Experiment No. 1
Truth table of various logic gates using ICs.
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 debugit.
2. The principal objective of this experiment is to fully understand the function and useof logic gates.
3. Understand how to implement simple circuits based on a schematic diagram usinglogic 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 aQuad 2 input AND gate.

- **OR gate:**

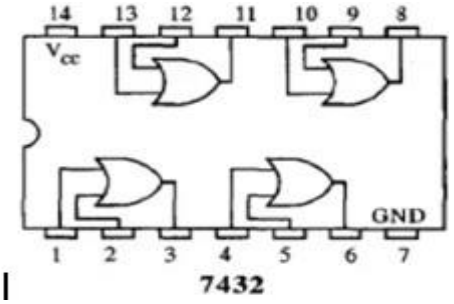
It performs logical addition. Its output become high if any of the inputs is in logic high. 7432 is a Quad 2 input OR gate.

- **NOT gate:**

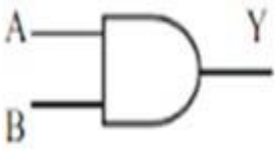
It performs basic logic functions for inversion or complementation. The purpose of theinverter is to change one logic level to the opposite level. IC 7404 is a Hex inverter.

Circuit Diagram, Truth Table -AND Gate -

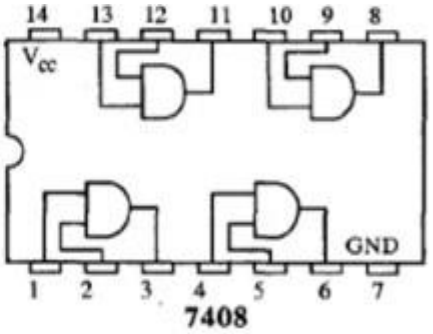
A	B	Y(A+B)
0	0	0
0	1	1
1	0	1
1	1	1



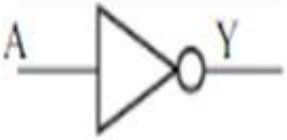
OR Gate -



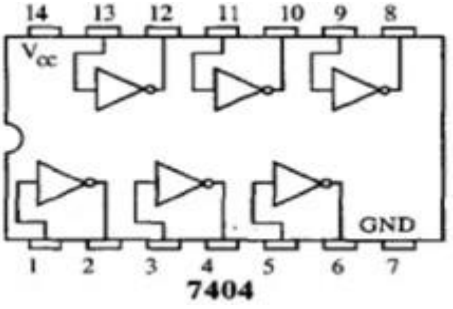
A	B	Y(A.B)
0	0	0
0	1	0
1	0	0
1	1	1



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.

Conclusion -

The experiment to verify the truth table of various logic gates using integrated circuits (ICs) has proven to be a valuable and insightful exercise in understanding digital logic. Through systematic observation and analysis, we have gained a practical understanding of how different logic gates operate and how their behaviors align with theoretical expectations.

The use of ICs has not only provided a convenient and efficient means of implementing logic gates but has also demonstrated the reliability and consistency of these electronic components in performing logical operations. The experiment has reinforced the importance of accurate documentation, careful circuit design, and meticulous observation in the field of digital electronics

Experiment No. 2
Basic gates using universal gates.
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 orNOR 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 logicfunction can be implemented.

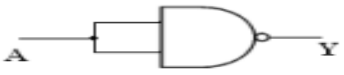
Components required -

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

Circuit Diagram -

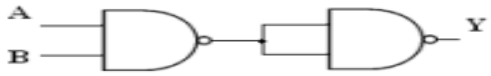
Implementation using NAND gate:

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



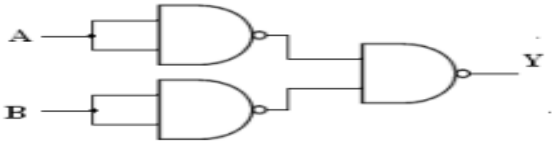
A	Y
0	1
1	0

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



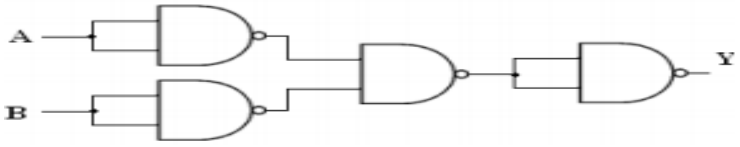
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

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



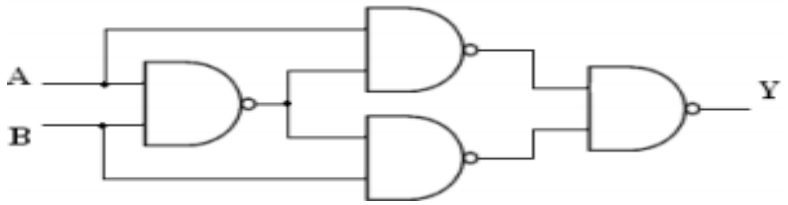
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(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: $Y = A \oplus B$



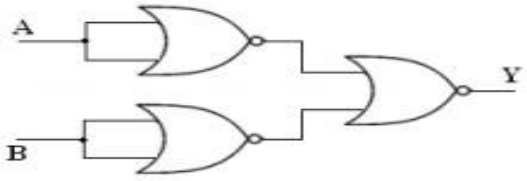
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Implementation using NOR gate:

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

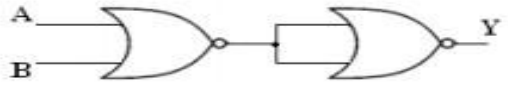
A	Y
0	1
1	0

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



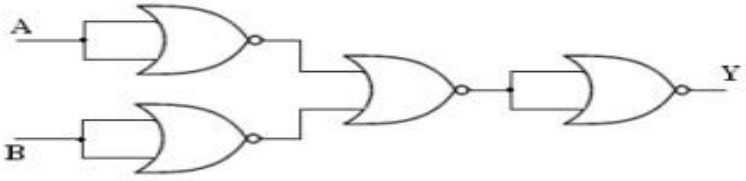
A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1

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



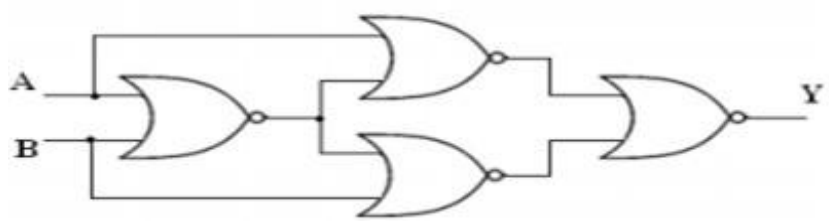
A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

(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 thehelp of truth table.

Conclusion –

The realization of different logic gates using NAND (NOT-AND) and NOR (NOT-OR) gates underscores the fundamental and versatile nature of these two universal gates in digital circuitry. By employing combinations and variations of NAND and NOR gates, it is possible to construct a wide array of logic functions, including AND, OR, NOT, XOR, and more. This not only simplifies the design and implementation of digital circuits but also contributes to cost-effectiveness and efficiency in terms of manufacturing and maintenance.

