ALU

INTRODUCTION

This project involved designing and implementing a flexible Arithmetic Logic Unit (ALU) using Verilog, with a default 8-bit configuration and support for multiple operations. The ALU is parameterized, meaning it can be easily scaled to different bit-widths depending on application needs. The design supports two main categories of functions: arithmetic/comparison and logical/rotation operations, selected using a mode control signal (MODE). Each operation within these categories is triggered through a dedicated 4-bit command input (CMD).

In the arithmetic/comparison mode (MODE = 0), the ALU handles operations such as addition, subtraction, signed arithmetic, multiplication, and value comparisons (greater, equal, less). In logical/rotation mode (MODE = 1), it supports common bitwise functions like AND, OR, XOR, NOT, along with shift and rotate operations. The ALU also generates key status flags—carry out (COUT), signed overflow (OFLOW), and comparison results (G, E, L)—along with an error flag (ERR) for unsupported operations.

Control signals such as clock (CLK), reset (RST), and chip enable (CE) are included to ensure proper operation within a synchronous system. The entire design is modular, synthesizable, and suitable for implementation on hardware platforms like FPGAs or ASICs, making it relevant for embedded processors, digital signal processing applications, and custom logic blocks.

For validation, a detailed testbench was developed. It reads test cases from a stimulus file and uses modular components like drivers, monitors, and scoreboards to verify each operation. Special cases, such as signed overflows and invalid commands, were thoroughly tested to ensure reliability and correctness.

OBJECTIVES

The main goal of this project was to build a reliable and flexible ALU (Arithmetic Logic Unit) using Verilog, with the ability to handle a broad range of operations. The design was intended to be clean, scalable, and suitable for both simulation and hardware implementation. Below are the key objectives that guided the project:

- **To create a working ALU that is parameterized**—so that its bit-width can be adjusted based on the application requirements. The default setup was 8-bit, but the design is easily extendable.
- **To include a wide set of supported operations**, such as:
    - Standard arithmetic like addition and subtraction, including versions with carry-in and carry-out.
    - Signed operations that correctly handle overflow and flag updates.
    - Bitwise logic operations like AND, OR, XOR, and NOT.
    - Shift instructions (left/right), as well as rotate instructions for circular bit movements.
    - A few additional operations like conditional multiplication, depending on control inputs.
    - Comparison features that set output flags based on whether the input values are greater than, equal to, or less than each other.
- **To add a mode-selection logic** using a simple multiplexer setup, allowing the ALU to switch between arithmetic/comparison and logic/rotation operations based on a single control signal.
- **To thoroughly test and verify the ALU using simulations**, ensuring that every function performs as expected. This included viewing waveforms and checking all corner cases like overflow, invalid operations, etc.
- **To design the ALU in a modular way**, so that features like pipelining or formal verification can be added later without rewriting the whole system.

ARCHITECTURE

The internal structure of the ALU is organized into several functional blocks, each responsible for a specific part of the data flow and control logic. The overall design ensures synchronous operation and clean separation between arithmetic and logical functionality. Below is a breakdown of each block and its role in the system:

- **Input Interface**

  This block is responsible for receiving all the necessary inputs to the ALU. These include the two operands (opa, opb), control signals like clock (clk), reset (rst), operation mode (mode), command (cmd), carry-in (cin), input valid (inp_valid), and chip enable (ce). It ensures that all inputs are correctly received and ready for processing.

- **Internal Register Block**

  This section uses edge-triggered flip-flops to latch the input values. By synchronizing the incoming data with the system clock, it ensures stable operation and avoids glitches or timing issues in downstream blocks.

- **Mode Selector (MUX)**

  A simple 2:1 multiplexer is used here to decide which processing block should be active. Based on the mode signal, it routes the input either to the Arithmetic & Comparator Block or to the Logic & Rotate Block. This allows a single ALU unit to handle a wide variety of instructions while keeping the hardware efficient.

