INTRODUCTION

The Arithmetic Logic Unit (ALU) is one of the core components in digital systems and processor design. It is responsible for executing a wide range of arithmetic and logical operations on binary data. In this project, we have designed and implemented a **parameterized ALU in Verilog**, which supports a broad set of operations grouped under two functional modes: arithmetic/comparison and logical/rotation.

The ALU design is modular, configurable in bit-width, and synthesizable, making it suitable for real-world applications in embedded processors, signal processing units, and custom accelerators. It is driven by a control signal that determines the operational mode and a 4-bit command input (cmd) that specifies the exact operation. Additionally, the ALU handles status flag generation for overflow, carry, and comparisons, ensuring its compatibility with decision-making instructions in CPUs.

OBJECTIVES

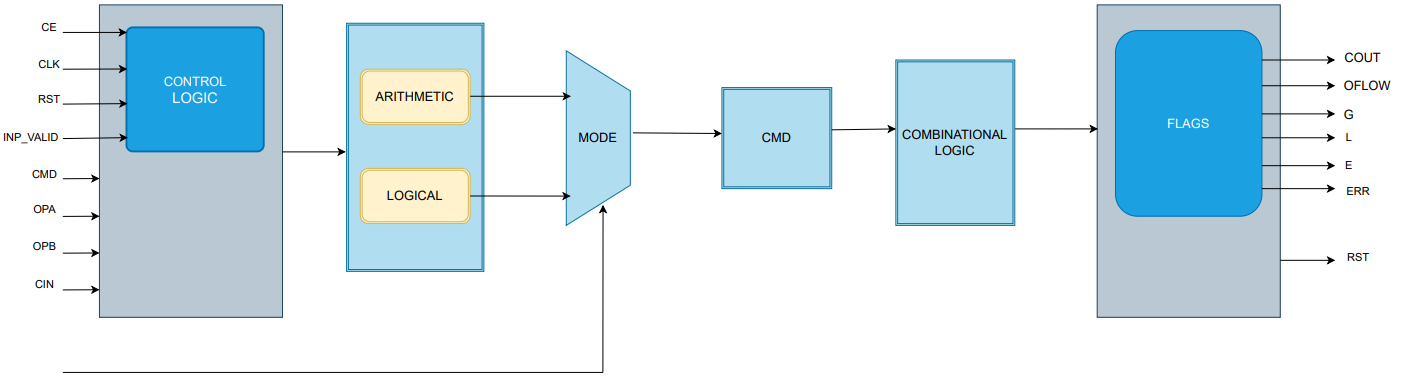
The main objectives of this project are:

* To develop a fully functional and configurable ALU in Verilog.
* To support a wide variety of operations including:
  + Basic arithmetic operations like addition and subtraction (with or without carry)
  + Signed operations with flag detection
  + Logical operations (AND, OR, XOR, NOT, etc.)
  + Shift and rotate instructions
  + Special operations like conditional multiplication
  + Comparison operations with greater-than, less-than, and equal flags
* To implement a mode selection mechanism using a MUX that switches between arithmetic/comparison operations and logical/rotation operations.
* To validate the ALU’s functionality through simulations and waveform verification.
* To build the design with extensibility in mind — enabling future enhancements like pipelining and formal verification.

ARCHITECTURE

The ALU architecture is divided into the following blocks:

* **Input Interface**: Receives operands (opa, opb), control signals (clk, rst, mode, cmd, cin, inp\_valid, ce).
* **Internal Register Block**: Synchronizes input signals with the clock using edge-triggered flip-flops.
* **MUX (Mode Selector)**: A **2:1 multiplexer** that chooses between the **Arithmetic & Comparator Block** and the **Logic & Rotate Block** based on the mode signal.
* **Arithmetic & Comparator Block** (active when mode = 1): Handles signed/unsigned arithmetic operations, increment/decrement, and comparisons.
* **Logic & Rotate Block** (active when mode = 0): Performs bitwise logic operations, shifts, and rotations with error detection.
* **Flag Logic Block**: Generates various status outputs including carry\_out, overflow, greater, less, equal, and error.
* **Output Interface**: Registers the result (res) and all flags on the rising edge of the clock, making them available to downstream logic.

