Digital Electronics (BEEE206L) - Course Based Design Project

DESIGN AND IMPLEMENTATION OF A 16-BIT ALU FROM DISCRETE LOGIC GATES

Submitted By:

Atharva Verma (24BEE0157)
Jatin Satapathy (24BEE0169)
Ayan Saikia (24BEE0123)
Naman Agarwal (24BEE0173)

Department of Electrical and Electronics Engineering
School of Electrical Engineering
SELECT

Under the Guidance of:

Dr. Rajesh Kumar Lenka
Assistant Professor Senior
Department of Electrical and Electronics Engineering
School of Electrical Engineering

Department of Electrical and Electronics Engineering
Vellore Institute of Technology
Vellore, Tamil Nadu-632014

Academic Year: 2025-26

Acknowledgement

We would like to express our heartfelt gratitude to our project guide, **Dr. Rajesh Kumar Lenka**, *Assistant Professor Senior*, for his invaluable guidance, constant support, and encouragement throughout this project. His insights were instrumental in navigating the complexities of the design and implementation phases.

We also extend our thanks to the Head of Department, the faculty members of the Department of Electrical and Electronics Engineering, and **Vellore Institute of Technology** for providing the necessary facilities and a conducive environment for this work.

Date: November 2, 2025

Abstract

The Arithmetic Logic Unit (ALU) is a fundamental building block of all modern central processing units (CPUs). This project presents the design and implementation of a 16-bit, 4-function ALU constructed entirely from basic logic gate ICs, **deliberately avoiding the use of pre-integrated ALU or adder ICs. The primary objective is to demonstrate a comprehensive understanding of digital logic design from first principles.** The ALU performs four operations: 16-bit addition, subtraction, bitwise AND, and bitwise OR. The design is based on a modular, bit-sliced architecture. The arithmetic unit employs 2's complement methodology to perform subtraction using the same hardware as addition. Operation selection is achieved through a 4-to-1 Multiplexer-based control unit for each bit, using 74HC153 ICs. The system was first designed and simulated at a 1-bit level before being scaled to the full 16-bit implementation on breadboards. This report details the complete design process, from the underlying theory of 2's complement arithmetic and full adders to the final system architecture, hardware selection (74HC-series ICs), and testing procedures. The successful operation of the built prototype validates the design and provides a tangible demonstration of core computer architecture concepts.

Contents

1 Introduction

The Arithmetic Logic Unit (ALU) is the computational heart of a microprocessor. It is the combinational logic circuit responsible for performing all arithmetic operations (like addition, subtraction) and bitwise logic operations (like AND, OR, NOT) on data operands. Without an ALU, a computer would not be able to perform calculations.

While modern computing relies on highly integrated microprocessors where the ALU is a microscopic, complex component, there is immense educational value in building one from the ground up. The motivation for this project is to demystify this ”black box” by constructing a functional ALU using **only** basic, discrete logic gate ICs (Integrated Circuits).

This project focuses on the design and hardware implementation of a 16-bit ALU. The unit will accept two 16-bit operands (A and B) and a 2-bit operation selector code ($S[1:0]$) as inputs. Based on the selector code, it will perform one of four operations (Addition, Subtraction, AND, OR) and output a 16-bit result. The scope of this project is limited to these four functions and is implemented as a combinational circuit (i.e., it has no memory or clock).

2 Literature Review / Background Study

The design of this ALU is founded on several key principles of digital electronics.

2.1 Boolean Algebra and Logic Gates

At the lowest level, all digital computation is based on Boolean algebra. The basic logic gates-AND (model 74HC08), OR (model 74HC32), and XOR (model 74HC86) are the primary components used in this project. These gates perform the fundamental bitwise operations.

2.2 The 1-bit Full Adder

The core of any arithmetic unit is the adder. A 1-bit Full Adder is a combinational circuit that adds three 1-bit inputs: A, B, and a Carry-In (C_{in}). It produces two 1-bit outputs: a Sum (S) and a Carry-Out (C_{out}). The logic equations are:

$$S = A \oplus B \oplus C_{in}$$

$$C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$$

By connecting 16 of these full adders in series, with the C_{out} of one stage feeding the C_{in} of the next, a 16-bit ”Ripple-Carry Adder” can be built.

2.3 2’s Complement Arithmetic

To perform subtraction using an adder, the 2’s complement number system is used. This system is the standard method for representing signed integers in modern computers. The 2’s complement of a binary number is found by inverting all its bits (1’s complement) and then adding 1.

The utility of this system is that subtraction becomes addition. The operation $A - B$ is equivalent to $A + (-B)$. In 2’s complement, $-B$ is represented as $\overline{B} + 1$. Therefore, the subtraction $A - B$ is performed as the addition:

$$A + \overline{B} + 1$$

This is a critical design insight, as it allows us to perform both addition and subtraction with a single adder circuit, using some simple control logic.

3 Problem Definition

The problem is to design and build a 16-bit Arithmetic Logic Unit from scratch using only standard 74HC-series logic gate ICs. The final hardware, built on breadboards, must be able to perform four distinct operations on two 16-bit inputs (A and B) selected by a 2-bit control signal ($S[1:0]$).

The specific challenges include:

- Designing a scalable 1-bit "slice" that can be replicated 16 times.
- Implementing both addition and subtraction efficiently, ideally using a single adder circuit.
- Managing the complexity of wiring (hundreds of connections) and power distribution on a large-scale breadboard prototype.
- Ensuring the high-frequency switching noise from dozens of ICs does not cause instability or logic errors.