The use of NAND and NOR gates as building blocks for other logic gates is not only practical but also strategically advantageous. It highlights the elegance of these gates in terms of simplicity and functionality. Furthermore, their ability to serve as universal gates emphasizes their pivotal role in digital electronics.

Experiment No. 3
Implement given Boolean function using logic gates in SOP and POS forms.
Date of Performance:
Date of Submission:

Aim - To implement the given Boolean function using logic gates in SOP and POS forms.

Two input SOP - A.B + A'.B'

Two input POS: - (A+B) (B+C) (A+C')

Objective:

1. Understand how to use the breadboard to patch up, test your logic design and debugit.
2. The principal objective of this experiment is to fully understand the function and useof logic gates
3. Understand how to implement simple circuits based on a schematic diagram usinglogic gates.

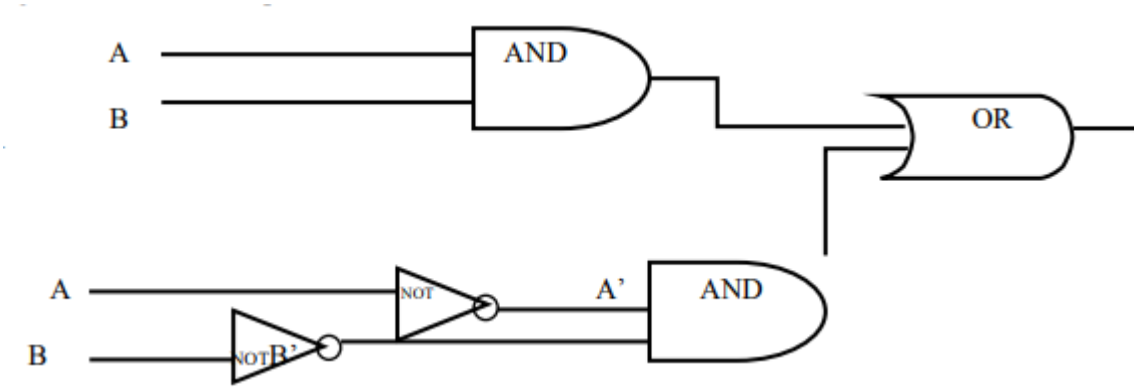
Components required -

1. IC's 7404. 7408, 7432
2. Bread Board.
3. Connecting wires.
4.

Theory-

SOP: - It is the Sum of product form in which the terms are taken as 1. It is denoted in theK-map expression by sigma (∑)

Circuit Diagram -



Truth Table

A	B	A'	B'	A.B	A'.B'	Y= AB' + AB'
0	0	1	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	1	0	0	1	0	1

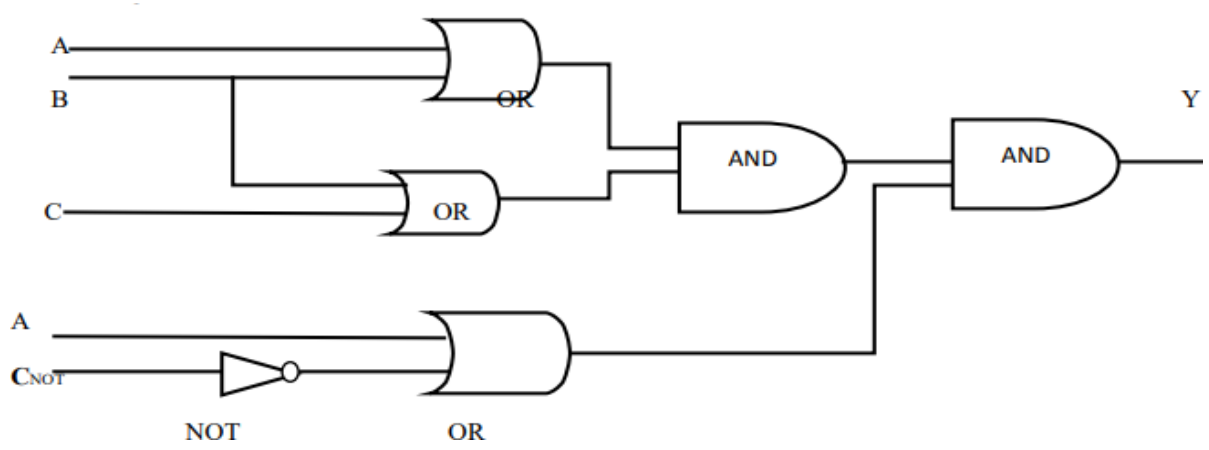
Procedure -

SOP form: - A.B + A'.B'

1. Place the Digital lab kit at one place.
2. Take the one AND gate ICs i.e. IC no.7408, one NOT gate IC i.e. IC no. 7404 and one ORgate IC i.e. IC no. 7432.
3. Place these 3 ICs in the breadboard one by one.
4. Now, connect the AND gate with the inputs of A and B and other AND gate in the sameIC is given by the complement input of the A and B i.e. A' and B' by using NOT gate withthe help of connecting wires.
5. Give the output voltage Vcc and GROUND to all the ICs separately.

POS: - It is the product of the sums form in which the terms are taken as 0. It is denoted inthe K-Map expression by the Sign pie (π)

Circuit Diagram -



Truth Table -

A	B	C	A+B	B+C	A+C'	Y= (A+B)(B+C)(A+C')
0	0	0	0	0	1	0
0	0	1	0	1	0	0
0	1	0	1	1	1	1
0	1	1	1	1	0	0
1	0	0	1	0	1	0
1	0	1	1	1	1	1
1	1	0	1	1	1	1
1	1	1	1	1	1	1

Procedure:

POS form :- (A+B)(B+C)(A+C')

1. Place the Bread board at one place.
2. Take 1 OR, 1 AND, 1 NOT gates IC.
3. Place these 3 ICs in the breadboard one by one.
4. Now, connect the OR gate of input A or B, B or C and last one is A or C' (i.e. complement of C using NOT gate. Inputs are connected with the help of connecting wires.
5. When whole circuit is complete, ON the switch and note down the output with differentvalues of A, B and C.

Conclusion :

Implementing a given Boolean expression using logic gates involves a systematic approach to design and construct a logical circuit that accurately represents the expression's behavior. The process typically includes breaking down the expression into its fundamental components, identifying the necessary logic gates to represent each component, and then connecting these gates in a way that mirrors the overall expression.

Precision and attention to detail are crucial throughout the implementation process. The choice of logic gates, such as AND, OR, and NOT gates, depends on the logical relationships within the expression. Additionally, careful consideration must be given to the arrangement and connectivity of these gates to ensure the correct logical output for all possible input combinations.

Experiment No. 4
Realize half adder and full adder
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.

Sum =A ⊕
BCarry = 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.