- **Arithmetic & Comparator Block** *(active when mode = 1)*

  This block performs all arithmetic functions, both signed and unsigned. It handles basic operations like addition and subtraction, along with increment and decrement. It also includes comparison logic to evaluate whether one operand is greater than, less than, or equal to the other.

- **Logic & Rotate Block** *(active when mode = 0)*

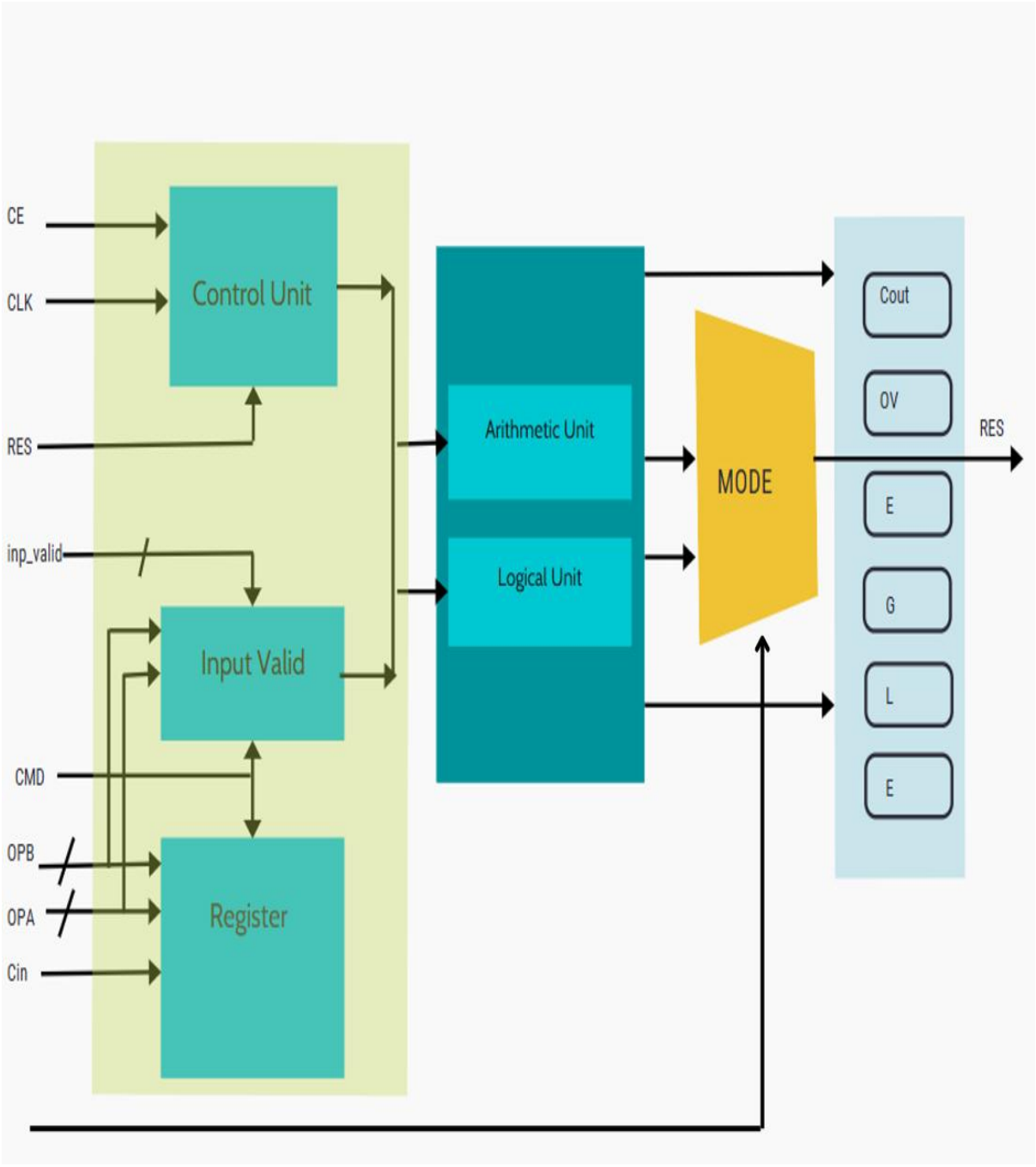
This unit takes care of all logic operations such as AND, OR, XOR, and NOT. In addition, it supports bitwise shifts and circular rotations. It includes basic error handling to flag unsupported or illegal operations.