4 Objectives

The primary objectives for this project are:

- To design a 16-bit ALU capable of performing 16-bit ADD, SUBTRACT, bitwise AND, and bitwise OR.
- To implement this design in hardware using only 74HC-series discrete logic ICs (no ALU or adder chips).
- To use a 2-bit selector $S[1:0]$ to control the operation as per the defined truth table.
- To demonstrate the principle of 2's complement subtraction using a unified adder/subtractor circuit.
- To successfully test and validate all four operations on the physical hardware prototype.

5 System Design / Methodology

The design is modular, based on replicating a "1-bit ALU slice" 16 times. Each 1-bit slice computes one bit of the final result.

5.1 Overall Architecture

The system is built from four main parallel computation units that perform all operations simultaneously. The 2-bit select lines $S[1:0]$ are used at the very end to pick the desired result.

For each of the 16 bits (from $i = 0$ to $i = 15$), the circuit performs:

1. A_i AND B_i

2. A_i OR B_i
3. A_i + B_i (as part of a 16-bit adder)
4. A_i - B_i (as part of the same 16-bit adder)

The four results for each bit are fed into a 4-to-1 Multiplexer (MUX), as shown in Figure 1. We use the 74HC153, which is a Dual 4-to-1 MUX. This means one 74HC153 chip can handle 2 bits of our ALU.

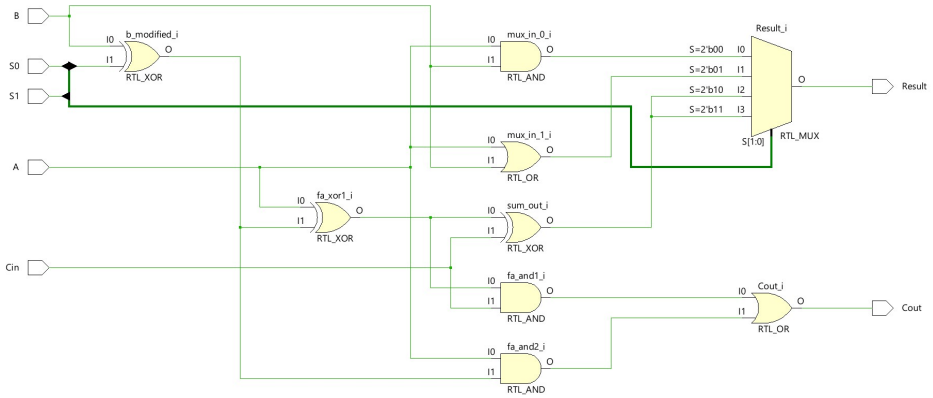

Figure 1: 1-Bit ALU Slice.

The 2-bit selector $S[1:0]$ is a global bus connected to all 16 MUXes in parallel. This ensures all 16 bits of the output are selected from the same operation.

5.2 Functional Units

5.2.1 Logic Unit (for I_0 and I_1)

This is the most straightforward part of the design.

- **16-bit AND:** 16 2-input AND gates (74HC08) take A_i and B_i as input. The 16 results are wired to the I_0 inputs of their respective MUXes.
- **16-bit OR:** 16 2-input OR gates (74HC32) take A_i and B_i as input. The 16 results are wired to the I_1 inputs of their respective MUXes.

5.2.2 Arithmetic Unit (for I_2 and I_3)

This is the most complex and elegant part of the ALU. A single 16-bit Ripple-Carry Adder is used to perform **both** addition and subtraction. This is achieved by using the $S[0]$ selector line as a control signal.

The design relies on 2's complement arithmetic, where $A-B$ is equivalent to $A+(\overline{B})+1$.

- **16-bit Ripple-Carry Adder:** This is built from 16 1-bit Full Adders chained together. The C_{out} of bit i becomes the C_{in} of bit $i+1$.
- **Controlled Inverter (B-Input):** Before B is fed to the adder, each B_i bit passes through an XOR gate (74HC86). The other input to this XOR gate is the $S[0]$ selector line.
- **The "$+1$" Control (Carry-In):** The C_{in} of the very first Full Adder (bit 0) is wired directly to the $S[0]$ selector line.

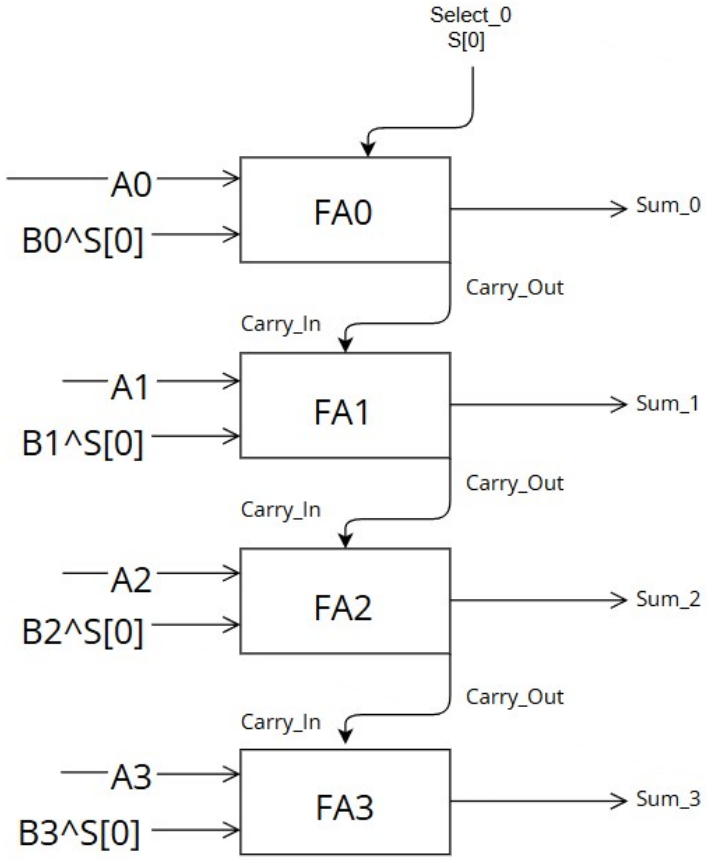

Figure 2: 4-Bit Ripple Carry Adder.