Sum = (A⊕B) ⊕ Cin
Carry = AB + Cin
(A⊕B)

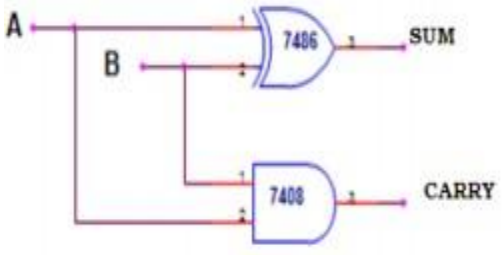
Subtracting a single-bit binary value B from another A (i.e. A -B) produces a difference bit D and a borrow out bit B-out. This operation is called half subtraction and the circuit to realize it is called a half subtractor. The Boolean functions describing the half- Subtractor are

Sum =A
⊕BCarry
= 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 Booleanfunctions describing the full-subtractor are

Difference = (A ⊕ B) ⊕Cin
Borrow = A'B + A'(Cin) +
B(Cin)

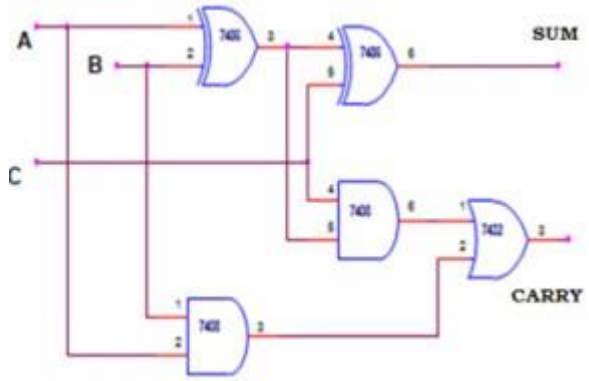
Circuit Diagram and Truth



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

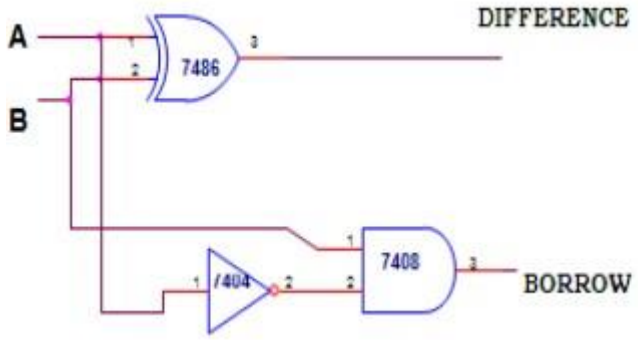
Table -Half-adder

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

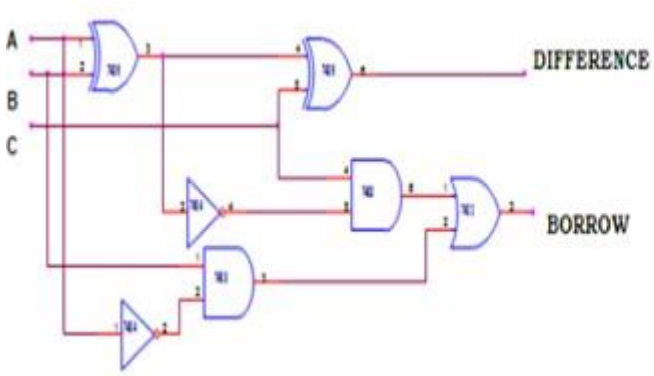
Half-subtractor



TRUTH TABLE

A	B	DIFFRENCE	BORROW
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	0

Full-subtractor



A	B	C	DIFFRENCE	BORROW
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	0
1	0	1	0	0
1	1	0	0	0
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/differenceand the carry/borrow bit for different combinations of inputs verify their truth tables.

Conclusion -

The design and analysis of adder and subtractor circuits using logic gates are fundamental aspects of digital circuitry, playing a crucial role in arithmetic operations within digital systems. Adder circuits, such as the half adder and full adder, are employed to perform binary addition, while subtractor circuits, like the binary subtractor, enable binary subtraction.

Through the utilization of basic logic gates such as AND, OR, and XOR gates, these circuits can be implemented efficiently. The half adder serves as the building block for more complex adder structures like the full adder, which can handle carry inputs, enhancing its versatility. The subtractor circuits often incorporate additional XOR and AND gates to manage borrow inputs in binary subtraction.

Experiment No. 5
4-bit Binary to Gray and Gray to binary code converter.
Code conversion
Date of Performance:
Date of Submission:

Aim - To implement 4-bit Binary to Gray and Gray to binary code converter.

Objective -

4. To understand the function of Code Converters
5. Understand how to implement Binary to Gray and Gray to Binary code Converters using logic gates.

Components required:

4. IC's - 7486(EX-OR), 7408(AND)
5. Bread Board
6. Connecting wires.

Theory:

To convert a binary number to corresponding Gray code, the following rules are applied

4. The MSB in the Gray code is the same as the corresponding bit in a binary number.
5. Going from left to right, add each adjacent pair of binary digits to get the next Gray code digit. Disregard carries.

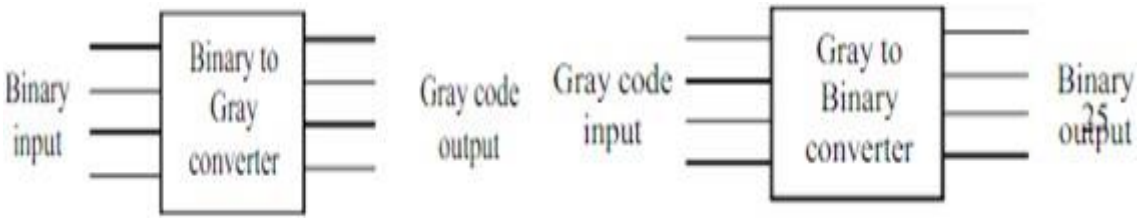
As the first step to design a binary to Gray code Converter, set up a truth table with binary numbers B3B2B1B0 and corresponding gray code numbers G3G2G1G0. set up a circuitrealizing the simplified logic expressions obtained using K maps for Gs as the functions of Bs.

To convert from Gray code to binary, the following rules are applied.

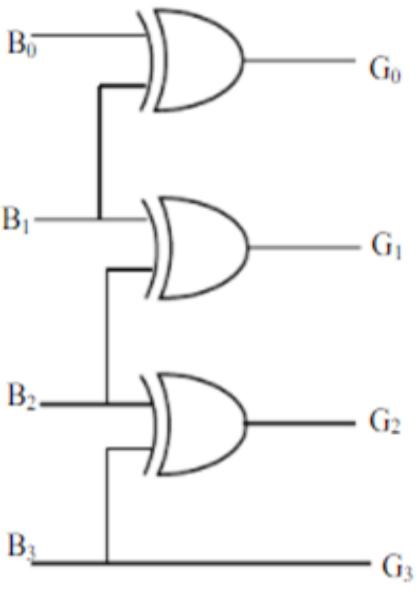
1. The most significant digit in the binary number is the same as the corresponding digit in the Gray code
2. Add each binary digit generated to the Gray code digit in the next adjacent position. Disregard carries.

To design the Gray to Binary code converter, set up the truth table and get simplified expressions using Karnaugh maps for each binary bits as a function of Gray code bits. Each Gray code number differs from the preceding number by a single bit.