- **Flag Logic Block**

After each operation, this block generates the necessary status flags. These include carry-out (carry_out), overflow (overflow), comparison results (greater, less, equal), and a general-purpose error flag (error) to signal invalid conditions.

- **Output Interface**

The final result (res) and all relevant flags are stored in output registers on the rising edge of the clock. This ensures that outputs are stable and synchronized before being passed on to any subsequent processing stage.



WORKING

The behavior of the ALU is directed by a combination of input operands, control signals, and a command input (cmd) that determines the operation. The design is largely **combinational**, which means that most operations are completed based solely on the current inputs, without depending on sequential logic—apart from a few special cases like multiplication, which are pipelined for stability.

Input Capture and Synchronization

At every positive clock edge, the ALU captures all incoming data and control signals—namely opa, opb, cmd, mode, ce, cin, and inp_valid. These inputs are stored in internal registers to align them with the system clock. This approach eliminates timing uncertainties such as metastability and ensures clean data handoff to the internal processing blocks.

Mode Selection

A dedicated 2:1 multiplexer uses the mode signal to select the operation path.

- If mode = 1, the Arithmetic & Comparator block is activated.
- If mode = 0, the Logic & Rotate block is selected.

This switching mechanism ensures that each operation is routed only to its relevant logic block, improving clarity and resource efficiency.

Command Decoding and Execution

Inside the selected block, the 4-bit command (cmd) is decoded to determine the exact operation to perform, such as ADD, SUB, AND, or ROL.

- For arithmetic commands, the operands are processed with signed or unsigned arithmetic logic as per the selected instruction.
- Logical commands apply bitwise operations directly to the inputs.
- Rotation commands use bits from opb to determine the rotation amount. Error detection logic flags invalid scenarios—like non-zero upper bits (opb[7:4] ≠ 0), which are not allowed for rotation instructions.

Combinational Execution

With the exception of multiplication-related commands, all operations are purely combinational. This means:

- The result is generated immediately based on the present inputs.
- There's no delay introduced by flip-flops or internal state machines.
- It ensures faster response and a simplified design path for typical instructions.

For operations involving multiplication (mult, mult1), the result is stabilized over two clock cycles using intermediate registers (temp1, temp2) to maintain timing accuracy.