5.2.3 The "Magic" of the $S[0]$ Signal

This single wire performs two different jobs depending on the operation:

Case 1: ADD (S[1:0] = 10)

- $S[1] = 1$ tells the MUX to select an arithmetic result (either I_2 or I_3).
- $S[0] = 0$ is fed to the XOR gates. The XOR gates calculate $B_i \oplus 0$, which just equals B_i. The B operand passes through unchanged.
- $S[0] = 0$ is fed to the first C_{in}. The adder calculates with an initial carry of 0.
- The final operation is: $A + B + 0$. This is standard addition. The result is wired to the I_2 inputs of the MUXes.

Case 2: SUBTRACT (S[1:0] = 11)

- $S[1] = 1$ tells the MUX to select an arithmetic result.
- $S[0] = 1$ is fed to the XOR gates. The XOR gates calculate $B_i \oplus 1$, which equals $\overline{B_i}$ (NOT B). The B operand is inverted.
- $S[0] = 1$ is fed to the first C_{in}. The adder calculates with an initial carry of 1.

- The final operation is: $A+\overline{B}+1$. This is the exact formula for 2's complement subtraction. The **same** result from the **same** adder is wired to the I_3 inputs of the MUXes.

5.3 Final Output Selection

5.3.1 Operation Truth Table

The 74HC153 MUX uses $S[1]$ as the "B" selector and $S[0]$ as the "A" selector. This gives the following truth table, which maps our logic to the chip's pins.

Table 1: ALU Operation Selection Truth Table

S[1]	S[0]	MUX Input	Operation
0	0	I_0	A AND B
0	1	I_1	A OR B
1	0	I_2	A + B (Add)
1	1	I_3	A - B (Subtract)

5.3.2 Hardware Bill of Materials (Approximate)

- **74HC08 (Quad 2-in AND):** 32 (for Adder, and we are taking bitwise input to MUX from Adder only) = 32 gates / 4 per chip = 8 chips
- **74HC32 (Quad 2-in OR):** 16 (for OR) + 16 (for Adder) = 32 gates / 4 per chip = 8 chips
- **74HC86 (Quad 2-in XOR):** 16 (B-invert) + 32 (for Adder) = 48 gates / 4 per chip = 12 chips
- **74HC153 (Dual 4-to-1 MUX):** 16 bits / 2 per chip = 8 chips

Total (Estimated): $\approx$ 8 (AND) + 8 (OR) + 12 (XOR) + 8 (MUX) = **36 ICs**.

6 Implementation / Experimental Setup

The project was implemented on 8-bus breadboards, as the large number of ICs and wires requires significant space.

6.1 Power Distribution

With $\approx$40 ICs, power distribution is critical. Dedicated Vcc (5V) and GND rails were run across all breadboards. Each **every single IC** received its own 0.1μF ceramic decoupling capacitor, placed as close as possible to its Vcc and GND pins. This is essential to filter high-frequency switching noise and prevent "ground bounce" which can cause logic errors.

6.2 Construction and Testing

The build process was modular:

1. **1-bit Full Adder:** A single 1-bit Full Adder was built and its truth table was verified with LEDs.

2. **4-bit Ripple-Carry Adder:** Four adders were chained. Carry propagation was tested.
3. **4-bit Adder/Subtractor:** The $S[0]$-controlled XOR gates and C_{in} were added. All 4 arithmetic combinations (ADD/SUB) were tested.
4. **4-bit Logic Unit:** The AND and OR gates were added and tested.
5. **4-bit ALU Slice:** The MUXes were added and the full 4-bit ALU was tested.
6. **Scaling to 16-bits:** Three more 4-bit ALU modules were built and connected, paying careful attention to the carry-chain, which now "ripples" across all 16 bits.

7 Results and Discussion

The 16-bit ALU was tested by setting the $S[1:0]$ selectors and providing known inputs for A and B via DIP switches. The 16-bit result was read via LEDs. All four operations were validated.

7.1 Logic Operation Tests (AND/OR)

These tests were trivial and verified that the logic gates and multiplexers were wired correctly.

Test 1: A AND B

- **A** = `1010 1010 1010 1010`
- **B** = `0000 1111 0000 1111`
- **S[1:0]** = 00 (AND)
- **Expected:** `0000 1010 0000 1010`
- **Result:** `0000 1010 0000 1010` (Correct)

Test 2: A OR B

- **A** = `1010 1010 1010 1010`
- **B** = `0000 1111 0000 1111`
- **S[1:0]** = 01 (OR)
- **Expected:** `1010 1111 1011 1111`
- **Result:** `1010 1111 1011 1111` (Correct)

7.2 Arithmetic Operation Tests (ADD/SUB)

These tests are more complex as they involve 2's complement interpretation.