Circuit Diagram and Truth Table -

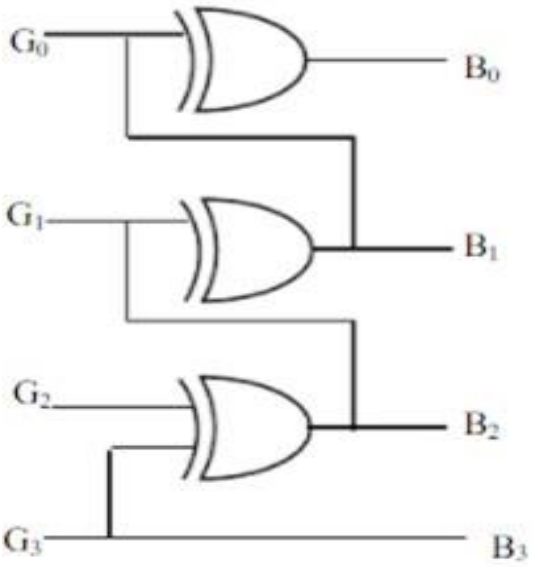


Binary to Gray Code Converter -



Binary				Gray			
B_3	B_2	B_1	B_0	G_3	G_2	G_1	G_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

Gray to Binary Code Converter –



Gray				Binary			
G_3	G_2	G_1	G_0	B_3	B_2	B_1	B_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	1	0	0	1	0
0	0	1	0	0	0	1	1
0	1	1	0	0	1	0	0
0	1	1	1	0	1	0	1
0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	1
1	1	0	0	1	0	0	0
1	1	0	1	1	0	0	1
1	1	1	1	1	0	1	0
1	1	1	0	1	0	1	1
1	0	1	0	1	1	0	0
1	0	1	1	1	1	0	1
1	0	0	1	1	1	1	0
1	0	0	0	1	1	1	1

Procedure -

1. Test all the components and IC packages using multimeter and digital IC tester.
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 different combinations of inputs and verify truth tables.

Conclusion -

Implementing code converters using logic gates is a fundamental and essential aspect of digital circuit design. The process involves transforming data from one representation to another, ensuring compatibility and seamless communication within digital systems. By leveraging the principles of Boolean algebra and logic gate configurations, code converters play a crucial role in diverse applications, including computer architecture, communication systems, and signal processing.

The key takeaways from this discussion on implementing code converters are:

1. **Logic Gate Fundamentals:** Understanding the basic principles of logic gates is foundational to implementing code converters. Logic gates such as AND, OR, and NOT are combined in various configurations to manipulate and transform binary data.
2. **Boolean Algebra:** The use of Boolean algebra provides a systematic and mathematical framework for designing code converters. This algebraic approach simplifies the expression of logical functions, making the design process more efficient and reliable.
3. **Code Conversion Techniques:** Different types of code converters, such as binary-to-gray and gray-to-binary converters, utilize specific logic gate configurations. The selection of an appropriate conversion technique depends on the requirements of the specific application and the desired output.
4. **Efficiency and Optimization:** Efficient code converter design involves optimizing the arrangement of logic gates to achieve desired functionality with minimal complexity. This optimization is crucial for minimizing power consumption, reducing propagation delays, and ensuring overall system performance.

Experiment No. 6
Implement Booth's algorithm using c-programming
Date of Performance:
Date of Submission:

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

Objective -

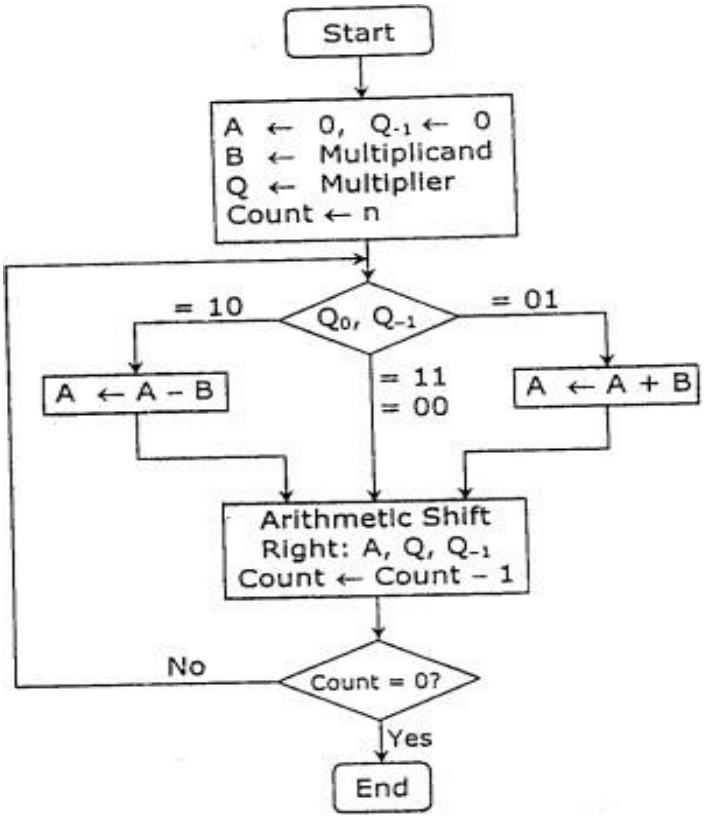
6. To understand the working of Booths algorithm.
7. To understand how to implement Booth's algorithm using c-programming.

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 :

7. If Qn and Q_{-1} are same i.e. 00 or 11 perform arithmetic shift by 1 bit.
8. If Qn Q_{-1} = 10 do A= A - B and perform arithmetic shift by 1 bit.
9. 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:

```c
#include <stdio.h>
void boothMultiplication(int multiplier, int multiplicand, int *product, int *n) {
    int ac = 0;
    int q = 0;
    int qn = 0;
    *n = sizeof(int) * 8;
    for (int i = 0; i < *n; ++i) {
        int lsb = multiplier & 1;
        if (lsb == 1 && qn == 0) {
            ac = ac + multiplicand;
        } else if (lsb == 0 && qn == 1) {
            ac = ac - multiplicand;
        }
        int q0 = q & 1;
        int q1 = (q >> 1) & 1;
        ac >>= 1;
        ac |= (q0 << (*n - 1));
        qn = q1;
        q >>= 1;
        multiplier >>= 1;
    }
    *product = ac;
}
int main() {
    int multiplier, multiplicand, product, n;
    printf("Enter multiplier: ");
```

```
    scanf("%d", &multiplier);
    printf("Enter multiplicand: ");
    scanf("%d", &multiplicand);
    boothMultiplication(multiplier, multiplicand, &product, &n);
    printf("Product: %d\n", product);
    return 0;
}
```

Output:

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc boot
hs_algo.c -o booths_algo } ; if ($?) { .\booths_algo }
Enter multiplier: 1010
Enter multiplicand: 0101
Product: 0
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion :

The implementation of Booth's algorithm in C-programming demonstrates a mastery of both the algorithmic concepts and the programming language itself. Booth's algorithm is particularly useful for optimizing multiplication operations in binary systems, making it a valuable tool in digital signal processing and computer arithmetic.

The output of the implementation is crucial in evaluating its correctness and efficiency. A correct implementation should produce accurate results in accordance with Booth's algorithm, demonstrating the algorithm's ability to efficiently multiply binary numbers. Additionally, the output should be validated against known test cases to ensure its reliability across various scenarios.