Flag Generation

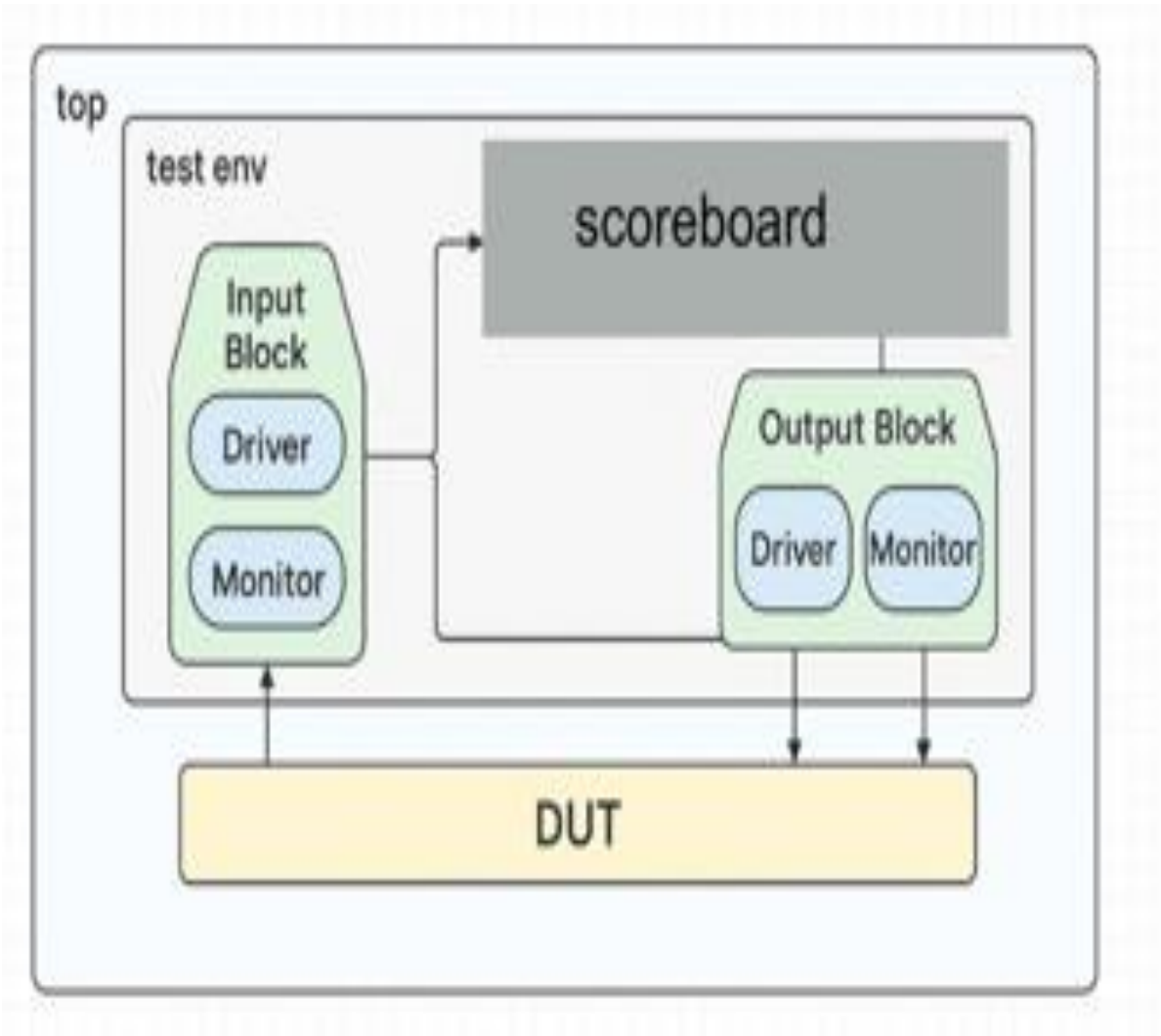
Once an operation is executed, the ALU evaluates and updates multiple status flags:

- cout: Set if there is a carry-out from addition/subtraction.
- oflow: Indicates signed overflow.
- g, l, e: Comparison flags representing "greater than," "less than," and "equal to," respectively.
- err: Raised when invalid inputs are detected—especially in rotation operations where upper bits of opb must be zero.

Output Registration

Finally, the operation result (res) and all the generated status flags are latched at the next clock edge. This ensures that outputs are synchronized with the system timing and remain stable until the next operation cycle. The output is then made available to the external logic or testbench for further use or verification.

TESTBENCH ARCHITECTURE



1. Top-Level Module (top)

The top module integrates all verification components with the DUT. It instantiates the **test environment** and connects it to the DUT's interfaces. This layer controls simulation startup and termination.

2. Test Environment (test env)

The test_env encapsulates all reusable components necessary for driving inputs, monitoring outputs, and validating DUT behavior. It consists of:

a. Input Block

- **Driver**: Applies stimulus to the DUT input ports. In Verilog, this is typically done using procedural blocks (initial or always) or tasks that generate signal-level transactions.
- **Monitor**: Observes and captures the actual inputs driven to the DUT. This allows for sending reference inputs to the scoreboard for result checking.

b. Output Block

- **Driver**: Rarely used in the output path unless a feedback mechanism is required.
- **Monitor**: Observes the DUT's output signals and converts them into transaction-level data for analysis. These observed outputs are sent to the scoreboard.

c. Scoreboard

The scoreboard is a functional checker. It compares the DUT's actual outputs (received from the Output Monitor) with the expected outputs (based on Input Monitor data and DUT behavior specification). Any mismatches are logged, enabling result analysis.

