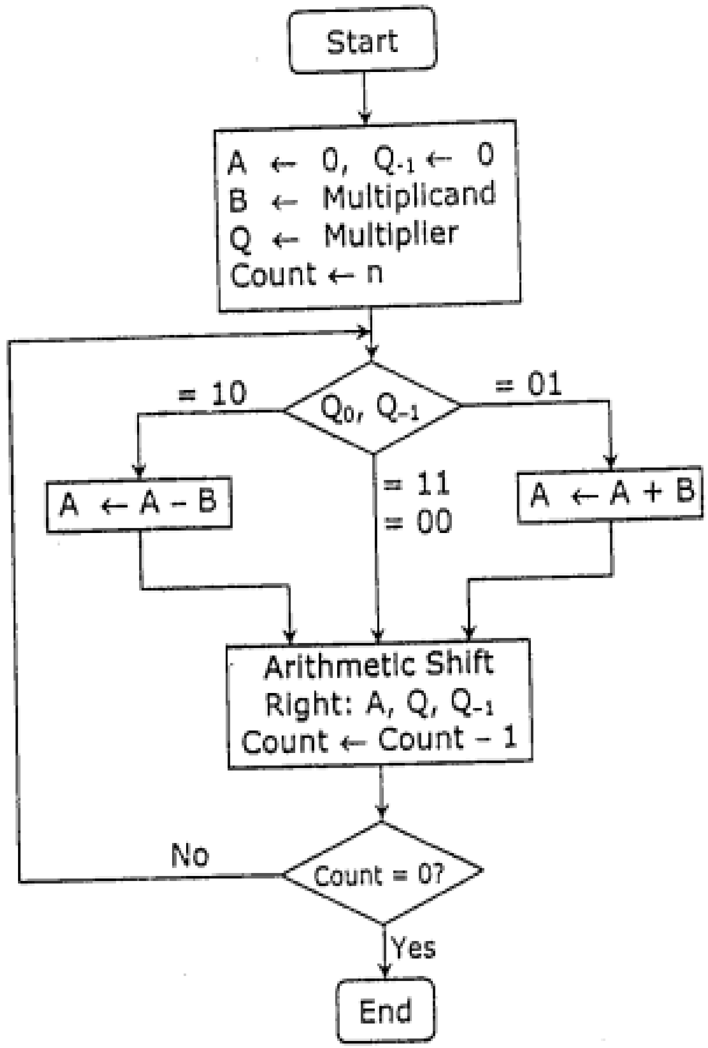
1. To understand the working of Booths algorithm.
2. To understand how to implement Booth's algorithm using c-programming.

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 <stdio.h>

#include <math.h>

 int a = 0,b = 0, c = 0, a1 = 0, b1 = 0, com[5] = { 1, 0, 0, 0, 0};

int anum[5] = {0}, anumcp[5] = {0}, bnum[5] = {0};

int acomp[5] = {0}, bcomp[5] = {0}, pro[5] = {0}, res[5] = {0};


void binary(){

    a1 = fabs(a);

    b1 = fabs(b);

    int r, r2, i, temp;

    for (i = 0; i < 5; i++){

        r = a1 % 2;

        a1 = a1 / 2;

        r2 = b1 % 2;
```

```
        b1 = b1 / 2;
        anum[i] = r;
        anumcp[i] = r;
        bnum[i] = r2;
        if(r2 == 0){
            bcomp[i] = 1;
        }
        if(r == 0){
            acomp[i] =1;
        }
    }
    //part for two's complementing
    c = 0;
    for ( i = 0; i < 5; i++){
        res[i] = com[i]+ bcomp[i] + c;
        if(res[i] >= 2){
            c = 1;
        }
        else
            c = 0;
        res[i] = res[i] % 2;
    }
    for (i = 4; i >= 0; i--){
```

```
  bcomp[i] = res[i];
 }
 //in case of negative inputs
 if (a < 0){
   c = 0;
  for (i = 4; i >= 0; i--){
        res[i] = 0;
  }
  for ( i = 0; i < 5; i++){
        res[i] = com[i] + acomp[i] + c;
        if (res[i] >= 2){
             c = 1;
        }
        else
             c = 0;
        res[i] = res[i]%2;
  }
  for (i = 4; i >= 0; i--){
        anum[i] = res[i];
        anumcp[i] = res[i];
  }