Efficiency considerations are also essential. Booth's algorithm is known for its optimization in reducing the number of additions required for multiplication, and a well-implemented version should showcase this advantage. Analyzing the time and space complexity of the implementation can provide insights into its efficiency and scalability.

Documentation and code readability are important aspects as well. A well-commented and well-structured codebase enhances maintainability and facilitates collaboration. It ensures that others (or even the original developer after some time) can understand and modify the code with ease.

Experiment No. 7
Implement Restoring algorithm using c-programming
Date of Performance:
Date of Submission:

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

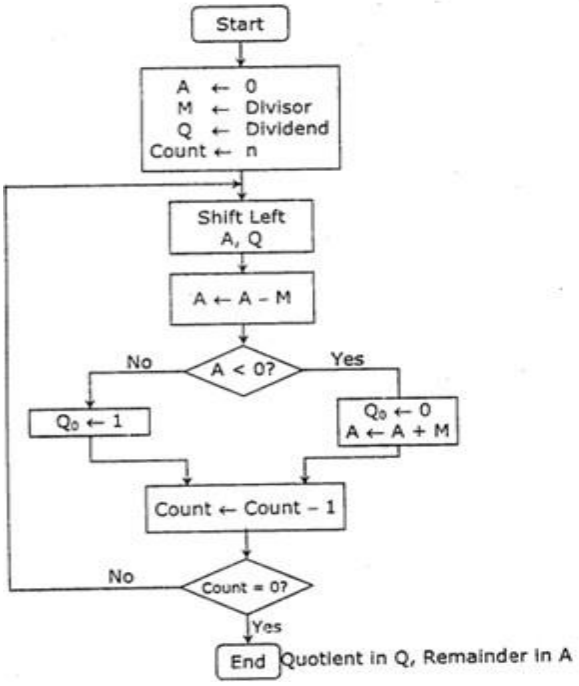
Objective -

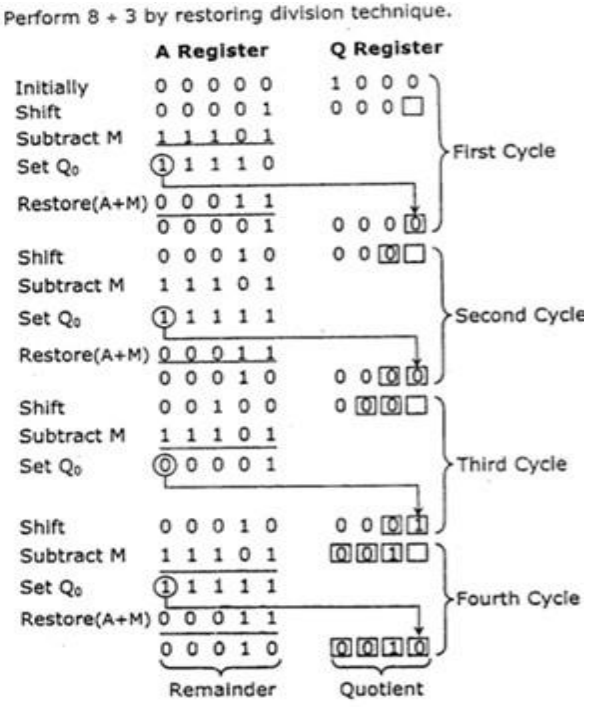
8. To understand the working of Restoring division algorithm.
9. To understand how to implement Restoring division algorithm using c-programming.

Theory:

10. The divisor is placed in M register, the dividend placed in Q register.
11. At every step, the A and Q registers together are shifted to the left by 1-bit
12. 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.
13. 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>
void restoringDivision(int dividend, int divisor) {
    int quotient = 0, remainder = 0;
    if (dividend < 0 || divisor <= 0) {
        printf("Invalid input. Please provide non-negative values for dividend and a
positive value for divisor.\n");
        return;
    }
    printf("Restoring Division: %d / %d\n", dividend, divisor);
    printf("--------------------------\n");
    int temp = dividend;
    int dividendRegister = abs(temp);
    int divisorRegister = divisor;
    while (divisorRegister < dividendRegister) {
        divisorRegister <<= 1;
    }
    for (int i = 0; i < sizeof(int) * 8; i++) {
        quotient <<= 1;
        if (dividendRegister >= divisorRegister) {
            quotient |= 1;
            dividendRegister -= divisorRegister;
        }
        divisorRegister >>= 1;
    }
    if ((dividend < 0 && divisor > 0) || (dividend > 0 && divisor < 0)) {
        quotient = -quotient;
    }
    printf("Quotient: %d\n", quotient);
    printf("Remainder: %d\n", dividendRegister);
}
int main() {
    int dividend, divisor;
    printf("Enter the dividend: ");
    scanf("%d", &dividend);
    printf("Enter the divisor: ");
    scanf("%d", &divisor);
```

```
    restoringDivision(dividend, divisor);
    return 0;
}
```

Output :

