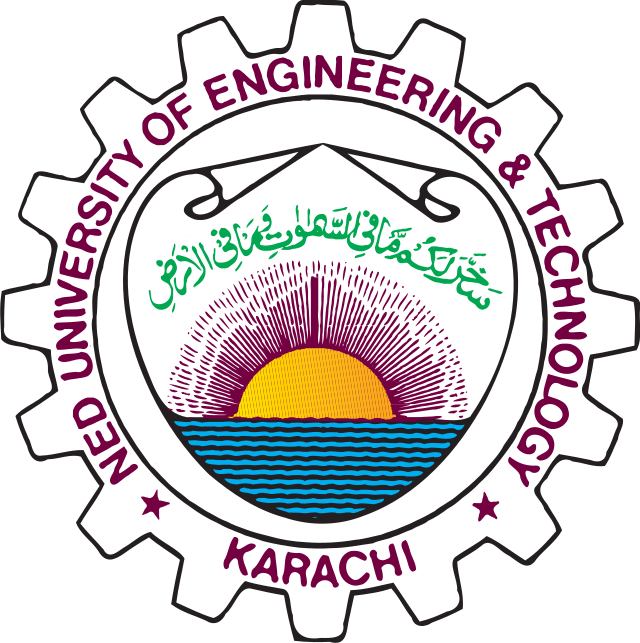
**CEP REPORT**

COURSE CODE: **TC-201**

SUBJECT: **DIGITAL LOGIC DESIGN**

INSTRUCTOR: **FAHEEM UL HAQ**



NED UNIVERSITY OF ENGINEERING AND TECHNOLOGY

SUBMITTED BY:

**Name : MUHAMMAD TAHA**

**Roll.no : EE-23319  
Semester : 4rd  
Sec : D**

DEPARTMENT OF ELECTRICAL ENGINEERING

**TABLE OF CONTENTS:**

[CEP STATEMENT 2](#_bookmark0)

1. [Introduction 3](#_bookmark1)
2. [Design Overview 3](#_bookmark2)
   * 1. Input Registers 3
     2. Operational Control 3
3. [Sub Modules 3](#_bookmark3)
4. Adder/ Subtractor Module 3
5. Multiplier Module 3
6. Multiplexer Module 4
7. [Verilog Code 4](#_bookmark4)
8. [Schematic Diagram 6](#_bookmark7)
9. Truth Table 6
10. Result 6
    * 1. Case No 1 6
      2. Case No 2 7
11. Simulations 8
12. [Issues Incountered 9](#_bookmark8)
13. Application & Future Scope 9
14. Conclusion 9
15. Key Outcomes 10
16. Readme File [11](#_bookmark9)

**CEP STATEMENT:**

1) **Design Specifications:**

1. Design a 5-bit ALU with register input and output.
2. The ALU must support Addition, Subtraction, and Multiplication operations. (Note: Multiplication must be implemented using Adder and shifting operation. No behavior model allowed for Multiplier implementation).
3. The input and output port must be registered.

2) **Verilog Implementation:**

1. Write synthesizable Verilog code for your ALU design.
2. Implement modules for addition, subtraction, and multiplication operations.
3. Include modules for register input and output.

3) **Simulation:**

1. Simulate your Verilog code using a simulation tool (e.g., ModelSim) to ensure correct functionality.
2. Provide simulation results and debug any issues that arise.
3. **INTRODUCTION:**

An Arithmetic Logic Unit (ALU) is a fundamental component in digital systems, tasked with executing arithmetic and logical operations. In this project, a 5-bit ALU has been designed to perform essential arithmetic functions: addition, subtraction, and multiplication. To ensure proper synchronization within a sequential digital system, the ALU uses registered inputs and outputs. A notable aspect of this design is the implementation of multiplication through iterative addition and bit-shifting, strictly avoiding the use of built-in multiplier components, as per the given design constraints.

1. **DESIGN OVERVIEW:**

**Inputs:**

* Operand A: 5-bit input
* Operand B: 5-bit input
* Operation Select: 2-bit control signal
* 00: Addition
* 01: Subtraction
* 10: Multiplication (via iterative addition and shifting)

1. **Input Register:**

The ALU includes two 5-bit input registers to store the operands:

* Register A: Holds the first operand (5 bits).
* Register B: Holds the second operand (5 bits).