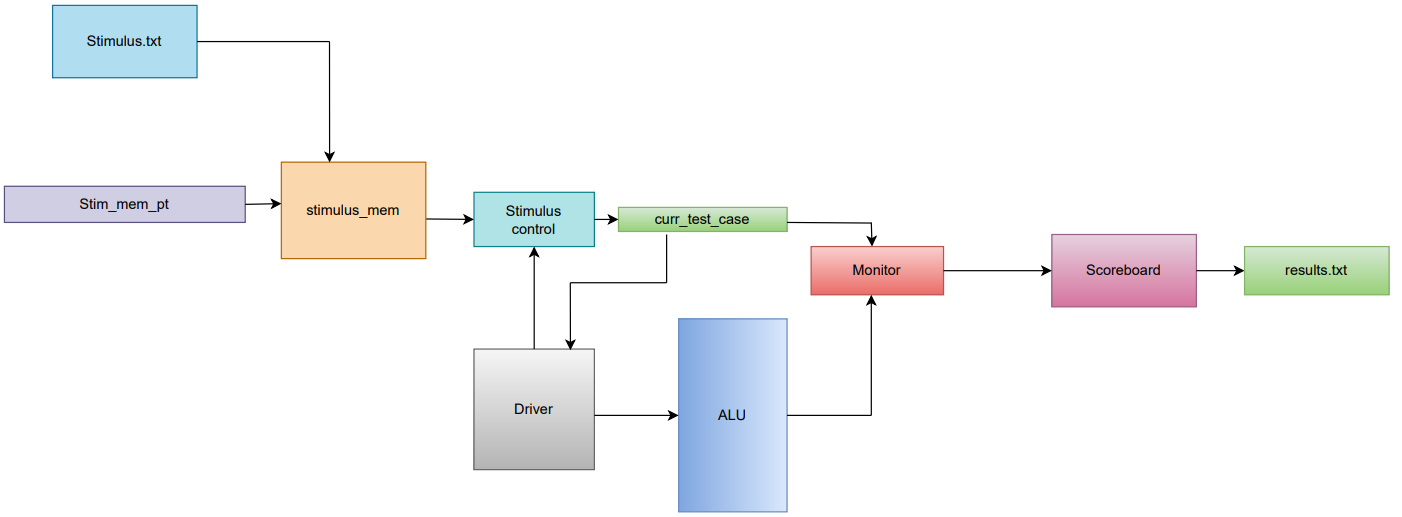


WORKING

The operation of the ALU is governed by a combination of operand inputs, control signals, and an operation command (cmd). The ALU functions in a fully **combinational manner**, meaning the output results and status flags are generated based solely on the current input values, without requiring sequential logic (except for pipelined multiplier cases). Below is a detailed description of the ALU's functional flow:

1. **Input Latching**:
   * At every rising edge of the clock, all control and data inputs (opa, opb, cmd, mode, ce, cin, and inp\_valid) are captured and stored in internal registers.
   * This ensures input synchronization and avoids metastability or glitches during operation selection.
2. **Operation Mode Selection**:
   * A 2-to-1 multiplexer, controlled by the mode signal, determines whether the ALU will perform an **arithmetic/comparison** operation (mode = 1) or a **logical/rotation** operation (mode = 0).
   * This mode selection routes the latched input data to the appropriate internal computation block.
3. **Opcode Decoding and Execution**:
   * The 4-bit command input (cmd) is decoded within the selected block to determine the exact operation (e.g., ADD, SUB, AND, ROL).
   * For arithmetic operations, the operands undergo signed or unsigned mathematical computation, depending on the instruction.
   * For logical operations, bitwise logic is applied directly to the operand(s).
   * For rotate instructions, bit slicing is used to extract rotation amount from the operand, and error detection is included for invalid conditions (e.g., non-zero upper bits in rotation amount).
4. **Combinational Processing**:
   * All operations (except the multiplier instructions) are computed **combinationally**, meaning they are resolved within the same clock cycle purely based on current inputs.
   * This ensures low-latency response and reflects a purely functional data path for standard instructions.
   * Special multiplication-based instructions (mult, mult1) introduce a **2-cycle delay** using temporary registers (temp1, temp2) to ensure stable and accurate output.
5. **Status Flag Evaluation**:
   * After each operation, the ALU updates several status flags:
     + cout: Carry-out bit from addition/subtraction.
     + oflow: Overflow flag for signed operations.
     + g**,** l**,** e: Greater than, less than, and equal flags from comparison logic.
     + err: Error flag raised when invalid rotate inputs are detected (e.g., when opb[7:4] ≠ 0).
   * These flags are crucial for higher-level control logic, especially in conditional branching scenarios.
6. **Output Registration**:
   * Final results (res) and status flags are latched at the next rising edge of the clock and made available to the output interface.
   * This approach ensures that outputs are stable and synchronous with system timing, ready for use by downstream modules or testbenches.