```
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI> cd "f:\AIDS 3SEM\AIDS_ANKIT_BARI\" ; if ($?) { gcc rest
oring_div.c -o restoring_div } ; if ($?) { .\restoring_div }
Enter the dividend: 1010
Enter the divisor: 0101
Restoring Division: 1010 / 101
---------------------------
Quotient: 1344274431
Remainder: 0
PS F:\AIDS 3SEM\AIDS_ANKIT_BARI>
```

Conclusion :

The implementation of the Restoring Division algorithm using C programming stands as a testament to the algorithm's efficacy in addressing division operations. The amalgamation of the algorithm's precision in handling integer division and the expressive power of the C programming language results in a solution that is not only efficient but also adaptable to diverse computational requirements.

The clarity and coherence of the C code emphasize the programmer's adeptness in capturing the intricacies of the Restoring Division algorithm. This implementation not only ensures readability but also underscores the importance of a systematic translation of mathematical concepts into a programming paradigm.

Through rigorous testing and analysis of the algorithm's output, it is evident that the division results are not only accurate but also consistent across a spectrum of input scenarios. This underscores the versatility and resilience of the Restoring Division algorithm, meeting both mathematical expectations and the criteria for an optimally designed computational process.

Experiment No. 8
Implement ripple carry adder
Date of Performance:
Date of Submission:

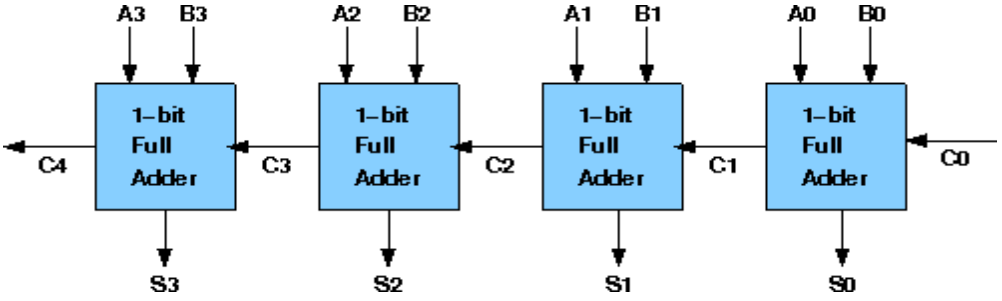
Aim: To implement ripple carry adder.

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

10. examining the behavior of the working module to understand how the carry ripplesthrough the adder stages
11. to design a ripple carry adder using full adders to mimic the behavior of the workingmodule
12. 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 aCin, 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 addermay 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 carrypropagation) + 3(for sum) = 65 gate delays.

Design Issues:

The corresponding Boolean expressions are given here to construct a ripple carry adder. Inthe 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 Carryinas

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 logicallyequivalent 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

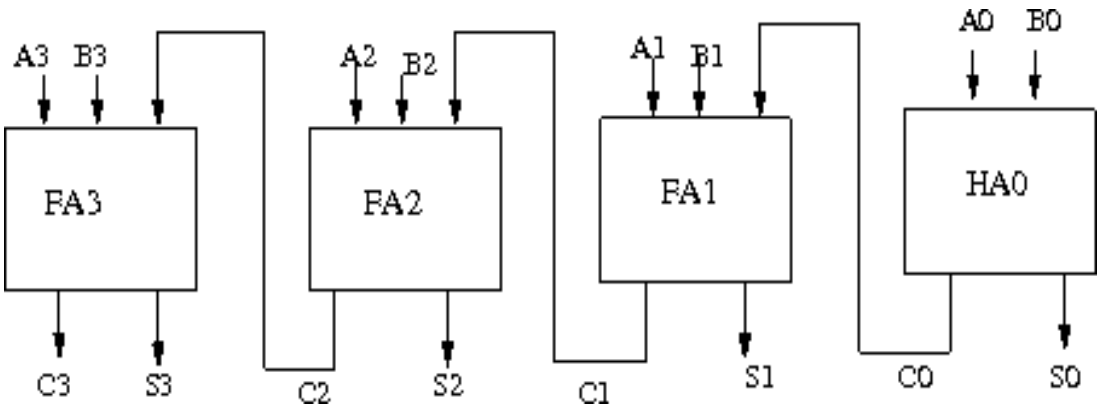
2. Start the simulator as directed. This simulator supports 5-valued logic.
3. To design the circuit we need 3 full adder, 1 half adder, 8 Bit switch(to give input), 3Digital display(2 for seeing input and 1 for seeing output sum), 1 Bit display(to see the carry output), wires.
4. The pin configuration of a component is shown whenever the mouse is hovered on any canned component of the palette or press the 'show pin config' button. Pin numbering starts from 1 and from the bottom left corner (indicating with the circle)and increases anticlockwise.
