DESIGN AND VERIFICATION OF ARITHMETIC & LOGIC UNIT

Tejas Poojary(6119)

Introduction:

The Arithmetic Logic Unit (ALU) is a core component of any processing system, performing both arithmetic and logical operations on input data. This project focuses on designing a flexible and parameterized ALU in Verilog HDL, capable of handling a wide variety of instructions including signed and unsigned arithmetic, logical operations, shift and rotate instructions, and multiplication (enabled through MUL define).

The design supports dynamic operand sizes via parameters and ensures correct output flag signaling for overflow, carry, error detection, and comparison results. The ALU is designed to be compatible with pipelined architectures by introducing input latching for normal operations and conditional delays, especially for multiplication operations.

Objectives:

1) To Design a Synthesizable Verilog ALU

- The ALU (Arithmetic Logic Unit) is designed in a way that it can be synthesized efficiently on FPGA or ASIC platforms. This means:Used only synthesizable Verilog constructs (avoided delays, initial blocks for logic, etc.).

- Used parameterization where it is possible to allow flexibility in data width and feature inclusion.

2) To Implement Both Signed and Unsigned Operations.

- The ALU handles arithmetic operations correctly for both signed and unsigned inputs. This involves:Supporting addition, subtraction, for both signed and unsigned integers using the signed keyword in Verilog wherever required.
- Properly interpreting the inputs and output data depending on the mode or command signal that specifies whether the operation is signed or unsigned.
- Ensuring overflow detection and correct flag updates for signed/unsigned cases.

3) To Support Logical and Bit Manipulation Operations like AND,OR, XOR, etc.

- In addition to arithmetic, the ALU supports a variety of bitwise logical operations:Bitwise AND, OR, XOR, NAND, NOR, XNOR.
- Bit manipulation operations like bit shifts (logical left/right), rotate operations are also taken care.

4) To Support Dynamic Instruction Decoding via CMD and Mode Handling.

- The ALU's behavior is fully controlled by an opcode (CMD) and a mode input.
- The opcode defines the specific operation to perform (e.g., ADD, SUB, AND, OR).

- The decoding logic is flexible and easy to extend for future instructions or modes.

5) To Introduce Multiplication Using a Macro Definition.

- Implement multiplication that can be conditionally compiled into the design using a macro (`ifdef MUL).
- This allows the ALU to exclude the potentially resource-heavy multiplier from synthesis if multiplication is not needed, saving area and power.

6) To Include Comparison Flags Like Greater-Than (G), Less-Than (L),and Equal-To (E).

- Besides generating a result, the ALU outputs status flags to help control flow and decision-making in the CPU:
- Implement flags based on both signed and unsigned comparisons depending on mode.

7) To Simulate and Verify the Functionality of the ALU with Testbenches.

- Developed detailed testbench to simulate all supported operations under a variety of conditions.
- Used test packets to validate arithmetic, logic, and comparison operations, including edge cases (overflow, carry-out,error).
- Tested both signed and unsigned modes thoroughly.

Architecture:

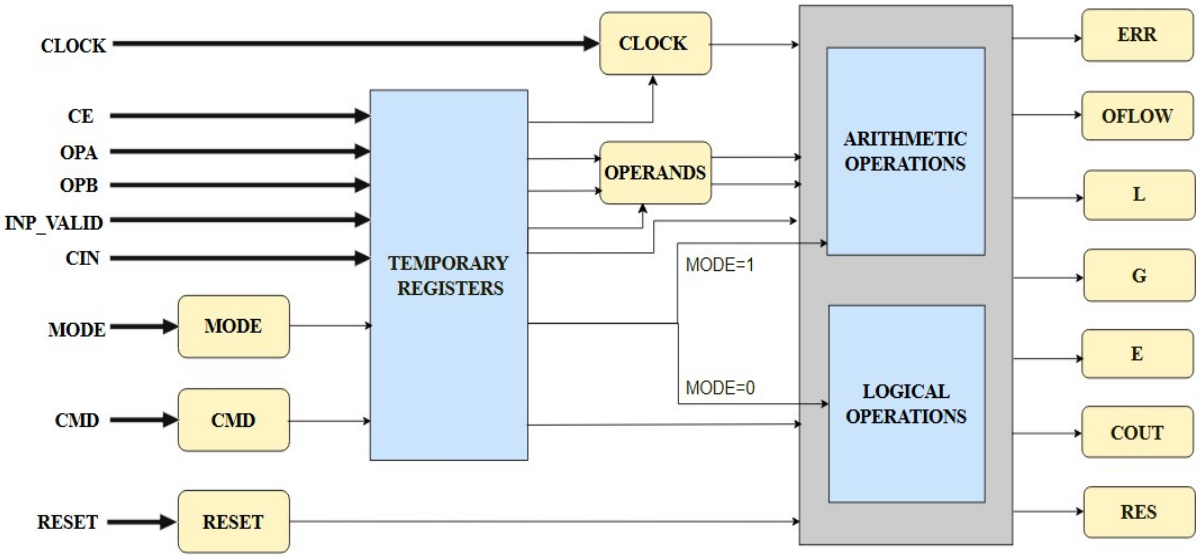

Figure 1: Design Architecture

The ALU has a modular architecture consisting of:

Inputs:

OPA, OPB: Operands.

CIN: Carry-in for certain arithmetic operations.

CMD: Operation command.

MODE: Mode selection (arithmetic/logical).

INP_VALID: Indicates the valid operand source.

CE: Clock Enable.

CLK, RST: Clock and Reset.

Registers:

Temporary storage for inputs (temp_a,temp_b, etc.) to synchronize with the clock.

Delay register (mul_delay) to introduce pipelining for multiplication.

Outputs:

RES: Result of the operation (width is n+1 for arithmetic or 2*n for multiplication).

COUT, OFLOW: Carry and overflow detection.

ERR: Flag for errors such as invalid shift amounts.

G, L, E: Comparison result flags.

The ALU supports operation modes, where MODE=1 enables arithmetic operations and MODE=0 enables logical operations. A 2-bit INP_VALID signal helps in deciding which input (A/B/both) is valid, simplifying control logic for unary operations like NOT_A.

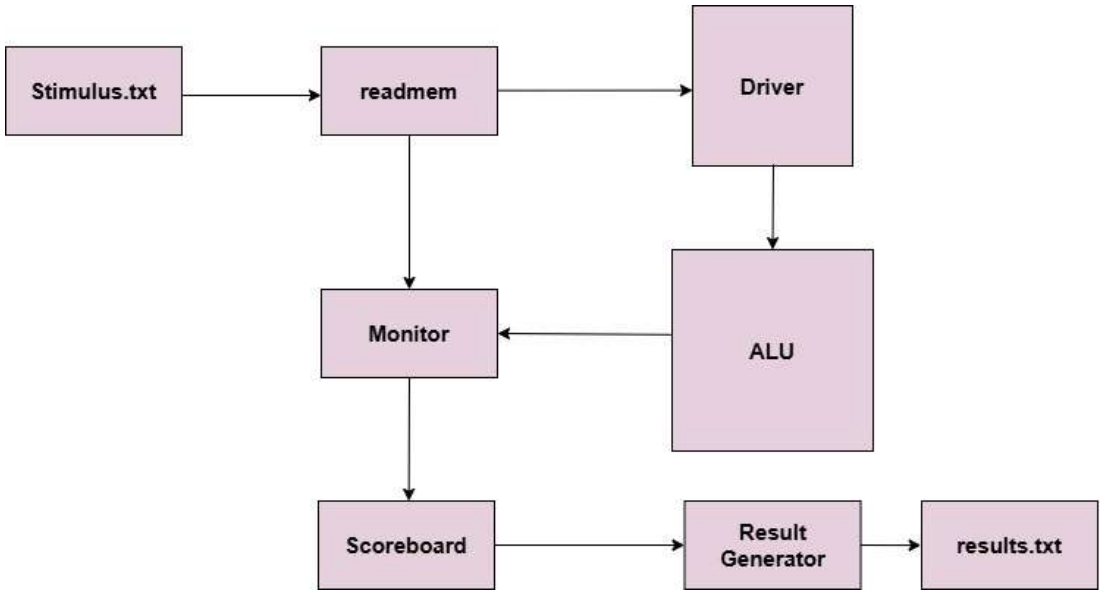

Figure 2: Testbench Architecture

The testbench architecture in Figure 2 verifies the ALU's functionality in a modular way. It begins with Stimulus.txt, which holds predefined test cases including operands, opcodes, and

control signals. These vectors are loaded into a memory array or queue via a readmem block. The data feeds two main components: the Driver, which applies inputs to the ALU in sync with clock and enable signals, and the Monitor, which observes ALU outputs (e.g., result, carry, overflow, zero) while logging the applied inputs. The Monitor sends this to the Scoreboard, which compares actual outputs with expected or golden values, determining pass/fail status. Finally, the Result Generator formats these results and writes a summary report to results.txt.