 }
```

```
  if(b < 0){
   for (i = 0; i < 5; i++){
      temp = bnum[i];
      bnum[i] = bcomp[i];
      bcomp[i] = temp;
   }
  }
}
void add(int num[]){
  int i;
  c = 0;
  for ( i = 0; i < 5; i++){
      res[i] = pro[i] + num[i] + c;
      if (res[i] >= 2){
         c = 1;
      }
      else{
         c = 0;
      }
      res[i] = res[i]%2;
   }
   for (i = 4; i >= 0; i--){
     pro[i] = res[i];
```

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

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

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

    printf("\nExpected product = %d", a * b);
    binary();
    printf("\n\nBinary Equivalents are: ");
    printf("\nA = ");
    for (i = 4; i >= 0; i--){
        printf("%d", anum[i]);
```

```
}
printf("\nB = ");
for (i = 4; i >= 0; i--){
   printf("%d", bnum[i]);
}
printf("\nB'+ 1 = ");
for (i = 4; i >= 0; i--){
   printf("%d", bcomp[i]);
}
printf("\n\n");
for (i = 0;i < 5; i++){
       if (anum[i] == q){//just shift for 00 or 11
           printf("\n-->");
           arshift();
           q = anum[i];
       }
       else if(anum[i] == 1 && q == 0){//subtract and shift for 10
          printf("\n-->");
          printf("\nSUB B: ");
          add(bcomp);//add two's complement to implement subtraction
          arshift();
          q = anum[i];
       }
```

```
        else{//add ans shift for 01
          printf("\n-->");
          printf("\nADD B: ");
          add(bnum);
          arshift();
          q = anum[i];
        }
    }

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

Output:

Terminal

```
BOOTH'S MULTIPLICATION ALGORITHM
Enter two numbers to multiply:
Both must be less than 16
Enter A: 10
Enter B: 05
Expected product = 50