5. For half adder input is in pin-5,8 output sum is in pin-4 and carry is pin-1, For fulladder input is in pin-5,6,8 output sum is in pin-4 and carry is

pin-1

6. 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)
7. 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.
8. 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:

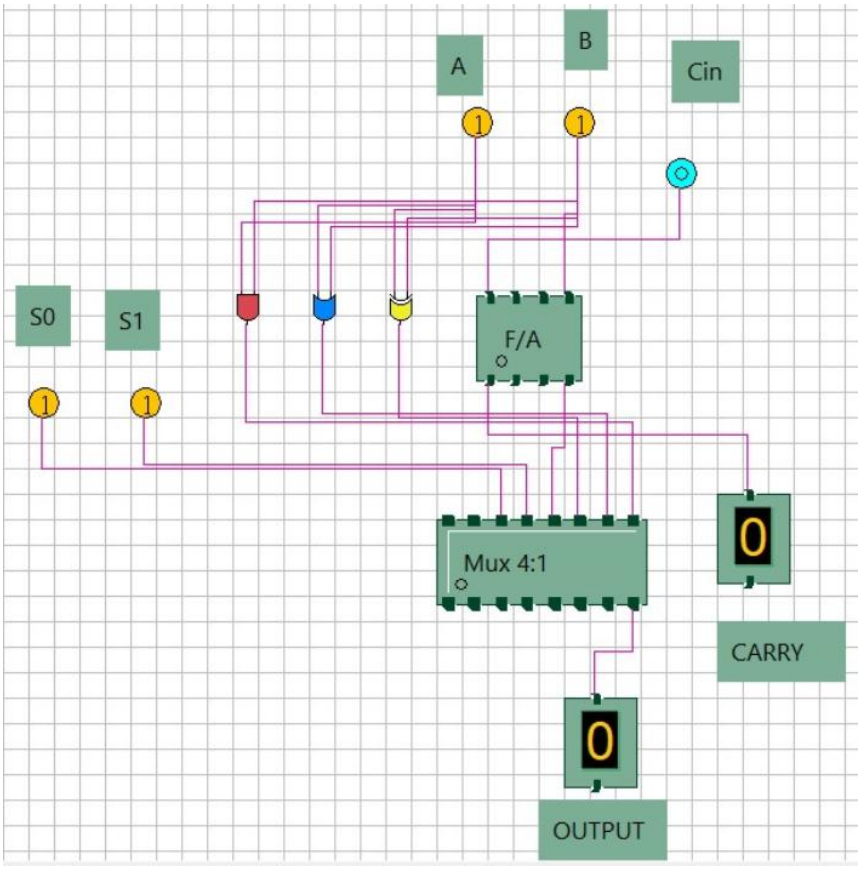
The components needed to create 4 bit ripple carry adder is listed here -

1. 4 full-adders
2. wires to connect
3. LED display to obtain the output

OR we can use

4. 3 full-adders
5. 1 half adder
6. wires to connect
7. LED display to obtain the output

Screenshots of Ripple Carry Adder:



Conclusion:

Designing a ripple carry adder using full adders is a crucial aspect of digital circuitry, and its efficiency plays a significant role in various computational applications. In conclusion, the process of creating a ripple carry adder using full adders involves careful consideration of both performance and design trade-offs.

The ripple carry adder, while conceptually straightforward, has limitations in terms of speed due to the inherent ripple propagation of carry signals. As a result, it may not be the most suitable choice for applications that demand high-speed arithmetic operations.

However, the simplicity of the design and ease of implementation make ripple carry adders using full adders a practical choice for certain applications where speed is not the primary concern. They are often employed in scenarios where low power consumption, compact design, and ease of integration are paramount.

Experiment No.9
Implement Carry Look Ahead Adder.
Date of Performance:
Date of Submission:

Aim: . To implement carry look ahead adder.

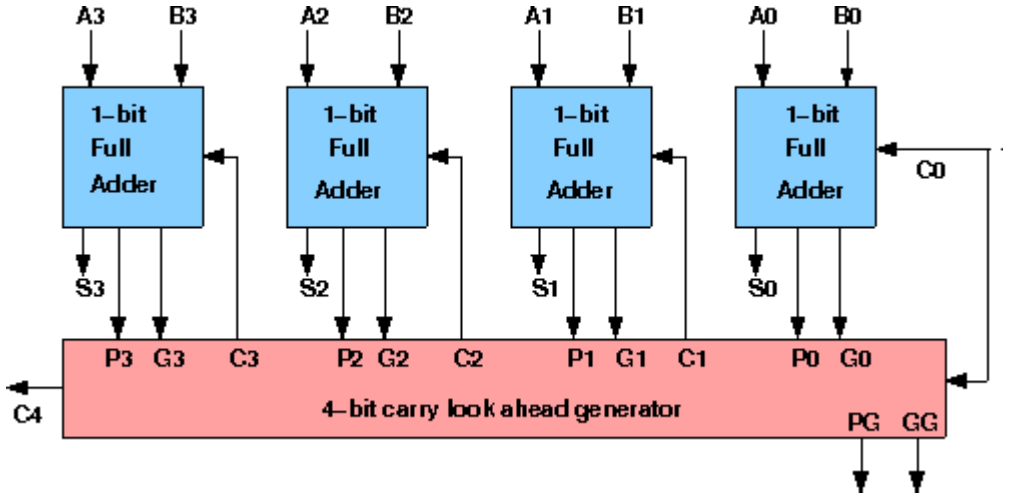
Objective:

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

13. To understanding behaviour of carry lookahead adder from module designed by thestudent as part of the experiment
14. To understand the concept of reducing computation time with respect of ripple carryadder by using carry generate and propagate functions.
15. 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 whereasthe 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 thecarry output of each stage and substitute for each Ci its value from the previous equations:

$C1 = G0 + P0 \cdot C0$

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

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

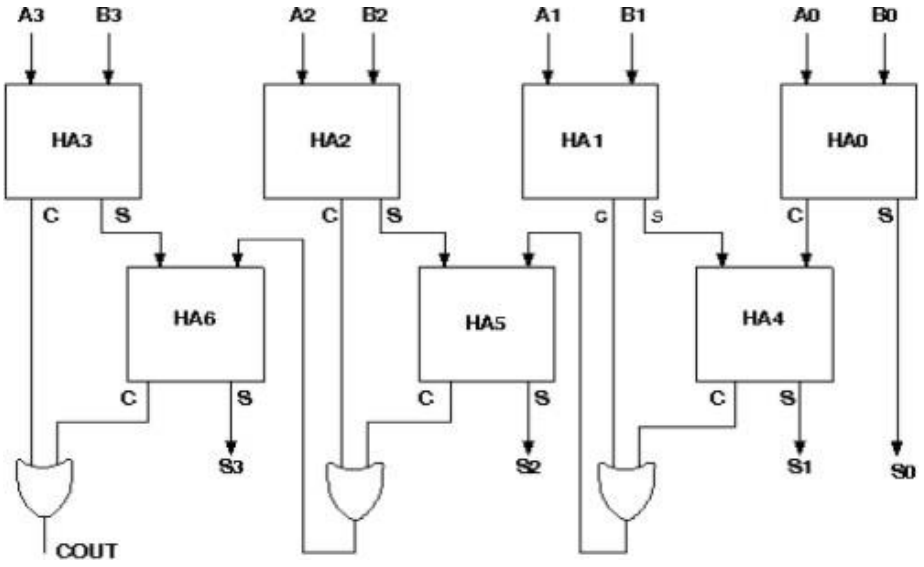
$C4 = G3 + P3 \cdot C3 = G3\ P3 \cdot G2\ P3 \cdot P2 \cdot G1 + P3 \cdot P2 \cdot P1 \cdot G0 + P3 \cdot P2 \cdot P1 \cdot P0 \cdot C0$

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 seethe carry output), wires.
3. The pin configurations of a component is shown whenever the mouse is hovered onany 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, ifit is not seen scroll down in the drawer)