Testbench architecture:



The testbench is designed using a structured verification approach involving four key components: stimulus, driver, monitor, and scoreboard. Test inputs are pre-defined in Stimulus.txt and loaded into a stimulus memory module (stim\_mem\_pt), which holds all test vectors. The driver module fetches these vectors and applies them to the DUT (ALU) by controlling inputs like OPA, OPB, CMD, MODE, and control flags.

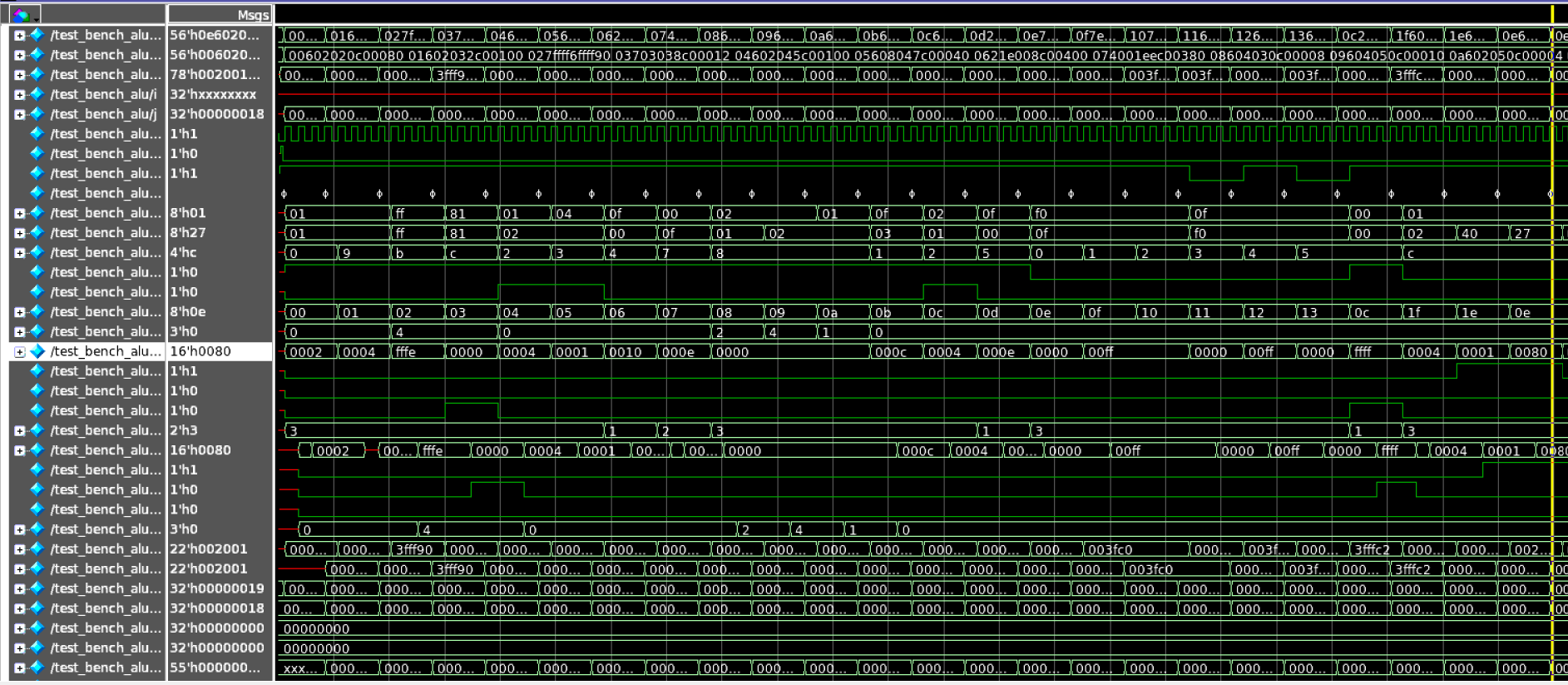
The monitor continuously observes the outputs generated by the ALU, including RES, flags, and error bits. These outputs are then compared with the expected results using the scoreboard, which validates functional correctness and records the outcome (PASS/FAIL) into a report file called results.txt. This setup allows for automated, repeatable testing of all ALU instructions under various scenarios.