Test 1: Addition (100 + 25)

- **A** = 100 = `0000 0000 0110 0100`
- **B** = 25 = `0000 0000 0001 1001`
- **S[1:0]** = 10 (Add)
- **Expected** = 125 = `0000 0000 0111 1101`
- **Result:** `0000 0000 0111 1101` (Correct)
- **Flags:** $C_{out} = 0$, $N = 0$

Test 2: Subtraction (100 - 25)

- **A** = 100 = `0000 0000 0110 0100`
- **B** = 25 = `0000 0000 0001 1001`
- **S[1:0]** = 11 (Subtract)
- **Internally:** The ALU calculates $A + (\overline{B}) + 1$
  - $\overline{B}$ = `1111 1111 1110 0110`
  - C_{in} = 1
  - $A + \overline{B} + 1$ = (`0000 0000 0110 0100`) + (`1111 1111 1110 0110`) + 1
- **Expected** = 75 = `0000 0000 0100 1011`
- **Result:** `0000 0000 0100 1011` (Correct)
- **Flags:** $C_{out} = 1$, $N = 0$

Test 3: Subtraction with Negative Result (25 - 100)

- **A** = 25 = `0000 0000 0001 1001`
- **B** = 100 = `0000 0000 0110 0100`
- **S[1:0]** = 11 (Subtract)
- **Internally:** The ALU calculates $A + (\overline{B}) + 1$
  - $\overline{B}$ = `1111 1111 1001 1011`
  - C_{in} = 1
  - $A + \overline{B} + 1$ = (`0000 0000 0001 1001`) + (`1111 1111 1001 1011`) + 1
- **Expected** = -75. The 2's complement for -75 is `1111 1111 1011 0101`.
- **Result:** `1111 1111 1011 0101` (Correct)
- **Flags:** $C_{out} = 0$, $N = 1$

7.2.1 Interpreting the Result (2's Complement)

The ALU hardware does not "know" about negative numbers. It simply performs the binary addition. It is our job to interpret the result, using the Negative (N) and Carry (C) flags.

When to Read Directly vs. When to Convert

1. **Check the Negative Flag (N):** The N flag is just the last output bit, $Result[15]$.
2. **If N = 0 (Sign bit is 0):** The result is positive. The binary output is the final answer. The value `0000 0000 0100 1011` is read directly as +75.
3. **If N = 1 (Sign bit is 1):** The result is negative and is in 2's complement format. To find its magnitude, we must perform the 2's complement conversion manually:
   - **Result:** `1111 1111 1011 0101`
   - **Invert all bits:** `0000 0000 0100 1010`
   - **Add 1:** `0000 0000 0100 1011`
   - This is $64 + 8 + 2 + 1 = 75$. Therefore, the result is **-75**.

7.2.2 Interpreting the Carry Flag (C_{out})

The final C_{out} from the 16-bit adder is one of the most important status flags, but its meaning is **inverted** for subtraction.

- **For Addition (S[0]=0):**
  - $C_{out} = 1$: An unsigned overflow occurred. The sum was too large to fit in 16 bits (e.g., 65000 + 1000).
  - $C_{out} = 0$: The sum fit within 16 bits.
- **For Subtraction (S[0]=1):**
  - $C_{out} = 1$: **No Borrow** was required. This means $A \geq B$ (e.g., $100 - 25$).
  - $C_{out} = 0$: A **Borrow** was required. This means $A < B$ (e.g., $25 - 100$).

Our test cases (Test 2 and 3) confirm this behavior.

8 Conclusion and Future Scope

This project successfully demonstrated the design and hardware implementation of a 16-bit, 4-function ALU using only discrete 74HC-series logic ICs. The design was validated through component-level and system-level testing.

The key learning from this project was the practical application of 2's complement arithmetic. The elegance of using a single adder circuit, controlled by the $S[0]$ selector, to perform both addition and subtraction was the most critical design takeaway. The project also highlighted the physical challenges of building a large-scale digital circuit, particularly the importance of meticulous wiring, power distribution, and noise decoupling.

The primary limitation of this design is the speed. A 16-bit ripple-carry adder is slow, as the carry bit must propagate through all 16 full adders. Future enhancements could include:

- **Lookahead-Carry Adder:** Implementing a carry-lookahead circuit to compute the carry bits in parallel, drastically increasing the speed.
- **More Functions:** Adding more logic functions (like NOT, NAND, XNOR) or arithmetic functions (like increment, decrement, or bit-shifting).
- **Integration:** This ALU could become the core of a "Homebrew CPU" project, to be combined with registers, a control unit, and memory.

9 Verilog Implementation

The following Verilog code provides a structural model of the ALU, matching the hardware implementation.

9.1 1-Bit ALU Slice (Structural Verilog)

