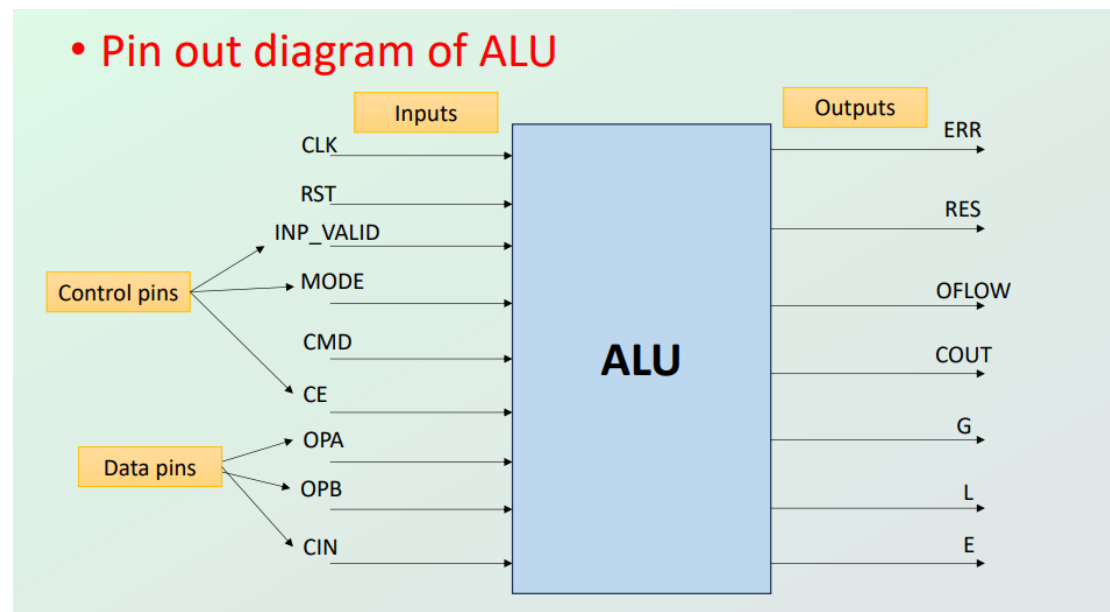# INTRODUCTION:-

This project presents the design and implementation of an Arithmetic Logic Unit (ALU) using Verilog HDL. The ALU is capable of performing a variety of arithmetic and logical operations, including multiplication and flag generation, making it suitable for general-purpose computational units in digital systems.

It processes two input operands along with control and mode signals to generate results and status flags.The design includes input buffering, control decoding, and separate arithmetic and logic execution blocks.Status flags such as carry, overflow, error, and comparator outputs (G, E, L) are generated.The ALU operates synchronously with a clock and includes reset and enable logic.An input validation signal ensures correct timing and sequencing of operations.Dedicated modules handle multiplication and logic functions separately for modularity.The design is synthesizable and verified through simulation and waveform analysis.This ALU can be integrated into larger digital processing systems or CPUs.

OBJECTIVES:-

The primary objective of this ALU project is to design a modular, efficient, and synthesizable Arithmetic Logic Unit using Verilog HDL. The ALU should be capable of performing a broad range of arithmetic and logical operations, including addition, subtraction, multiplication, AND, OR, and XOR. It must take input operands along with control signals and produce accurate results with appropriate status flags. The design should maintain synchronous operation with clock and reset support. The implementation should follow industry-standard practices in digital design, ensuring it can be synthesized for FPGA or ASIC platforms.

Another major goal is to incorporate comprehensive input control and validation mechanisms. Control signals such as MODE, CMD, and CE, along with the INP_VALID signal, should govern the data flow and functionality selection within the ALU. Buffer stages will be used to synchronize and hold input operands for predictable operation sequencing. Status flags including carry (COUT), overflow (OFLOW), greater-than (G), equal-to (E), less-than (L), and error (ERR) should be generated for each operation as required. This ensures that the ALU can be reliably used in complex systems requiring result interpretation and branching logic.