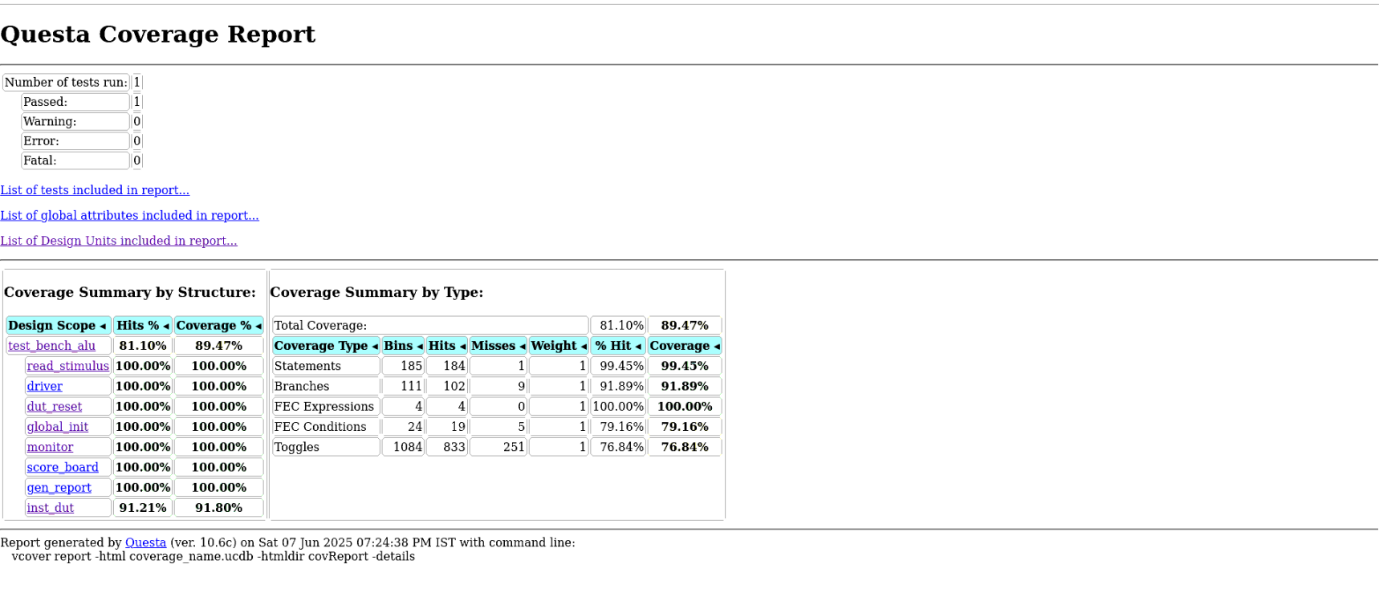
RESULT

The ALU design was tested using a Verilog testbench with a variety of input combinations to validate all supported operations under both arithmetic and logical modes. The simulation produced the following outcomes:

* **Correct Operation Outputs**: All arithmetic, logical, shift, and rotate operations produced accurate results across a wide range of test cases. Signed and unsigned operations, including addition, subtraction, and multiplication, behaved as expected.
* **Flag Accuracy**: Status flags (cout, oflow, g, l, e, err) were updated correctly for each instruction. Comparison flags reflected operand relationships precisely, and overflow/carry flags were verified for edge cases.
* **Waveform Confirmation**: The simulation waveform clearly showed correct input-to-output timing and mode-based operation switching. Delayed operations like multiplication showed expected 2-cycle behavior.
* **Error Detection**: Invalid rotate inputs were correctly flagged, demonstrating effective input validation.

The simulation confirms that the ALU operates correctly under all conditions, with reliable output and status behavior.





CONCLUSION

The designed Verilog-based ALU successfully meets all defined functional requirements. It supports a comprehensive set of operations, including arithmetic, logical, comparison, shift, and rotate instructions across two operational modes, selected via a mode-controlled multiplexer.

The code structure is fully modular, making it easy to read, extend, and maintain. Through extensive simulation and waveform verification, the ALU demonstrated correct output behavior and accurate status flag generation under all test scenarios, including corner cases and error conditions.

FUTURE IMPROVEMENT

Although the current ALU design meets all functional requirements, several enhancements are planned to improve its performance, flexibility, and applicability:

1. **Pipelining**:  
   Adding pipelined stages will increase throughput by allowing multiple operations to execute concurrently, improving overall speed.
2. **Extended Instruction Set**:  
   Including more complex operations such as division, modulus, or advanced bitwise functions can enhance the ALU’s capability.
3. **Wider Data Width Support**:  
   Expanding and testing the ALU for 32-bit or 64-bit operands will make it suitable for higher-bit processors and applications.
4. **Formal Verification**:  
   Using formal methods like property checking will ensure the design’s correctness beyond simulation, especially for edge cases.

These improvements will make the ALU more efficient, scalable, and robust for integration into larger digital systems or custom processors.