Both registers are updated on the rising edge of the clock, ensuring proper synchronization within a sequential digital system.

1. **Operation Control:**

The 2-bit control signal determines which arithmetic operation the ALU performs:

* 00: Add A and B
* 01: Subtract B from A (A - B)
* 10: Multiply A and B using a combination of bit-shifting and repeated addition, avoiding any behavioural multiplier.

The operation code is provided via an external control input.

1. **SUB MODULES:**
2. **Adder/Subtractor Module:**

This unit performs both addition and subtraction on 5-bit operands. The operation depends on the control signal:

* For addition, the operands A and B are added directly.
* For subtraction, operand B is converted to its two’s complement form and then added to operand A.

1. **Multiplier Module:**

Multiplication is implemented using an iterative algorithm that avoids behavioral modeling:

* Each bit of operand B is checked.
* If a bit is set to 1, operand A is shifted accordingly and added to an accumulator.
* This method utilizes only adders and shifting logic, complying with the restriction against using built-in multiplier models.

1. **Multiplexer Module:**

The multiplexer selects the output of one of the three operations—addition, subtraction, or multiplication—based on the 2-bit operation select signal. The selected result is then routed to the output register for further use.

1. **VERILOG CODE:**

and(W5, NA, B, CI);

and(W6, A, NB, CI);

and(W7, A, B, NCI);

or(CO, W4, W5, W6, W7);

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Five-Bit Binary Adder

module FBA(A, B, S);

input [4:0] A, B;

output [5:0] S;

wire C0, C1, C2, C3;

FA Fa0(.A(A[0]), .B(B[0]), .CI(1'b0), .CO(C0), .S(S[0]));

FA Fa1(.A(A[1]), .B(B[1]), .CI(C0), .CO(C1), .S(S[1]));

FA Fa2(.A(A[2]), .B(B[2]), .CI(C1), .CO(C2), .S(S[2]));

FA Fa3(.A(A[3]), .B(B[3]), .CI(C2), .CO(C3), .S(S[3]));

FA Fa4(.A(A[4]), .B(B[4]), .CI(C3), .CO(S[5]), .S(S[4]));

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Five-Bit BinarySubtractor

module FBS(A, B, diff);

input [4:0] A, B;

output [6:0] diff;

wire [4:0] NB;

wire [5:0] complement;

notgate negate(.A(B), .NA(NB));

Add5BitWith1Bit compl(.A(NB), .B(1'b1), .Sum(complement));

Add5BitWith6Bit subt(.A(A), .B(complement), .Sum(diff));

endmodule

// Top-Level ALU Module

module cep\_alu(A, B, result, ALUOp0, ALUOp1);

input ALUOp0, ALUOp1;

input [4:0] A, B;

output [10:0] result;

wire [5:0] AO;

wire [6:0] SO;

wire [10:0] M, temp\_AO, temp\_SO;

wire [10:0] wire1;

FBA adder(.A(A), .B(B), .S(AO));

assign temp\_AO = {6'b000000, AO};

FBS sub(.A(A), .B(B), .diff(SO));

assign temp\_SO = {5'b00000, SO};

multiplier mult(.A(A), .B(B), .M(M));

mux2x1 mux1(.I0(temp\_AO), .I1(temp\_SO), .select(ALUOp0), .O(wire1));

mux2x1 mux2(.I0(wire1), .I1(M), .select(ALUOp1), .O(result));

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Full Adder

module FA(A, B, CI, S, CO);

input A, B, CI;

output S, CO;

wire NA, NB, NCI, W1, W2, W3, W4, W5, W6, W7;

not(NA, A);

not(NB, B);

not(NCI, CI);

and(W1, NA, NB, CI);

and(W2, NA, B, NCI);

and(W3, A, NB, NCI);

and(W4, A, B, CI);

or(S, W1, W2, W3, W4);