3. Design Under Test (DUT)

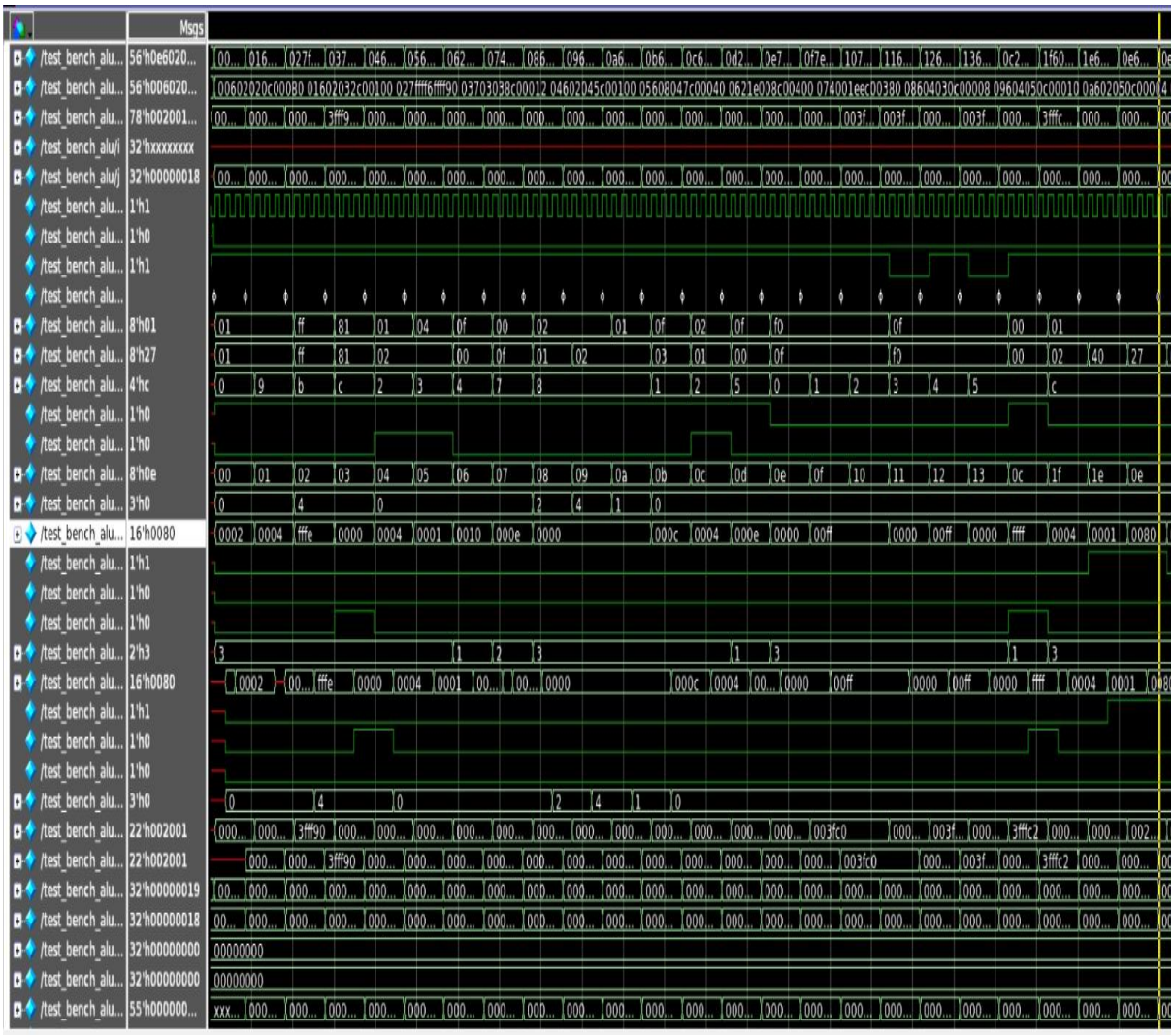
The DUT is the Verilog module being verified. It receives stimulus from the testbench and produces output responses which are then validated. This module is instantiated within the top-level design and interacts directly with the input/output blocks of the test environment.