Binary Equivalents are:
A = 01010
B = 00101
B'+ 1 = 11011



-->
AR-SHIFT: 00000:00101
-->
SUB B: 11011:00101
AR-SHIFT: 11101:10010
-->
ADD B: 00010:10010
AR-SHIFT: 00001:01001
-->
SUB B: 11100:01001
AR-SHIFT: 11110:00100
-->
ADD B: 00011:00100
AR-SHIFT: 00001:10010
Product is = 0000110010
```

Conclusion –

Our experiment on Booth's algorithm has underscored its crucial role in optimizing binary multiplication. This efficient algorithm effectively reduces the number of partial products and minimizes the overall multiplication operations, thereby enhancing computational speed and reducing hardware complexity. Booth's algorithm stands as a powerful tool for optimizing multiplication processes and is a fundamental concept in digital arithmetic. Our experiment has effectively showcased its practical applicability in computer architecture and digital circuit design.

Experiment No. 8
Implement Restoring algorithm using c-programming
Name: Ayush Mayekar
Roll Number: 28
Date of Performance:
Date of Submission:

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

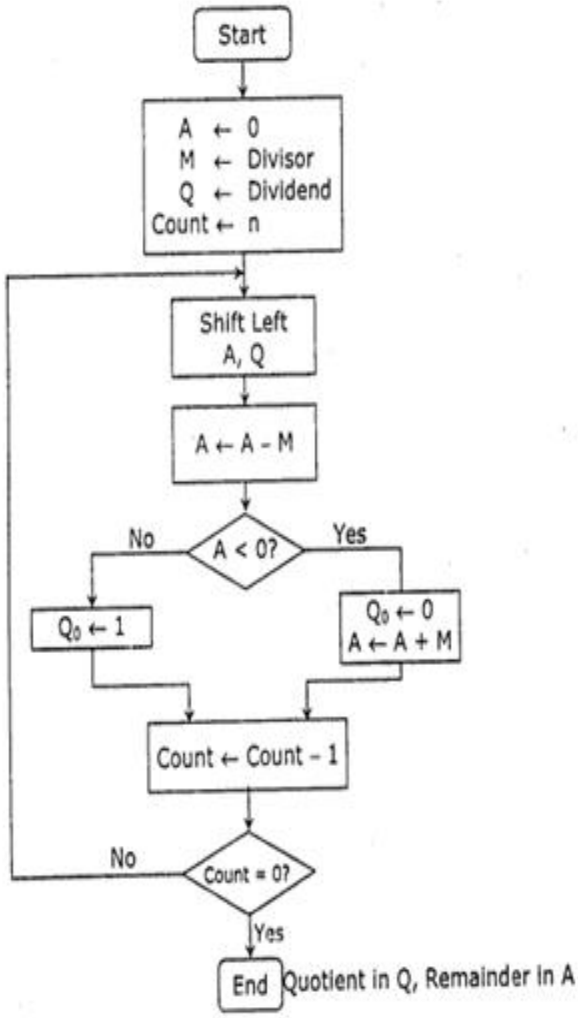
Objective -

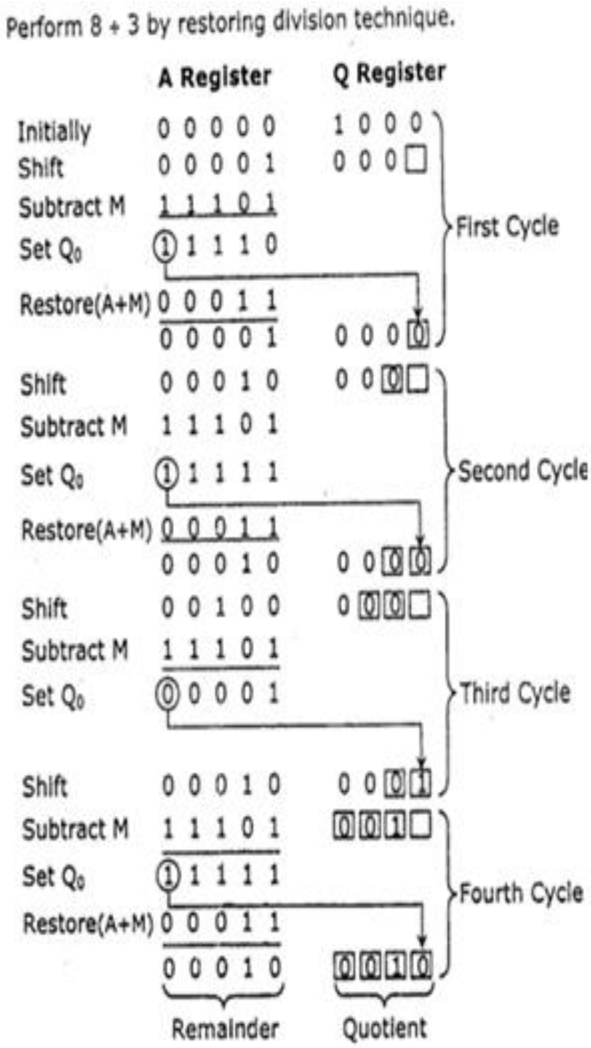
1. To understand the working of Restoring division algorithm.
2. To understand how to implement Restoring division algorithm using c-programming.

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





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 -

Terminal

```
Enter the Dividend: 15
Enter the Divisor: 5
A    Q    Comments
0000    1111    Start
0001    111_    Left Shift A,Q
1100    111_    A=A-M
0001    1110    Qo=0; A=A+M
0011    110_    Left Shift A,Q
1110    110_    A=A-M
0011    1100    Qo=0; A=A+M
0111    100_    Left Shift A,Q
0010    100_    A=A-M
0010    1001    Qo=1
0101    001_    Left Shift A,Q
0000    001_    A=A-M
0000    0011    Qo=1