Working:

The parameterized ALU module is designed to execute a broad range of arithmetic and logical functions on input operands of bit width N. It accepts two main operands (OPA and OPB), a carry-in (CIN), a 4-bit command signal (CMD), a mode control signal (MODE), input validity indicators (INP_VALID), and standard clocking and control inputs (CLK, RST, CE). On the rising edge of the clock, when the clock enable (CE) is asserted, all input signals are latched into internal registers. If reset (RST) is active, the ALU clears all internal state elements—registers, flags, and counters—to zero, effectively initializing the system.

The ALU functions in two separate modes, determined by the MODE input. When MODE is high, it enters arithmetic operation mode. The exact behaviour in this mode is driven by the CMD and INP_VALID signals. When both operands are valid (INP_VALID = 2'b11), the ALU can carry out typical unsigned operations like addition and subtraction, along with extended capabilities such as addition with carry, subtraction with borrow, and signed arithmetic

(with proper handling of overflow, negative, and zero flags). It also supports two types of multiplication: (OPA + 1) * (OPB + 1) and (OPA << 1) * OPB. These multiplication operations are handled through a three-cycle pipeline mechanism.Moreover, the ALU includes comparison functions that set greater (G), less (L), or equal (E) flags based on operand comparisons.

If only one operand is valid (INP_VALID = 2'b10 for OPA or 2'b01 for OPB), the ALU executes single-operand actions like increment and decrement.

When MODE is low, the ALU transitions to logical operation mode. With both inputs valid, it performs logical functions such as AND, OR, XOR, NAND, NOR, and XNOR. Additionally, it offers bitwise rotation operations (ROL and ROR), where the value of OPB determines by how many positions OPA should be rotated. If the rotation count exceeds the operand width N, the ALU sets the error flag. When only one input is valid, it handles unary logical functions like bitwise NOT, as well as single-bit shift operations (left or right), all with boundary checks and validation.

For each operation, relevant output flags are activated to indicate status. The result (RES) is extended to a width of 2N+1 to support wide operations like multiplication using `ifdef. For signed/unsigned arithmetic, the ALU uses COUT and OFLOW flags.The ERR flag is used to report invalid commands or mismatches between operand validity and the type of operation.This setup ensures a robust, synthesizable, and verification-friendly ALU design, making it well-suited for integration into larger digital subsystems.

Results:

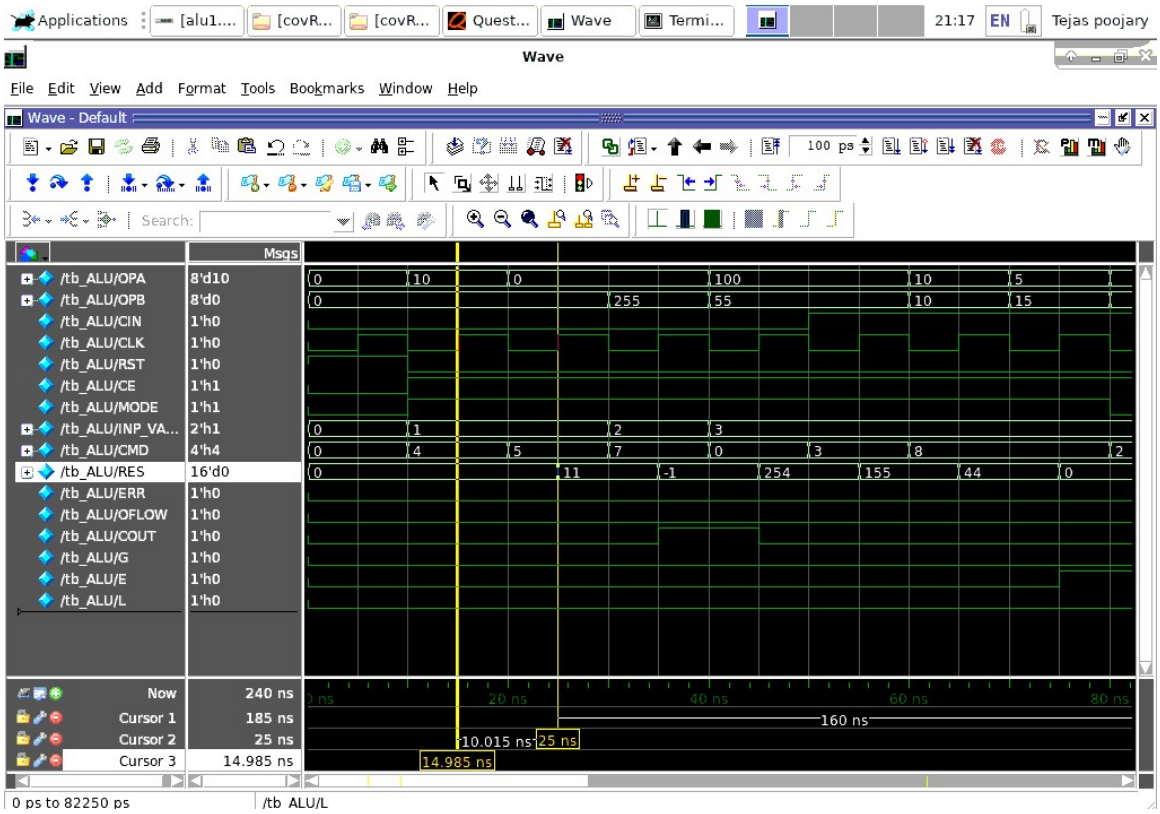

Figure 3:Normal operations

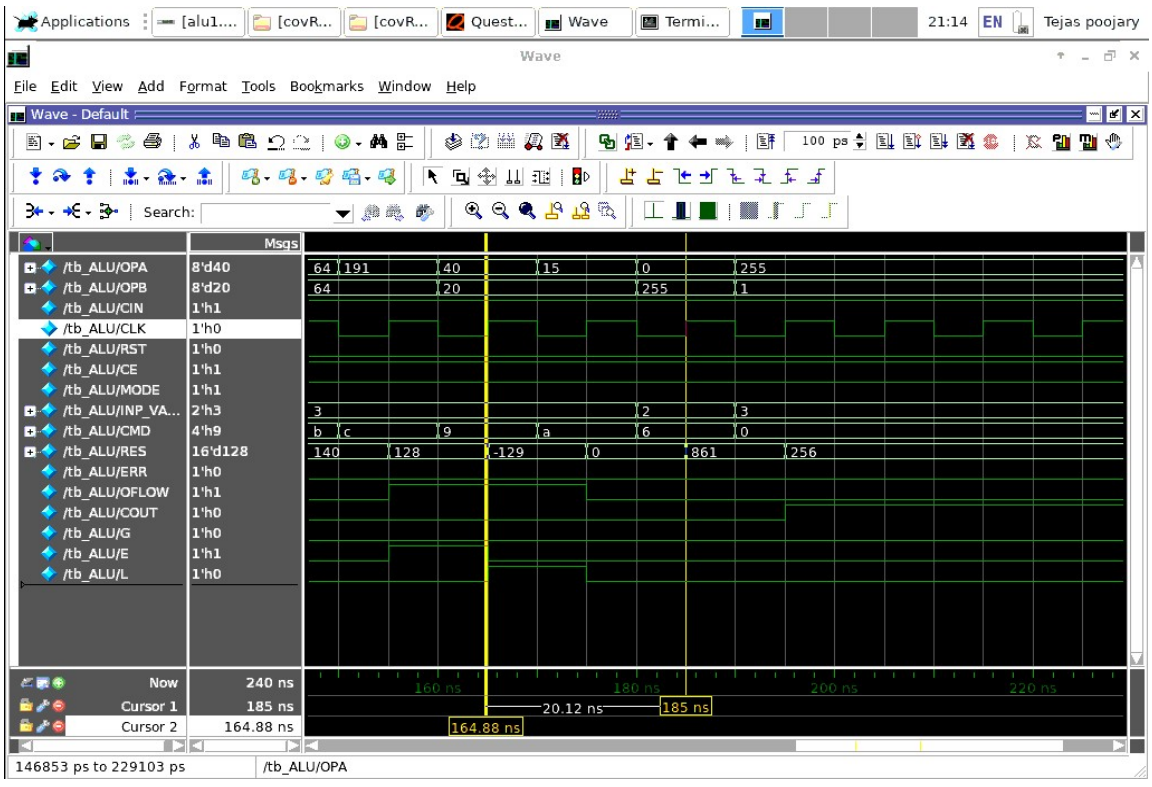

Figure 4:Multiplication operation

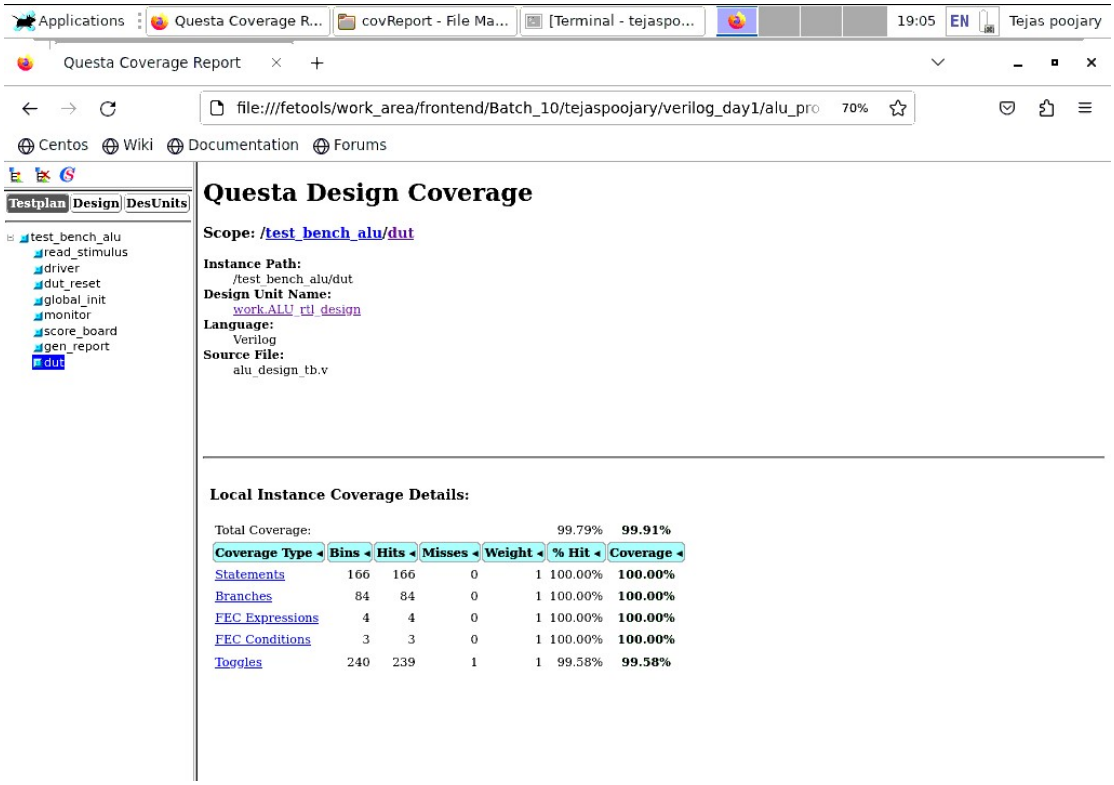

Figure 5:Coverage Report

Conclusion:

In conclusion, the ALU was successfully implemented in Verilog and verified through simulation. It supports a comprehensive range of operations, from simple arithmetic and logical instructions to more advanced shift, rotate, and comparison functions. The design includes detailed flag generation and error detection, making it suitable for integration into more complex systems such as processors or controllers. The code is modular and parameterized, ensuring reusability and ease of expansion. Simulation results confirm the functional correctness of the design and validate that all requirements were met.

Future Improvements:

Introduce pipelining to break the ALU into multiple stages (fetch, decode, execute, write-back), enabling higher throughput and making it suitable for use in CPUs and high-performance systems.

To make the ALU better and more efficient, we can improve its design by using pipeline execution, which helps it work faster and with less delay. We can also add more features like support for floating-point numbers, complex number calculations, and cryptographic operations to handle more advanced tasks.With these changes, the ALU can become a stronger, faster, and smarter unit that works well for advanced digital applications while still keeping high accuracy and performance.