The final objective is to validate the correctness and robustness of the design through simulation. Thorough testbenches must be developed to verify each operation, control path, and flag output. Simulations should generate waveforms that clearly illustrate the functional behavior of the ALU under various input scenarios. The results should confirm that all modules interact correctly and the outputs match the expected values. Furthermore, the design should be prepared for future enhancements such as pipelining, expanded instruction sets, and formal verification to extend its usability in more advanced applications.

ARCHITECTURE:-
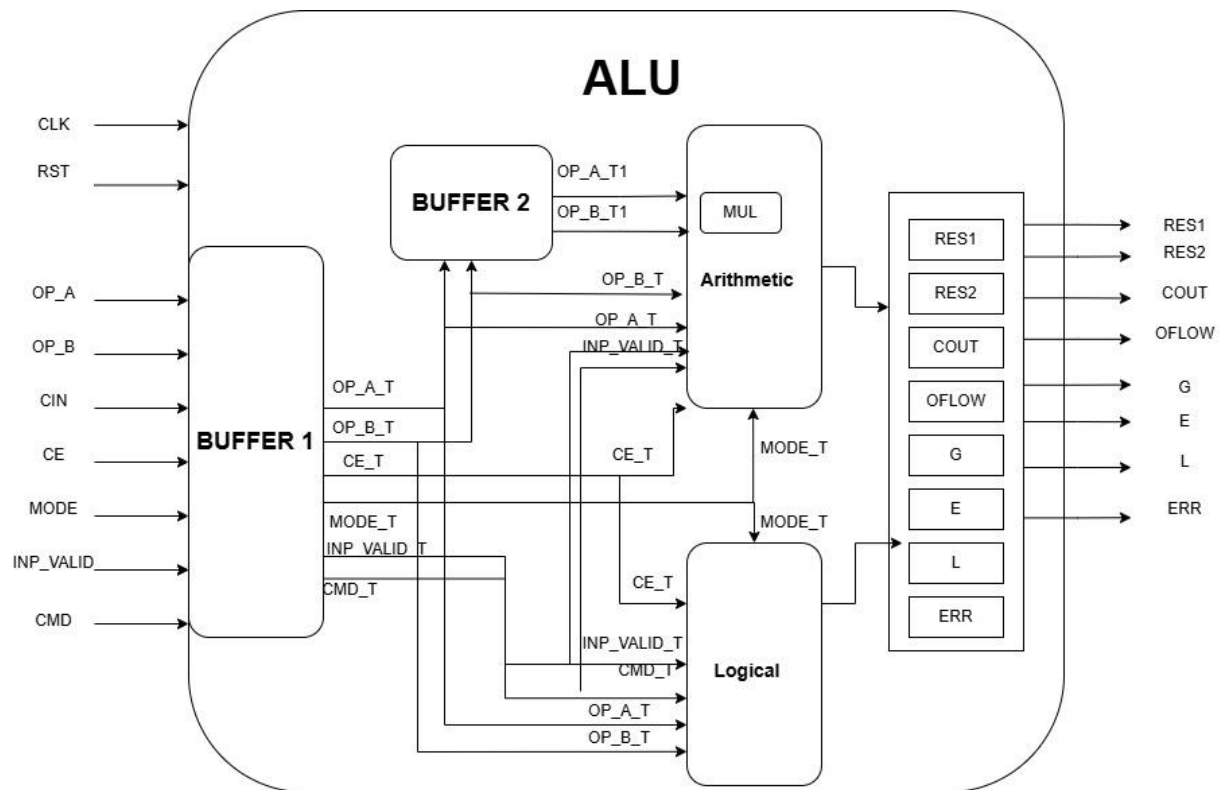
The ALU architecture depicted above illustrates a modular and parameterized combinationallogic unit designed for efficient execution of both arithmetic and logical operations. Thearchitecture is structured into functional blocks, making the design highly scalable, testable,and synthesizable.

Input Buffering Block:

• Inputs like operands OPA, OPB, carry-in CIN, command CMD, control signals MODE,
IN_VALID, RST, CE, and clock are first passed through buffers for synchronization
and signal conditioning.
• Internal signals such as OPA_T, OPB_T, and control lines like CMD_T, MODE_T, and
IN_VALID_T are generated.