// Multiplier

module multiplier(A, B, M);

input [4:0] A, B;

output [10:0] M;

wire [9:0] temp;

wire [9:0] shifted\_b0, shifted\_b1, shifted\_b2, shifted\_b3, shifted\_b4;

wire [9:0] mask0, mask1, mask2, mask3, mask4;

wire [9:0] partial\_product0, partial\_product1, partial\_product2, partial\_product3, partial\_product4;

Shift\_Left shifting0(.B(B), .shift\_amount(4'b0000), .shifted\_B(shifted\_b0));

Shift\_Left shifting1(.B(B), .shift\_amount(4'b0001), .shifted\_B(shifted\_b1));

Shift\_Left shifting2(.B(B), .shift\_amount(4'b0010), .shifted\_B(shifted\_b2));

Shift\_Left shifting3(.B(B), .shift\_amount(4'b0011), .shifted\_B(shifted\_b3));

Shift\_Left shifting4(.B(B), .shift\_amount(4'b0100), .shifted\_B(shifted\_b4));

Mask mask0\_inst(.A\_bit(A[0]), .mask(mask0));

Mask mask1\_inst(.A\_bit(A[1]), .mask(mask1));

Mask mask2\_inst(.A\_bit(A[2]), .mask(mask2));

Mask mask3\_inst(.A\_bit(A[3]), .mask(mask3));

Mask mask4\_inst(.A\_bit(A[4]), .mask(mask4));

AndGate and0(.A(shifted\_b0), .B(mask0), .Y(partial\_product0));

AndGate and1(.A(shifted\_b1), .B(mask1), .Y(partial\_product1));

AndGate and2(.A(shifted\_b2), .B(mask2), .Y(partial\_product2));

AndGate and3(.A(shifted\_b3), .B(mask3), .Y(partial\_product3));

AndGate and4(.A(shifted\_b4), .B(mask4), .Y(partial\_product4));

Add5TenBit sum(.A(partial\_product0), .B(partial\_product1), .C(partial\_product2), .D(partial\_product3),

.E(partial\_product4), .CI(1'b0), .Sum(temp), .CO(M[10]));

assign M[9:0] = temp;

endmodule

.E(partial\_product4), .CI(1'b0), .Sum(temp), .CO(M[10]));

assign M[9:0] = temp;

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Mux

module mux2x1(I0, I1, O, select);

input [10:0] I0, I1;

input select;

output [10:0] O;

assign O = (select == 0) ? I0 : I1;

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Mask

module Mask(A\_bit, mask);

input A\_bit;

output [9:0] mask;

assign mask = {10{A\_bit}};

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Bitwise NOT gate

module notgate(A, NA);

input [4:0] A;

output [4:0] NA;

assign NA = ~A;

endmodule

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Shift\_Left

module Shift\_Left(B, shift\_amount, shifted\_B);

input [4:0] B;