Quotient = 0011 Remainder = 0000
```

Conclusion –

Our experiment on the Restoring Division Algorithm has yielded a deep and thorough comprehension of this essential binary division technique. The algorithm's meticulous step-by-step restoration approach enables precise quotient calculation, rendering it an invaluable asset in computer arithmetic. Through this experiment, we've not only underscored the significance of grasping and utilizing division algorithms but also effectively showcased their real-world applications in diverse computer systems and data processing tasks.

Experiment No. 9
Implement Non-Restoring algorithm using c-programming
Name: Ayush Mayekar
Roll Number: 28
Date of Performance:
Date of Submission:

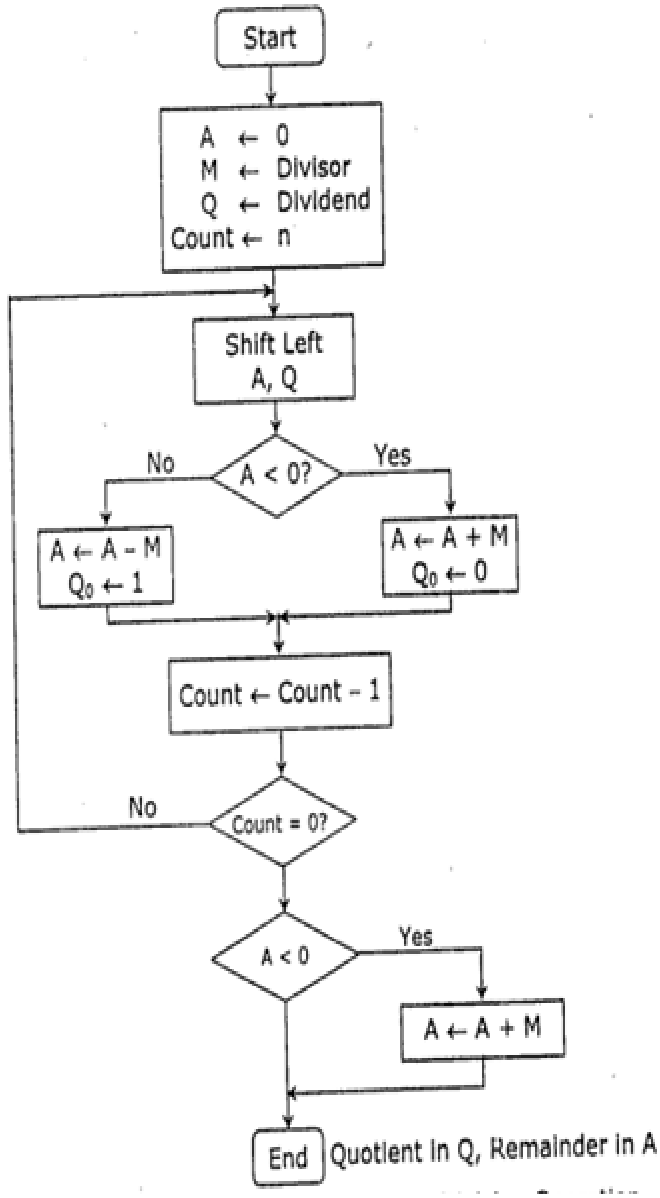
Aim - To implement Non-Restoring division algorithm using c-programming.

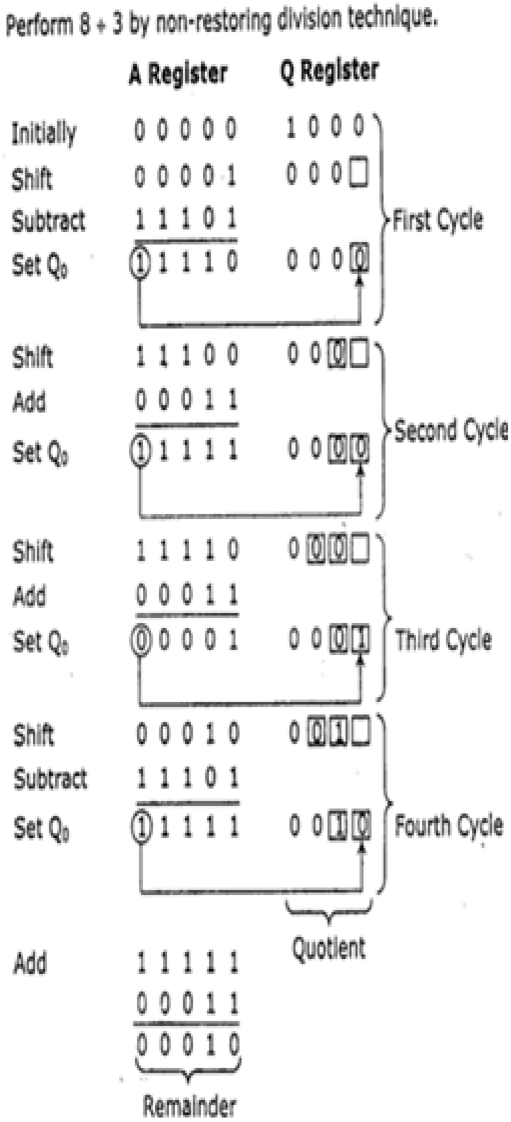
Objective -

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


Start
A ← 0
M ← Divisor
Q ← Dividend
Count ← n
Shift Left A, Q
A < 0?
No
Yes
A ← A − M
Q0 ← 1
A ← A + M
Q0 ← 0
Count ← Count − 1
Count = 0?
No
A < 0
Yes
A ← A + M
End
Quotient in Q, Remainder in A


Perform 8 ÷ 3 by non-restoring division technique.
A Register
Q Register
Initially 0 0 0 0 0 1 0 0 0
Shift 0 0 0 0 1 0 0 0
Subtract 1 1 1 0 1
Set Q0 1 1 1 1 0 0 0 0 0
First Cycle
Shift 1 1 1 0 0 0 0 0
Add 0 0 0 1 1
Set Q0 1 1 1 1 1 0 0 0 0
Second Cycle
Shift 1 1 1 1 0 0 0 0
Add 0 0 0 1 1
Set Q0 0 0 0 0 1 0 0 0 1
Third Cycle
Shift 0 0 0 1 0 0 0 1
Subtract 1 1 1 0 1
Set Q0 1 1 1 1 1 0 0 1 0
Fourth Cycle
Quotient
Add 1 1 1 1 1
0 0 0 1 1
0 0 0 1 0
Remainder

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:

Terminal