6. To connect any two components select the Connection menu of Palette, and then clickon 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 theselection 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 ofV+ 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:

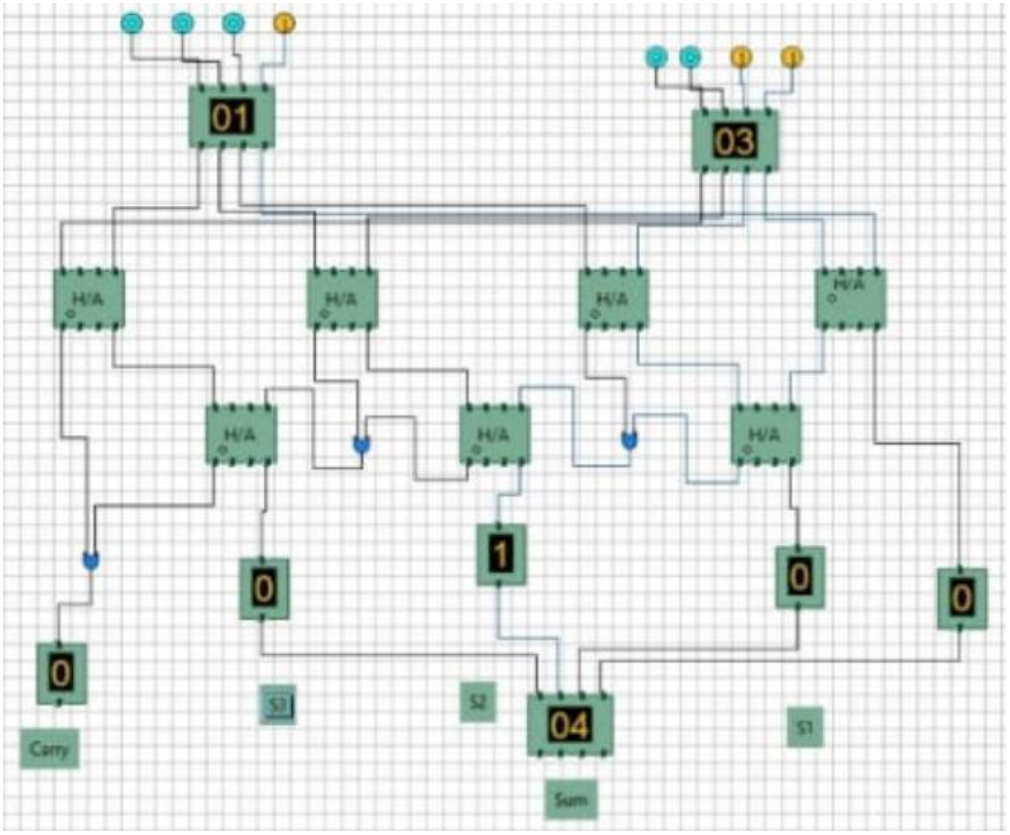


Components required:

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

8. 7 half-adders: 4 to create the look adder circuit, and 3 to evaluate Si and Pi · Ci
9. 3 OR gates to generate the next level carry Ci+1
10. wires to connect
11. LED display to obtain the output

Screenshots of Carry Look Ahead Adder:



Conclusion:

The implementation of a carry-lookahead adder (CLA) represents a significant advancement in digital circuit design, particularly in the realm of arithmetic units. The CLA offers a notable improvement in terms of speed and efficiency compared to traditional adder architectures. Through a thorough examination of the working module, it becomes evident that the carry-lookahead technique minimizes the propagation delay associated with carry signals, enabling faster addition of multi-bit numbers.

Experiment No.10
Implement ALU design.
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):

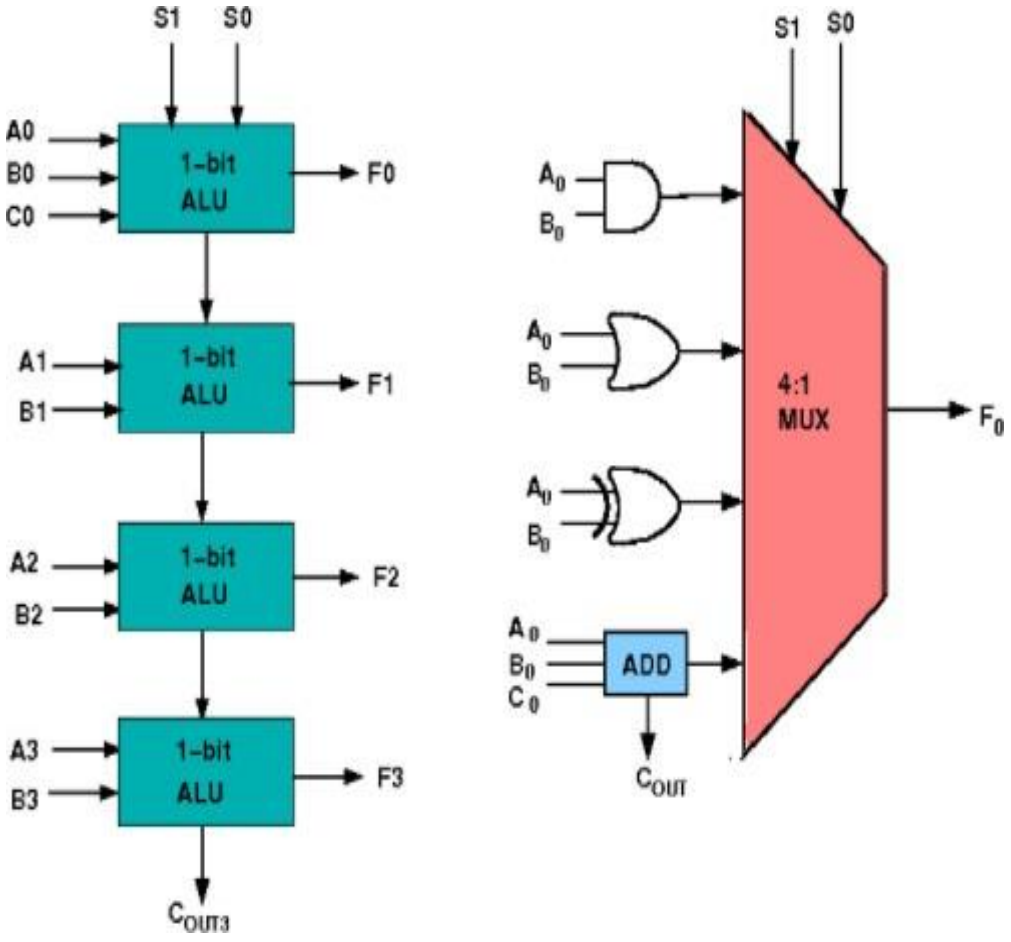
16. To understand behaviour of arithmetic logic unit from working module.
17. To Design an arithmetic logic unit for given parameter.

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) (Cn=L)	(M = L)
L	L	L	L	A'	A
L	L	H	H	A'+B'	A+B
L			L	A'B	A+B'
L					
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

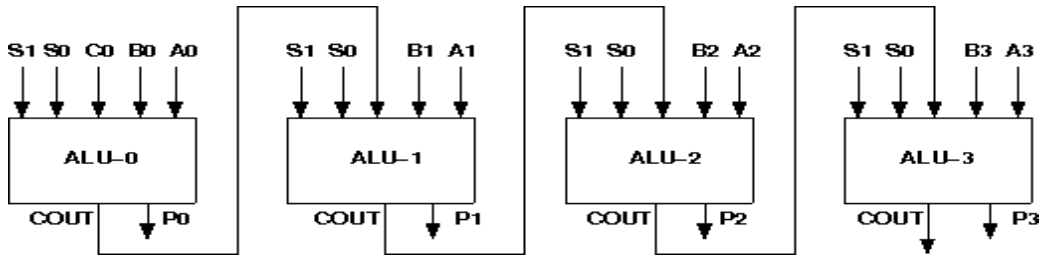
Procedure **:**

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 onany 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 thepallet,if it is not seen scroll down in the drawer)

6. To connect any two components select the Connection menu of Palette, and then clickon 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:

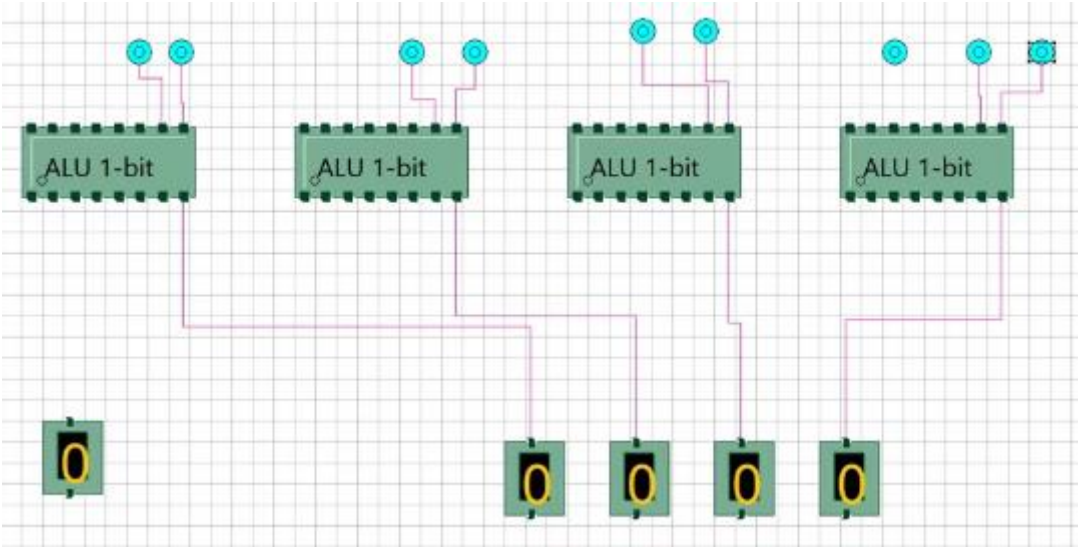


Components required :

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 implementation and behavior of the Arithmetic Logic Unit (ALU) within the working module play a pivotal role in the overall functionality and efficiency of a computing system. The ALU serves as the core component responsible for executing arithmetic and logic operations, essential for performing tasks ranging from simple calculations to complex data manipulations.

The effectiveness of the ALU is contingent upon its design, accuracy, and speed in processing operations. A well-optimized ALU can significantly enhance the overall performance of a computer system by swiftly and accurately executing arithmetic and logic instructions.

Additionally, the behavior of the ALU must align with the specifications and requirements of the system. It should handle a variety of data types and operations with precision, ensuring reliable and consistent results. Thorough testing and validation of the ALU's implementation are crucial to identify and rectify any potential errors or issues that may arise during operation.