This module structurally describes the 1-bit ALU slice. It explicitly defines the logic for the four parallel units and the final 4-to-1 MUX, matching the 74HC-series ICs used. The `case` statement is used to model the 74HC153 MUX directly.

```
/**
 * Verilog code for a 1-bit ALU slice.
 * This module directly corresponds to the hardware design using
 * discrete logic gates and a 4-to-1 MUX.
 *
 * Inputs:
 * A    : 1-bit data input from Operand A
 * B    : 1-bit data input from Operand B
 * Cin  : 1-bit Carry-In from the previous slice
 * S1   : Selector bit 1 (Top-level MUX control)
 * S0   : Selector bit 0 (Low-level MUX control and Arithmetic control)
 *
 * Outputs:
 * Result : 1-bit result of the selected operation
 * Cout   : 1-bit Carry-Out to the next slice
 *
 * Operation Table (S[1:0]):
 * 00 : A AND B
 * 01 : A OR B
 * 10 : A + B + Cin (Add)
 * 11 : A - B (Subtract, via A + (!B) + 1)
 */
module ALU_1bit_slice (
    input  wire A,
    input  wire B,
    input  wire Cin,
    input  wire S1,
    input  wire S0,
    output reg  Result, // Changed to 'reg' for use in always block
    output wire Cout
);

    // --- Internal Wires ---

    // Wires for the 4 MUX inputs
    wire mux_in_0; // (S[1:0] = 00)
    wire mux_in_1; // (S[1:0] = 01)
    wire mux_in_2; // (S[1:0] = 10)
    wire mux_in_3; // (S[1:0] = 11)

    // Wires for the unified Arithmetic unit
```

```
wire b_modified;
wire sum_out;

// Wires for full-adder internal logic (for clarity)
wire fa_xor1;
wire fa_and1;
wire fa_and2;


// --- 1. Logic Unit (for MUX inputs I_0 and I_1) ---

// MUX Input 0: A AND B
assign mux_in_0 = A & B;

// MUX Input 1: A OR B
assign mux_in_1 = A | B;


// --- 2. Unified Arithmetic Unit (for MUX inputs I_2 and I_3) ---

// This is the clever part: S0 controls the B operand.
// if S0=0 (Add),  b_modified = B ^ 0 = B
// if S0=1 (Sub),  b_modified = B ^ 1 = NOT B
assign b_modified = B ^ S0;

// Full Adder Logic (built from gates)
assign fa_xor1 = A ^ b_modified;
assign sum_out = fa_xor1 ^ Cin;       // Sum output of the adder

assign fa_and1 = fa_xor1 & Cin;
assign fa_and2 = A & b_modified;
assign Cout    = fa_and1 | fa_and2;   // Cout is the final Carry-Out
                                      // (This is independent of the MUX)

// The adder's sum output is fed to *both* I_2 and I_3 of the MUX
assign mux_in_2 = sum_out;
assign mux_in_3 = sum_out;


// --- 3. Final 4-to-1 MUX ---

// This always block with a case statement explicitly models a 4-to-1 MUX,
// which is a clearer structural representation of the 74HC153 hardware.
// Combinational logic, so we use @(*)
always @(*) begin
    case ({S1, S0})
        2'b00:   Result = mux_in_0; // AND
        2'b01:   Result = mux_in_1; // OR
        2'b10:   Result = mux_in_2; // ADD
        2'b11:   Result = mux_in_3; // SUB (which is sum_out, same as ADD)
        default: Result = 1'bx;     // Default to unknown to catch errors
```

```
        endcase
    end

endmodule
```

9.2 16-Bit ALU Top Module (Structural Verilog)

This top-level module instantiates the `ALU_1bit_slice` 16 times using a `generate` block. It connects the ripple-carry chain by wiring the C_{out} of one slice to the C_{in} of the next. It also correctly implements the 2's complement logic by wiring the main $S[0]$ control line to the very first carry-in ($carries[0]$).

```
/**
 * Verilog code for the full 16-bit ALU.
 * This module instantiates the 1-bit slice 16 times and
 * wires the ripple-carry chain.
 *
 * It correctly implements the S[0] connection to the first Cin
 * for 2's complement subtraction.
 */
module ALU_16bit_top (
    input  wire [15:0] A,
    input  wire [15:0] B,
    input  wire        S1,
    input  wire        S0,
    output wire [15:0] Result,
    output wire        Cout_final // Final Carry-Out from bit 15
);

    // Internal wire array for the ripple-carry chain
    // `carries[0]` will be the first carry-in
    // `carries[16]` will be the final carry-out
    wire [16:0] carries;

    // This is the key to subtraction:
    // The very first carry-in (carries[0]) is just the S0 line.
    // if S0=0 (Add),  Cin for bit 0 is 0.
    // if S0=1 (Sub),  Cin for bit 0 is 1. (This is the "+ 1")
    assign carries[0] = S0;

    // This 'generate' block creates 16 copies of our 1-bit slice.
    // It's like a for-loop for hardware.
    genvar i;
    generate
        for (i = 0; i < 16; i = i + 1) begin: alu_bit_slices

            // Instantiate one 1-bit slice
            ALU_1bit_slice slice (
                .A      (A[i]),             // Input A for this bit
                .B      (B[i]),             // Input B for this bit
                .Cin    (carries[i]),       // Carry-In from previous slice
                .S1     (S1),               // Global S1
```

```
                .S0     (S0),               // Global S0
                .Result (Result[i]),        // Result output for this bit
                .Cout   (carries[i+1])      // Carry-Out to next slice
            );

        end
    endgenerate

    // Assign the final carry-out (from the 16th slice) to the top-level output
    assign Cout_final = carries[16];

endmodule
```

References

[1] T. L. Floyd, *Digital Fundamentals*, 11th ed. Pearson, 2015.

[2] J. F. Wakerly, *Digital Design: Principles and Practices*, 5th ed. Pearson, 2018.

[3] M. M. Mano and M. D. Ciletti, *Digital Design: With an Introduction to the Verilog HDL, VHDL, and SystemVerilog*, 6th ed. Pearson, 2018.

[4] S. R. Sarangi, *Computer Organisation and Architecture*. McGraw Hill, 2015.