RESULT

To test the ALU design, a Verilog testbench was created that provided different sets of inputs covering all types of supported operations in both arithmetic and logical modes. Based on the simulation, the following observations were made:

- **Correct Outputs:** All operations—like add, sub, multiply, AND, OR, shift, rotate, etc.—gave correct results as expected. Signed and unsigned instructions also behaved properly across different inputs.
- **Flag Generation:** Status flags such as carry (cout), overflow (oflow), greater (g), less (l), equal (e), and error (err) were checked for each instruction. The flags responded correctly, especially during edge cases like overflow or comparisons.
- **Waveform Analysis:** The simulation waveform showed that data was properly latched on each clock cycle, and the mode switching (between arithmetic and logical) worked smoothly. Operations like multiplication showed a slight delay (2 cycles), which was expected.

- **Error Handling:** When inputs were intentionally given wrong (like rotation with invalid bits), the ALU flagged the error correctly. This shows the design can detect



Questa Coverage Report

Number of tests run:	1
Passed:	1
Warning:	0
Error:	0
Fatal:	0

List of tests included in report...

List of global attributes included in report...

List of Design Units included in report...

Coverage Summary by Structure:

Design Scope	Hits %	Coverage %
test_bench_alu	81.10%	89.47%
read_stimulus	100.00%	100.00%
driver	100.00%	100.00%
dut_reset	100.00%	100.00%
global_init	100.00%	100.00%
monitor	100.00%	100.00%
score_board	100.00%	100.00%
gen_report	100.00%	100.00%
inst_dut	91.21%	91.80%

Coverage Summary by Type:

Total Coverage:					81.10%	89.47%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	185	184	1	1	99.45%	99.45%
Branches	111	102	9	1	91.89%	91.89%
FEC Expressions	4	4	0	1	100.00%	100.00%
FEC Conditions	24	19	5	1	79.16%	79.16%
Toggles	1084	833	251	1	76.84%	76.84%

Report generated by Questa (ver. 10.6c) on Sat 07 Jun 2025 07:24:38 PM IST with command line:
vcover report -html coverage_name.ucdb -htmldir covReport -details

CONCLUSION

The Verilog-based ALU design successfully fulfills all specified functional requirements. It supports a wide range of operations, such as arithmetic, logical, comparison, shift, and rotate instructions, organized under two operational modes controlled by a mode-select multiplexer. The design's modular architecture enhances readability, maintainability, and scalability. Comprehensive simulation and waveform analysis confirmed correct operation outputs and precise status flag generation across all tested cases, including edge conditions and error scenarios. This validates the ALU's reliability and readiness for integration into larger digital systems.

FUTURE IMPROVEMENT

- **Pipelining**

  Introducing pipelined stages will increase throughput by enabling multiple operations to run concurrently, thereby improving overall speed.

- **Extended Instruction Set**

  Adding more complex operations such as division, modulus, and advanced bitwise functions will expand the ALU's capabilities.

- **Wider Data Width Support**

  Extending support and testing for 32-bit and 64-bit operands will allow the ALU to be used in higher-bit processors and applications.

- **Formal Verification**

  Applying formal verification methods like property checking will ensure design correctness beyond simulation, especially for corner cases and edge conditions.

Implementing these enhancements will make the ALU more efficient, scalable, and robust, suitable for integration into larger digital systems or custom processors.