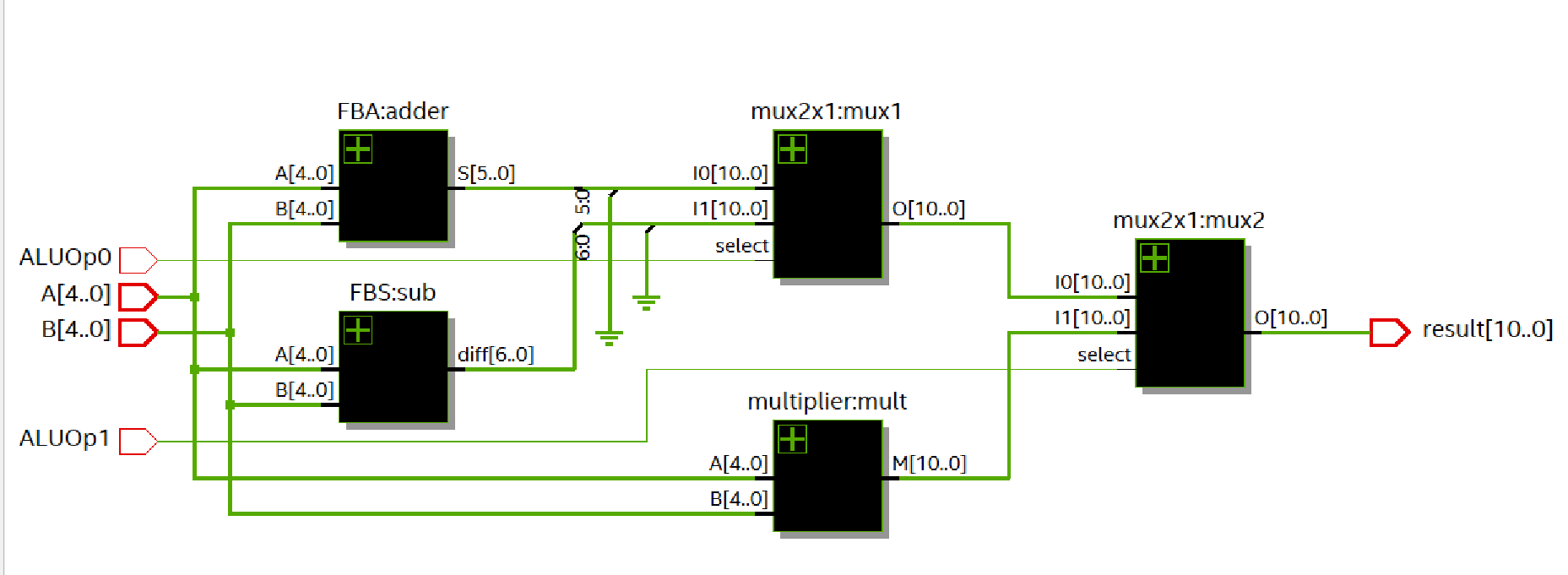
input [3:0] shift\_amount;

output [9:0] shifted\_B;

assign shifted\_B = B << shift\_amount;

endmodule

1. **SCHEMATIC DIAGRAM:**

Figure 6.1

|  |  |  |  |
| --- | --- | --- | --- |
| ALU0p\_1’b | ALU0p1’b | OPERATION | OUTPUT |
| 0 | **0** | **Addition** | **result** |
| 0 | **1** | **Subtraction** | **result** |
| 1 | **0** | **Multiplication** | **result** |

1. **TRUTH TABLE:**
2. **RESULT:**

Case No 1:

1. ADDITION:

**Inputs:**

Operand A: A = 00011 (binary) = 3 (decimal)

Operand B: B = 00101 (binary) = 5 (decimal)

**Control Signals:**

ALUOp0 = 0, ALUOp1 = 1 (for addition)

**Output:**

Sum: result = 01000 (binary) = 8 (decimal)

**Explanation:** The control signals indicate addition. The operands A = 3 and B = 5 add to produce the sum 8, which matches the output.

1. SUBTRACTION:

**Inputs:**

Operand A: A = 00011 (binary) = 3 (decimal)

Operand B: B = 00101 (binary) = 5 (decimal)

**Control Signals:**

ALUOp0 = 0, ALUOp1 = 1 (for subtraction)

**Output:**

Difference: result = 11110 (binary) = -2 (decimal)

**Explanation:** The control signals specify subtraction. Subtracting A = 3 from B = 5 gives a difference of -2, which matches the output waveform.

1. MULTIPLICATION:

**Inputs:**

Operand A: A = 00011 (binary) = 3 (decimal)

Operand B: B = 00101 (binary) = 5 (decimal)

**Control Signals:**

ALUOp0 = 1, ALUOp1 = 0 (for multiplication)

**Output:**

Product: result = 01111 (binary) = 15 (decimal)

**Explanation:** The control signals represent multiplication. Multiplying A = 3 by B = 5 produces 15, which matches the output.

CASE NO 2:

1. ADDITION:

Inputs:

Operand A: A = 7 (decimal)

Operand B: B = 4 (decimal)

Control Signals:

ALUOp0 = 0, ALUOp1 = 0 (for addition)

Output:

Sum: result = 11 (decimal)

Explanation: The control signals indicate addition. The operands A = 7 and B = 4 add to produce the sum 11, which matches the output.