[5] Texas Instruments, *SN74HC08 Quadruple 2-Input Positive-AND Gates Datasheet*, 2015. [Online]. Available: `https://www.ti.com/lit/ds/symlink/sn74hc08.pdf`

[6] Texas Instruments, *SN74HC32 Quadruple 2-Input Positive-OR Gates Datasheet*, 2015. [Online]. Available: `https://www.ti.com/lit/ds/symlink/sn74hc32.pdf`

[7] Texas Instruments, *SN74HC86 Quadruple 2-Input Exclusive-OR Gates Datasheet*, 2015. [Online]. Available: `https://www.ti.com/lit/ds/symlink/sn74hc86.pdf`

[8] Texas Instruments, *SN74HC153 Dual 4-Line to 1-Line Data Selectors/Multiplexers Datasheet*, 2015. [Online]. Available: `https://www.ti.com/lit/ds/symlink/sn74hc153.pdf`

Appendix A: Schematics

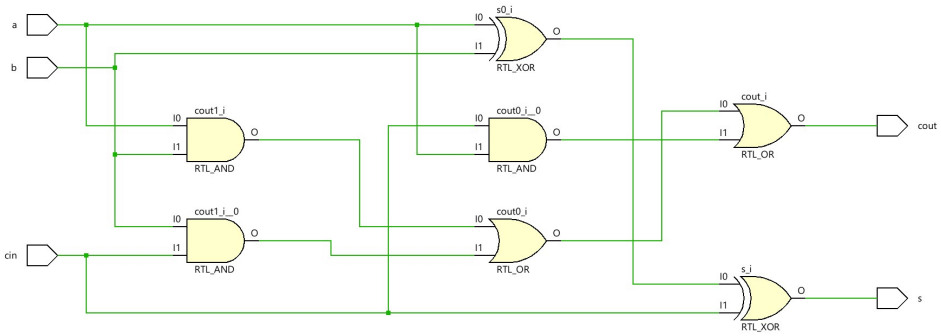

Figure 3: 1-Bit Full Adder Logic Diagram

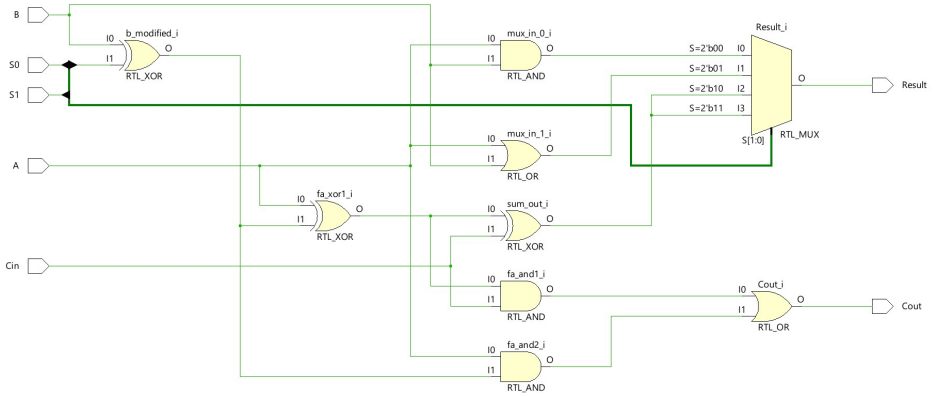

Figure 4: 1-Bit ALU Slice Schematic

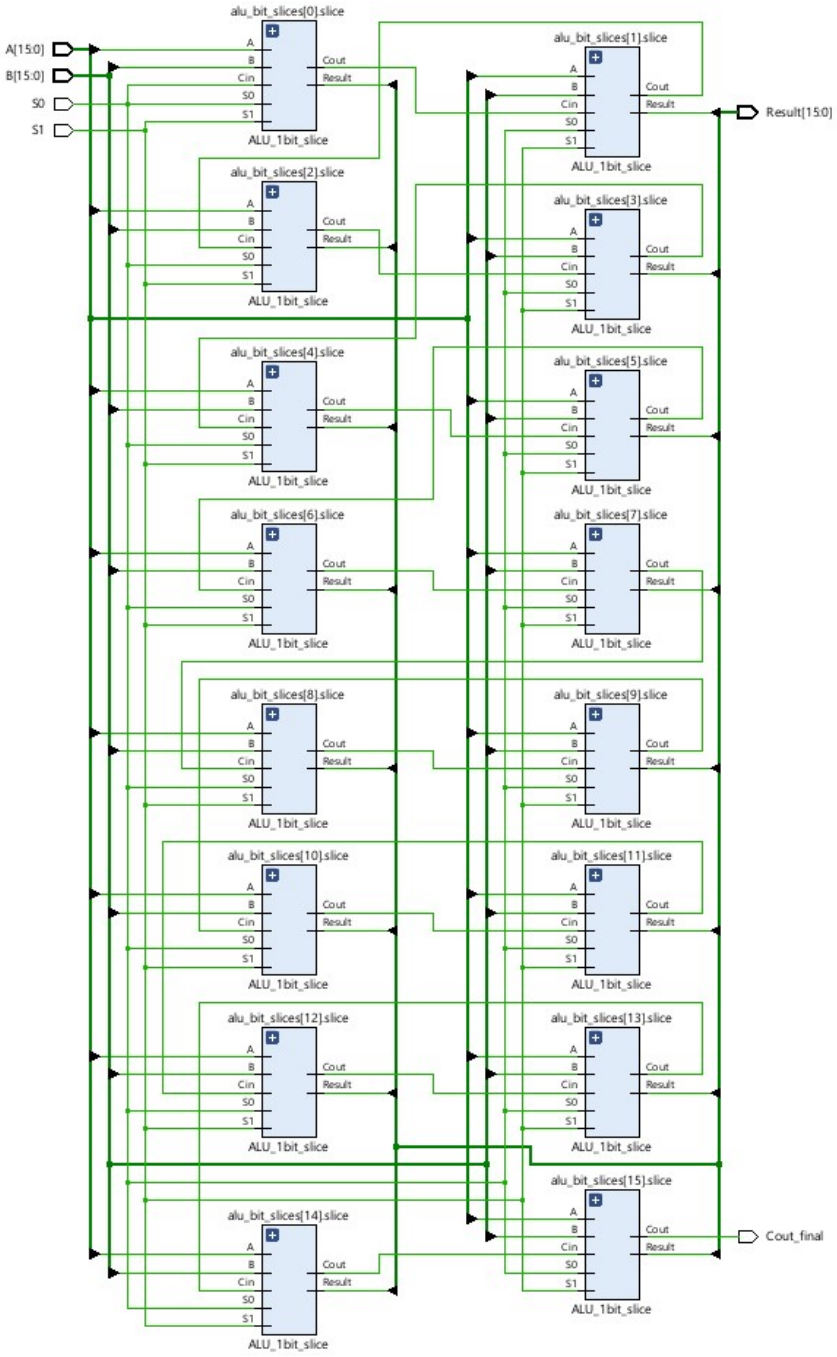