```
Enter the number of bits
4
Enter the divisor and dividend
1010
0010
A     Q     PROCESS
0000  1010
0001  010_  LEFT SHIFT
1111  010_  A-M
1111  0100  Q0=0
1110  100_  LEFT SHIFT
0000  100_  A+M
0000  1001  Q0=1
0001  001_  LEFT SHIFT
1111  001_  A-M
1111  0010  Q0=0
1110  010_  LEFT SHIFT
0000  010_  A+M
0000  0101  Q0=1

The quotient of the two nos is 5
The remainder is 0
```

Conclusion -

Our experiment and code implementation of the Non-Restoring Division Algorithm have furnished us with significant insights into the realm of binary division. We've effectively demonstrated the algorithm's prowess in dividing binary numbers without necessitating restoring operations, rendering it particularly apt for hardware implementations where efficiency holds paramount importance. This experiment not only highlights the potential of algorithmic optimization in digital computation but also serves as a practical illustration of non-restoring division's reliability in achieving precise binary division within a hardware context.

Experiment No.10
Implement ALU design.
Name: Ayush Mayekar
Roll Number: 28
Date of Performance:
Date of Submission:

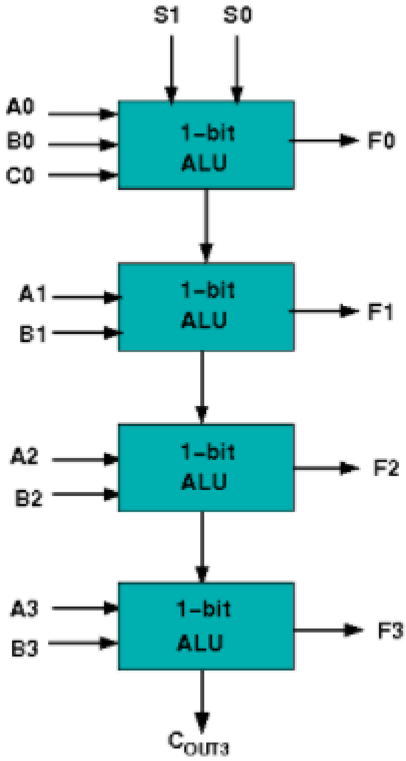
Aim: To implement ALU design

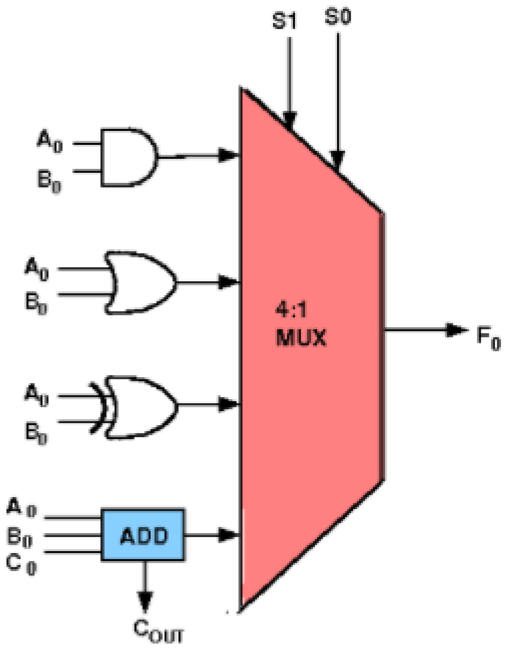
Objective : Objective of 4 bit arithmetic logic unit (with AND, OR, XOR, ADD operation):

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)

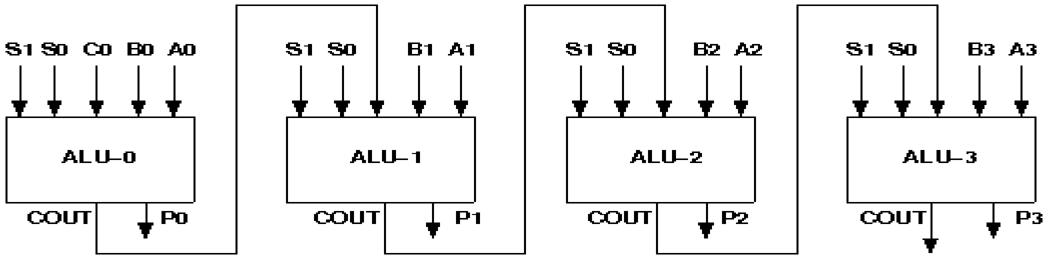
S3	S2	S1	S0	(M = H)	(M = L) (Cn=L)
L	L	L	L	A'	A
L	L	L	H	A'+B'	A+B
L	L	H	L	A'B	A+B'
L	L	H	H	Logic 0	minus 1
L	H	L	L	(AB)'	A plus AB'
L	H	L	H	B'	(A + B) plus AB'

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:

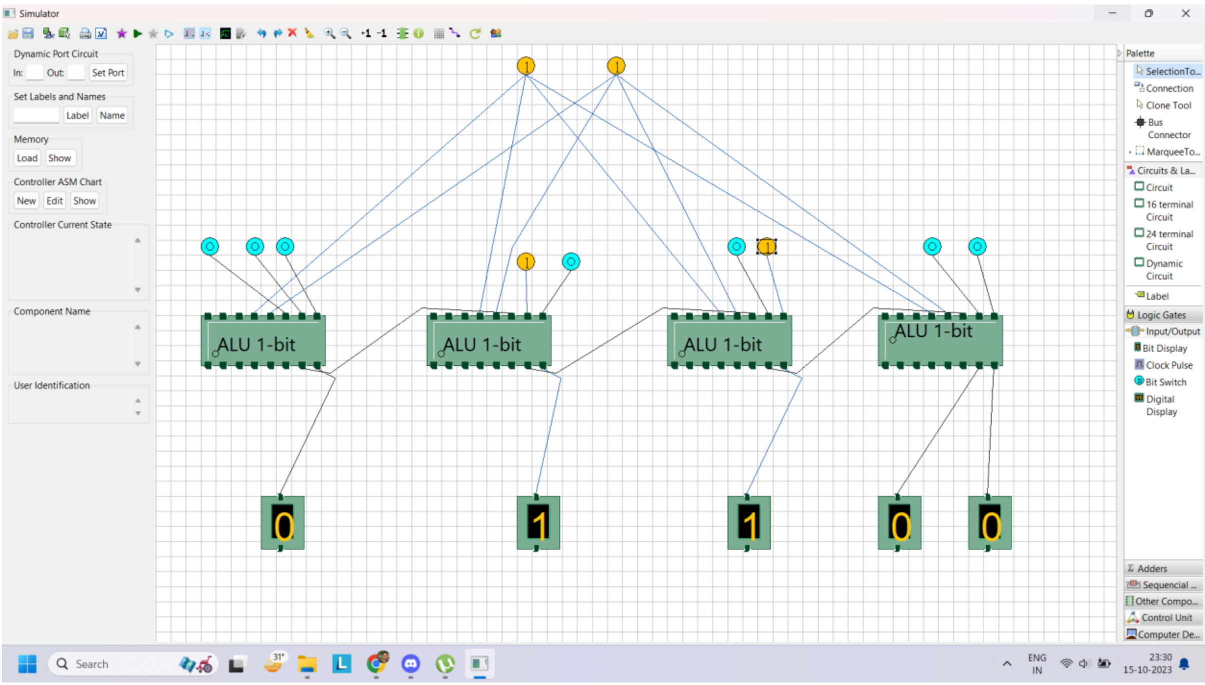


Components required :

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:

The experiment aimed to design a 4-bit ALU with AND, OR, XOR, and ADD operations. Understanding the behavior and configuration of the ALU is essential for digital circuit design. The experiment's success demonstrates the practicality of ALUs in computer systems and digital logic circuits.