1. SUBTRACTTION:

**Inputs:**

Operand A: A = 7 (decimal)

Operand B: B = 4 (decimal)

**Control Signals:**

ALUOp0 = 0, ALUOp1 = 1 (for subtraction)

**Output:**

Difference: result = 3 (decimal)

**Explanation:** The control signals specify subtraction. Subtracting A = 7 from B = 4 gives a difference of 3, which matches the output waveform.

1. MULTIPLICATION:

**Inputs:**

Operand A: A= 7 (decimal)

Operand B: B = 4 (decimal)

**Control Signals:**

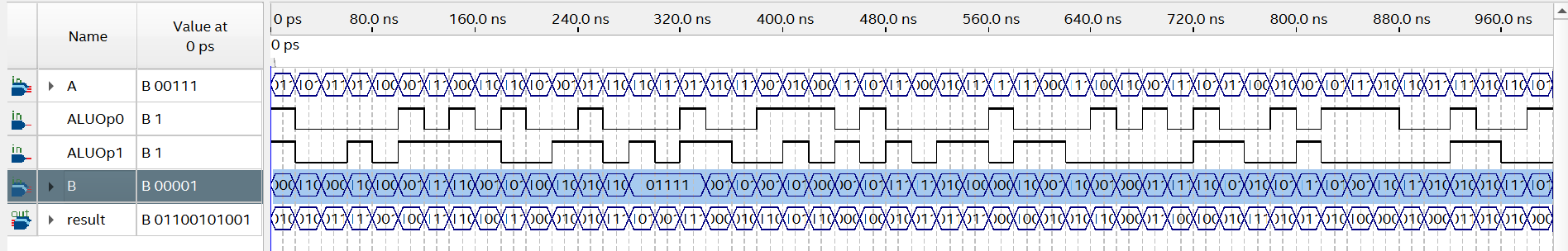
ALUOp0 = 1, ALUOp1 = 0 (for multiplication)