Operation Classification:

• **Arithmetic Unit** handles operations like:
o Basic: Addition, Subtraction, Add/Sub with Carry, Compare
o Advanced: Multiplication, Shift and Multiply, Signed operations, and Increment/Decrement
o Multiplication includes additional buffering for latency handling (i.e., Buffer_2
for Multiplication) and output delay management.
**Logical Unit** handles operations including:
• Bitwise: AND, OR, XOR, NAND, NOR, XNOR
• Rotations and Shifts: ROL, ROR, Shift Left/Right A/B
• Negations: NOT_A, NOT_B

Mode Selector:

• Based on the MODE signal, either the Arithmetic or Logical block is selected to
perform the required operation.
• Ensures correct routing of operands and control signals between functional units and
output interfaces.

Output Register Bank:

• Includes multiple result registers:
o RES for logic/arithmetic results
o MUL_RES for multiplication-specific output
o COUT, OFLOW for carry and overflow detection
o G, E, L for compare operations
o ERR for detecting invalid operations, control mismatches, or illegal inputs
• Support for signed/unsigned operations through operand conditioning.
• Command-driven operand validity ensures accurate operation triggering.
• Built-in error detection mechanisms handle command and operand validation.
• Internal pipelining and multiplexed control flow ensure data stability and timing accuracy,
especially during multi-cycle operations like multiplication.

Testbench Architecture
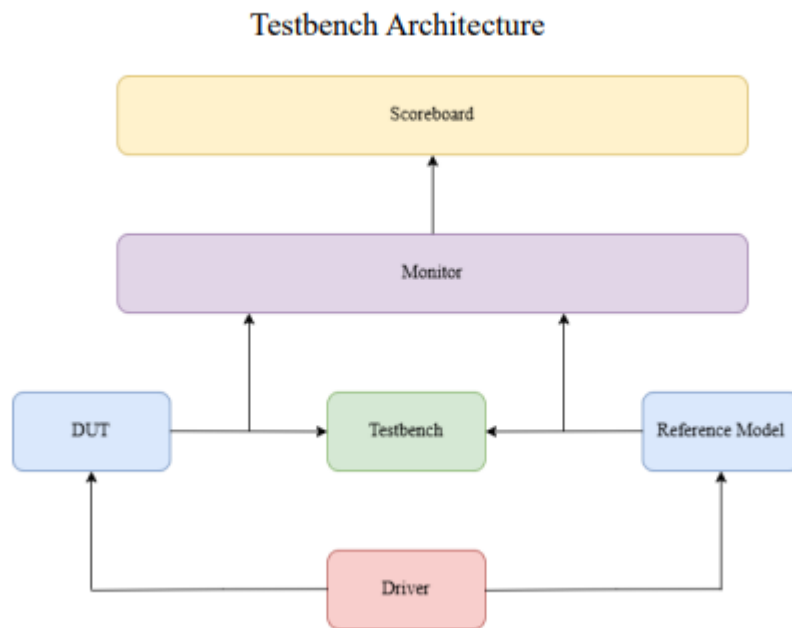
The testbench architecture shown above is a self-checking, modular verification environment
designed to validate the functionality of the ALU. It consists of the following key components:

• **Driver:** Responsible for generating stimulus by driving random and directed test cases
to both the Design Under Test (DUT) and the Reference Model. It controls input signals
like operands, command, mode, and control flags.

• **Testbench:** Acts as a wrapper that instantiates the DUT, Reference Model, Driver,
Monitor, and Scoreboard, facilitating communication among them.

• **DUT (Design Under Test):** The actual Verilog ALU being verified for functional
correctness under various scenarios.

• **Reference Model:** A golden model that mimics the ALU functionality and produces
expected outputs for comparison. It is assumed to be bug-free and helps in validating
DUT output accuracy.

• **Monitor:** Observes outputs from both the DUT and Reference Model. It captures
responses and forwards them to the scoreboard.

• **Scoreboard:** Compares the outputs from the DUT and the Reference Model. Anymismatch is flagged, helping in pinpointing incorrect behaviour in the DUT.This architecture enables automation, reusability, and scalability while ensuring accurate andefficient validation. The modular design also supports future extension to incorporate new commands, data types, or operational modes with minimal changes to the testbench structure.