Figure 5: 16-Bit ALU Schematic

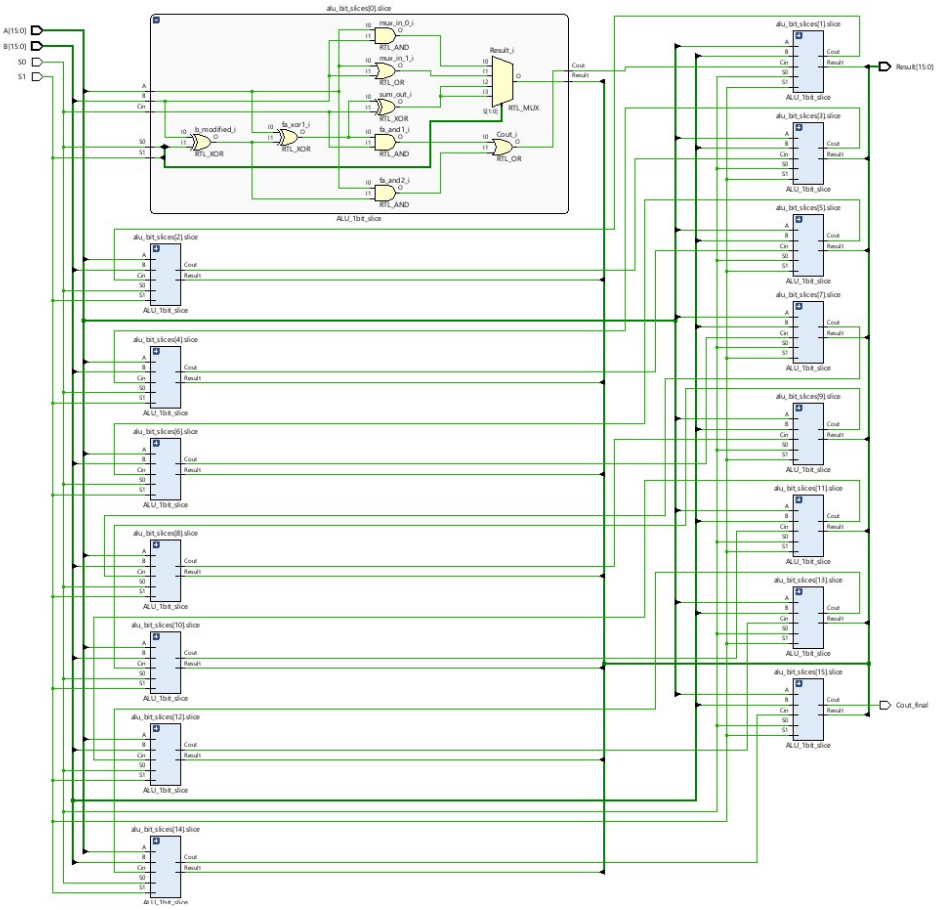

Figure 6: 1-Bit Elaborated: 16-Bit ALU Schematic

Appendix B: Implementation Photos

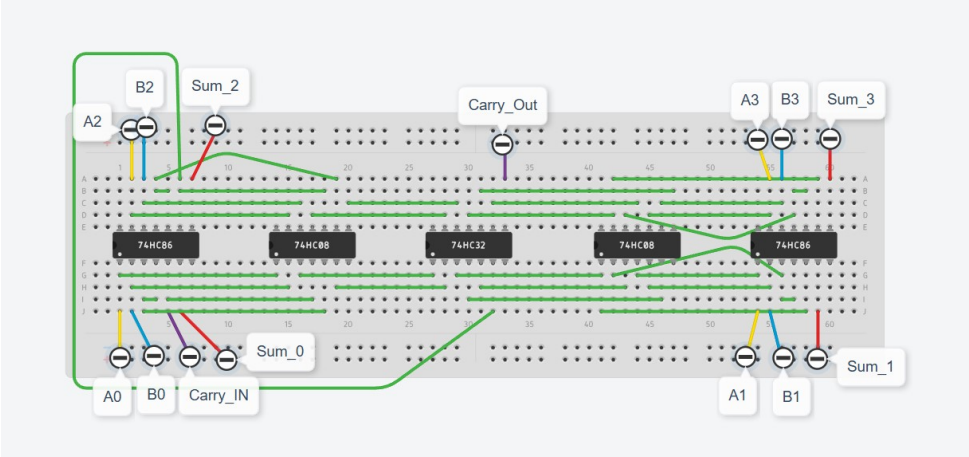

Figure 7: Wiring Diagram for 4-Bit Adder

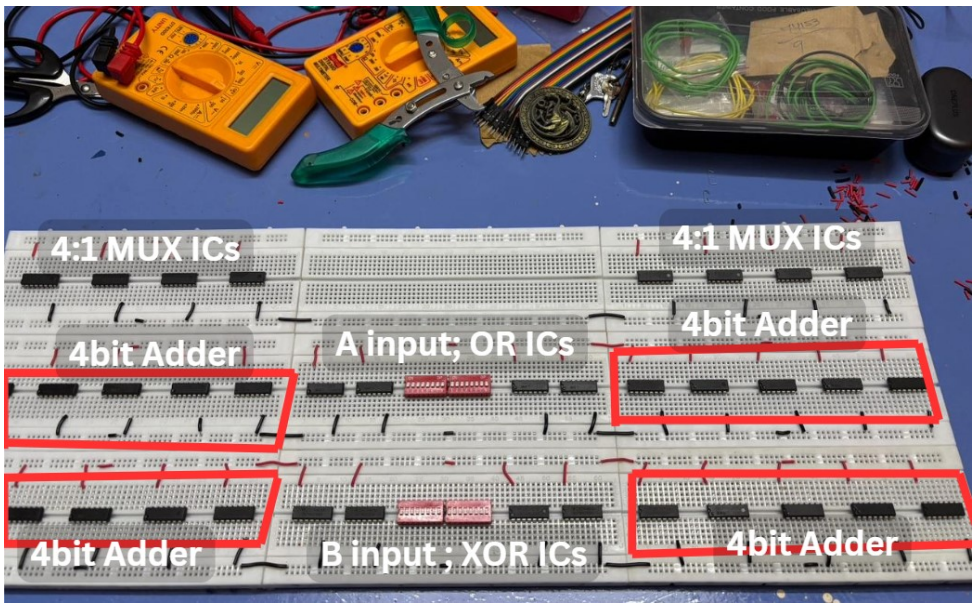

Figure 8: IC placement

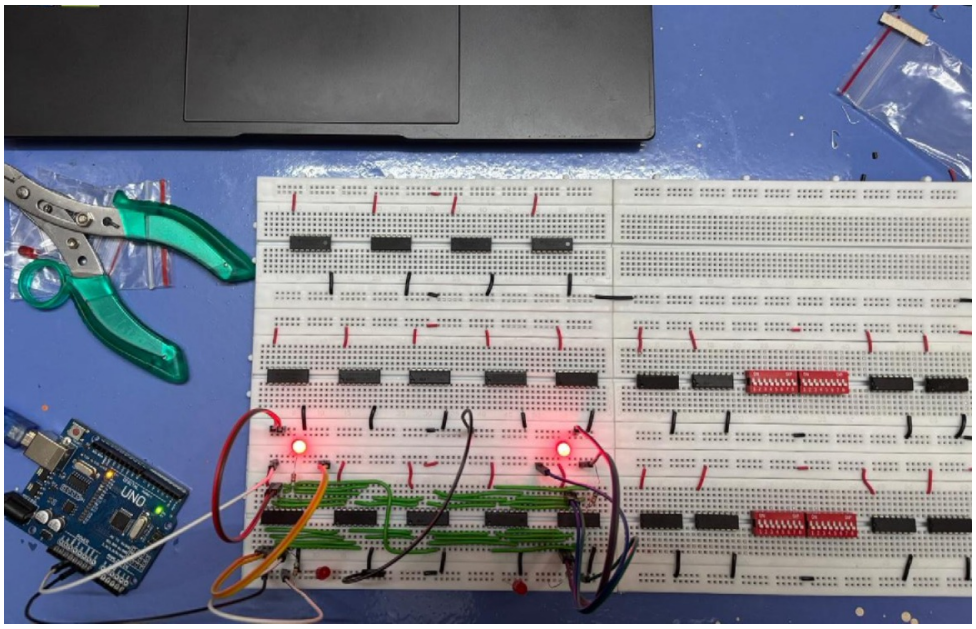

Figure 9: 4-Bit ALU Prototyping and Testing

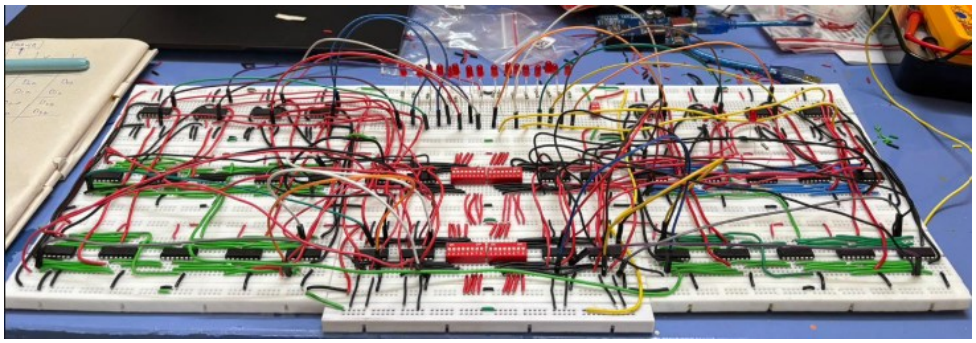

Figure 10: Final Project