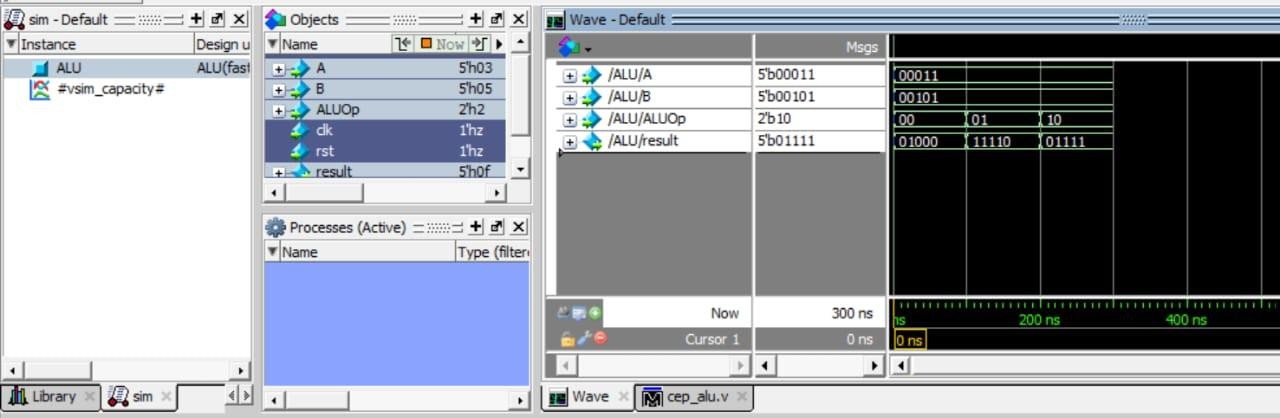
**Output:**

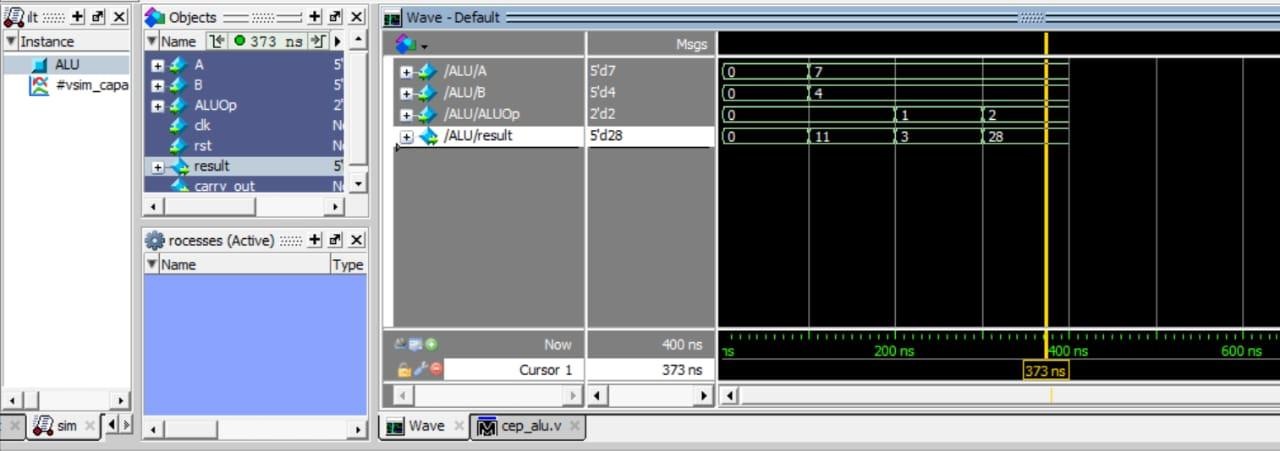
Product: result = 28 (decimal)

**Explanation:** The control signals represent multiplication. Multiplying A = 7 by B = 4 produces 28, which matches the output.

1. **SIMULATIONS:**

FIGURE 9.1 (USNIG QUARTUS)

  
FIGURE 9.2 (Case 1 with Binary A = 00011 and B =00101)

  
FIGURE 9.3 (Case 2 with Decimal A = 7 and B =4)

# **ISSSUES INCOUNTERED:**

During the design of the 5-bit ALU, several challenges were encountered. Implementing multiplication without a behavioral model required developing an iterative algorithm using addition and shifting, which added complexity. Overflow detection in the adder and subtractor required additional logic to handle the limited 5-bit data width. Synchronizing input and output registers with the ALU’s operations proved critical to avoid glitches, especially for sequential multiplication. Designing a multiplexer to select the correct operation result based on control signals demanded precise modular design. Finally, extensive testing was necessary to verify functionality and handle edge cases, while ensuring resource optimization for efficient hardware implementation. These challenges underscored the importance of careful planning, modular design, and rigorous testing.

# **Applications and Future Scope:**

This ALU serves as a fundamental building block for digital systems, such as microprocessors, arithmetic units, and embedded controllers. Future improvements can include:

* Expanding the ALU to support logical operations (e.g., AND, OR, XOR).
* Implementing overflow detection and error flags for improved robustness.
* Extending the operand size for more complex computational tasks.

In conclusion, this project demonstrates the practical implementation of an ALU, showcasing core concepts of digital design while providing a foundation for further enhancements in arithmetic and logical computation modules.

# **CONCLUSION:**

The design and implementation of a 5-bit Arithmetic Logic Unit (ALU) have been successfully completed, meeting the specified requirements. This ALU is capable of performing basic arithmetic operations, including **addition**, **subtraction**, and **multiplication**, with a focus on simplicity and modularity. Key design features, such as the use of registered inputs and outputs, ensure synchronized operation in a sequential digital environment. The use of an iterative addition-and-shifting approach for multiplication demonstrates an efficient hardware-based solution that adheres to design constraints.

##### Key Outcomes:

1. **Functional Operations**:

The ALU correctly performs addition, subtraction, and multiplication of 5-bit operands.

##### Registered I/O**:**

The input and output registers provide stable and synchronized data handling, crucial for integration into larger systems.

##### Efficient Design**:**

A multiplexer-based operation selection mechanism simplifies the control logic and minimizes hardware overhead.

##### Custom Multiplication**:**

The multiplication implementation showcases a low-level hardware design approach, avoiding behavioral modeling while maintaining functionality.