Working:

The ALU is designed as a combinational circuit that performs operations synchronized with
the positive edge of the clock. It supports both arithmetic and logical operations based on the
state of control and data signals. The following describes the working behaviour of the ALU in
detail:

Reset and Enable Behaviour:

• If the asynchronous RST signal is high, all outputs of the ALU are reset to zero.
• If the Clock Enable (CE) signal is low during operation, the ERR flag is set high, and
the result output is forced to zero, indicating that the ALU is disabled.

Operation Mode and Operand Validity:

• The MODE signal (1-bit) determines the operation type:
o MODE = 1: Arithmetic operation
o MODE = 0: Logical operation
• The INP_VALID signal determines the validity of the operands:
o 00: No operand is valid
o 01: Only Operand A is valid
o 10: Only Operand B is valid
o 11: Both operands A and B are valid

Command Execution and Timing:

• All inputs must be correctly driven based on the CMD:
o For operations requiring two operands (e.g., addition or subtraction), both
operands must be valid (IN_VALID = 11). Any mismatch, such as only one

operand being valid, will result in the ERR flag being set high.
o For operations requiring one operand, only that operand and its corresponding
valid bit should be driven. Incorrect IN_VALID values will also raise the ERRflag.

• Execution Latency:
o For multiplication commands (CMD = 9 or 10) under MODE = 1, the result is
produced after three clock cycles.
o For all other operations, the result is generated after two clock cycles.

Command Restrictions and Error Handling:

• Under logical mode (MODE = 0), if the CMD value is greater than 13, it is treated as
an invalid operation, and the ERR flag is raised.
• Under arithmetic mode (MODE = 1), if the CMD value exceeds 12, the ERR flag is set.
• For logical operations with CMD = 12 or 13, if bits 4 to 7 of OPB are all set to 1, the
ERR flag is asserted.

Compare Operation:

• When CMD = 8 in MODE = 1, the ALU performs a comparison between Operand A
and Operand B.
• Based on the result:
o G is set if A > B
o L is set if A < B
o E is set if A = B
This structured behaviour ensures that the ALU handles a wide range of valid operations while
robustly detecting and flagging invalid input scenarios through the ERR output.

Working of the Self-Checking Testbench:

To verify the functional correctness of the ALU design, a self-checking testbench has been implemented using a modular verification environment, as depicted in the block diagram. The testbench architecture follows a structured approach with key components that collaborate to automate and validate ALU operations against expected outcomes.

Driver:

The driver is responsible for generating a variety of test scenarios. It drives input stimuli such as operands, control signals (CMD, MODE, IN_VALID, etc.) - to both the Device Under Test (DUT) and the Reference Model. This ensures consistency in input conditions across both entities.

Testbench:

The testbench acts as the central module coordinating the signal routing and execution flow. It receives inputs from the Driver and applies them to the DUT and Reference Model simultaneously, maintaining the same environment for both.

DUT and Reference Model:

• The DUT is the actual ALU design being tested.
• The reference model is a golden model that performs the same operations as the DUT but is known to produce correct results. It serves as the baseline for comparison.

Monitor:

The monitor captures the outputs from both the DUT and the Reference Model. It collects the resulting values (such as RES, ERR, flags like COUT, OFLOW, etc.) and packages them for validation.

Scoreboard:

The scoreboard is the decision-making component. It compares the results captured by the Monitor from the DUT against the expected results from the Reference Model. Any mismatches are flagged as functional errors, while matches confirm correct behaviour. This automated self-checking testbench enables comprehensive testing by running multiple test cases without manual intervention. It enhances verification efficiency, ensures correctness of the ALU design under different operation modes and edge cases, and facilitates early detection of bugs in the hardware logic.
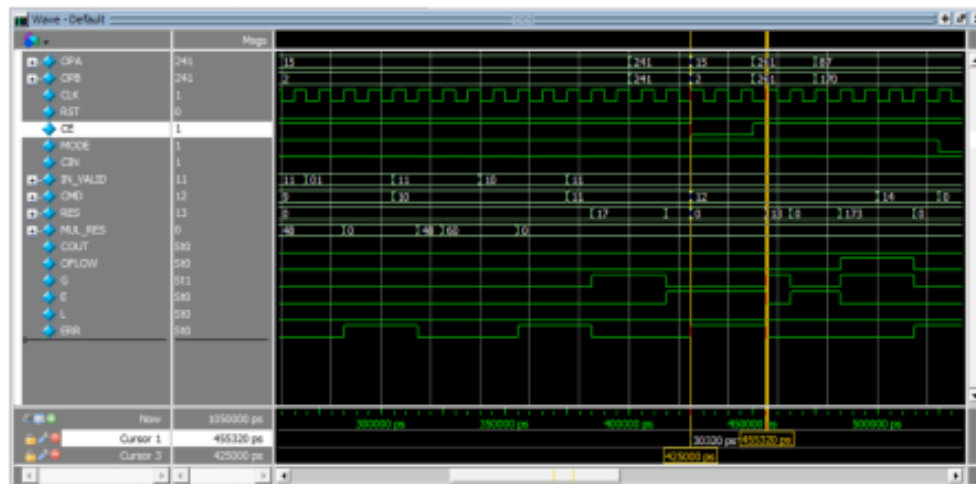
RESULT:-
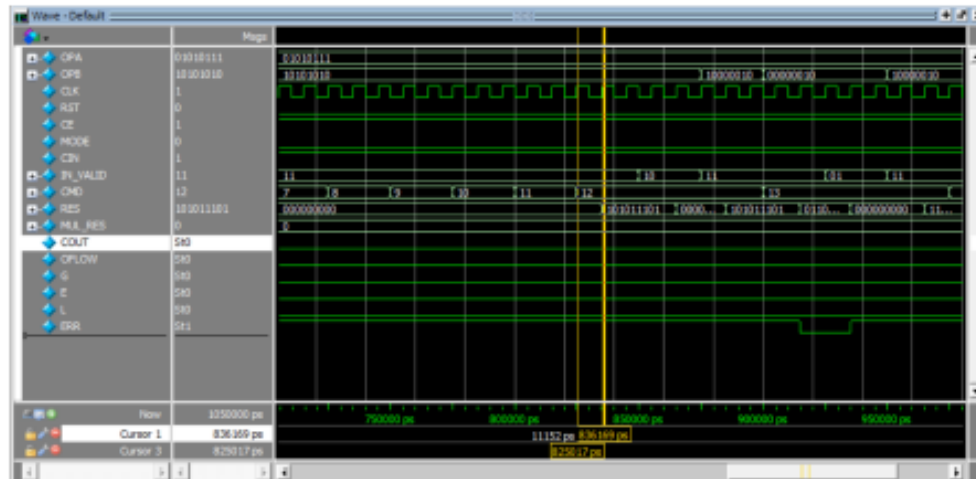
Operations that produce output after two clock cycles
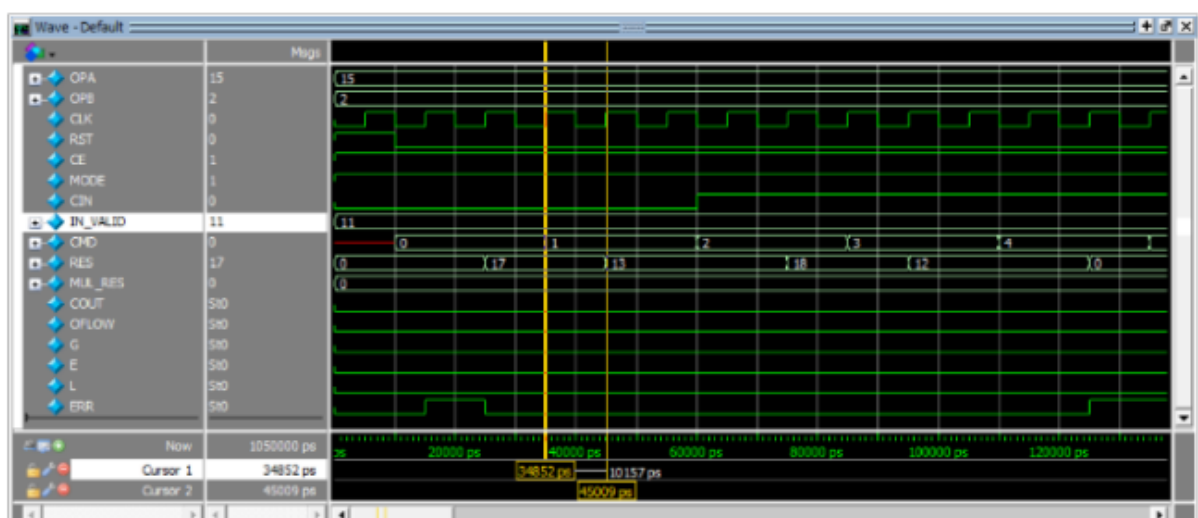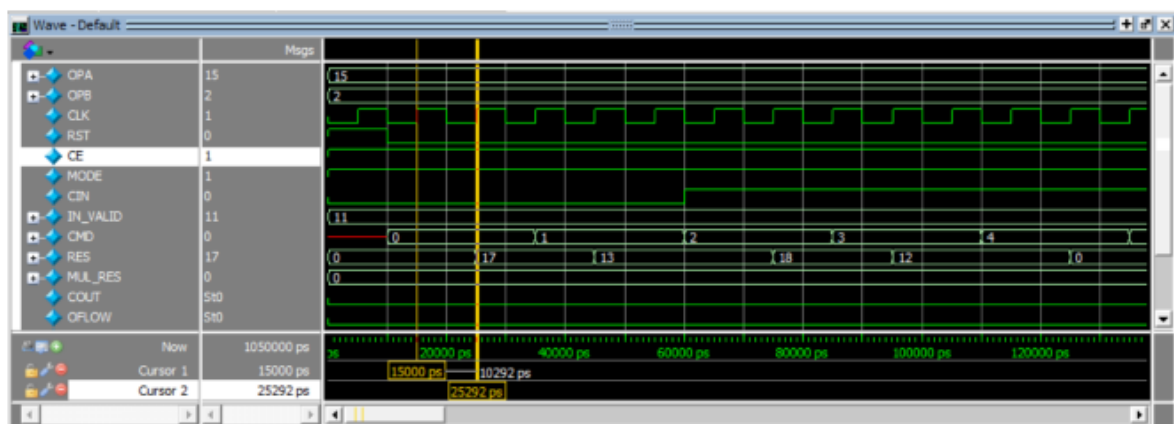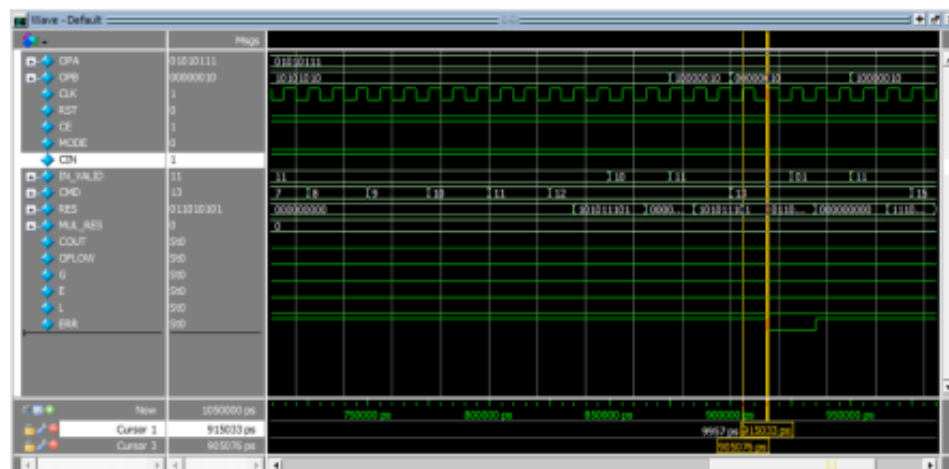
Operations that produce output after three clock cycle
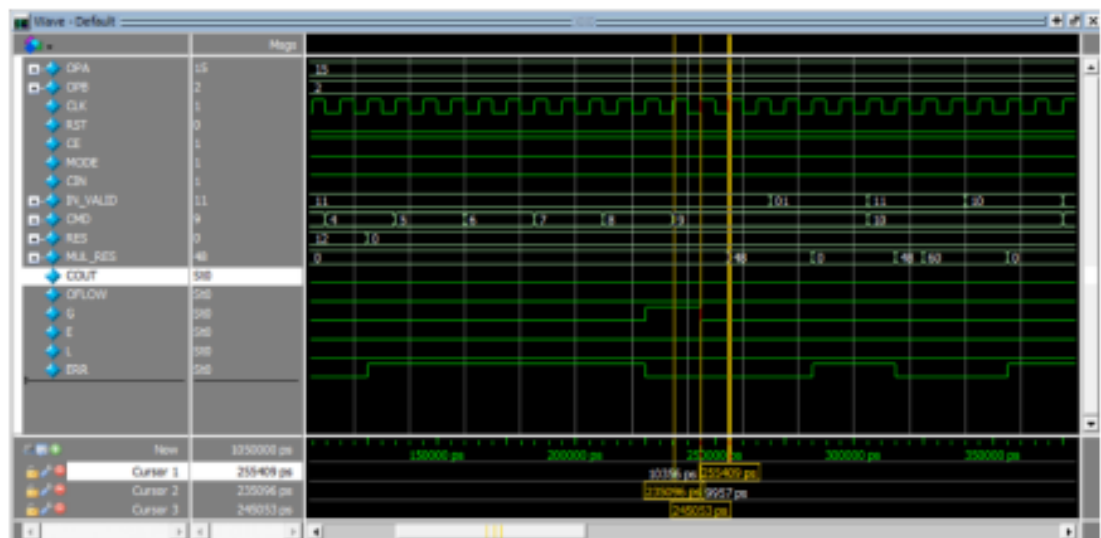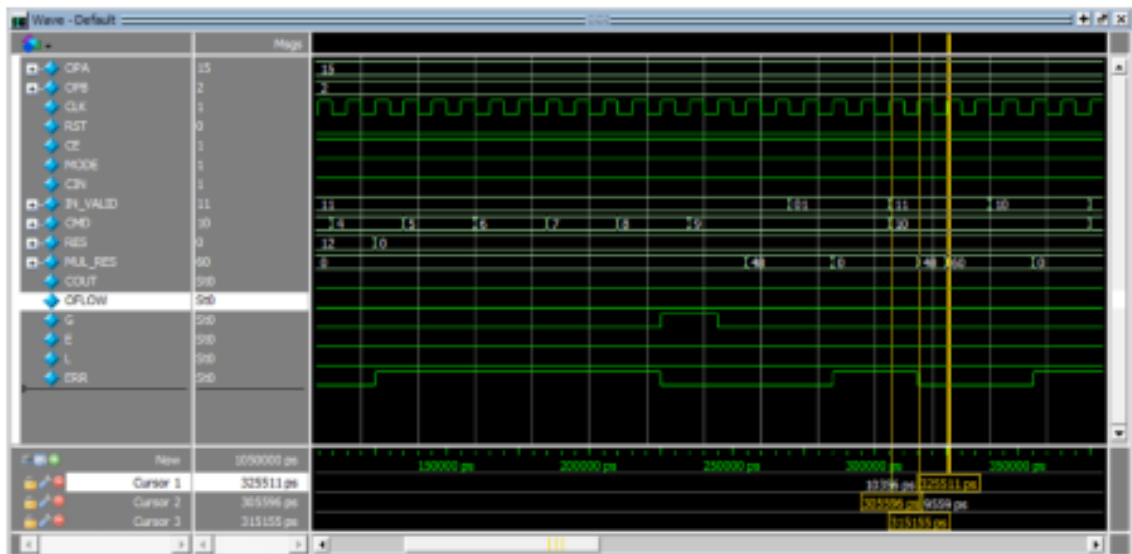
ERR flag being set high when the CE is low

Rotate left operation

Rotate right operation

CONCLUSION:-The Verilog-based ALU design successfully meets all functional requirements, supporting a wide range of arithmetic and logical operations as specified. The implementation follows a modular, synthesizable, and testable architecture, ensuring clarity, maintainability, and hardware compatibility. A comprehensive self-checking testbench was developed to validate the design under various input scenarios, enabling automated verification through comparison with a reference model.

All features and edge cases were thoroughly tested, and the functional correctness of the ALU was confirmed. Furthermore, the testbench achieved an impressive code coverage of 99.5%when analysed using QuestaSim, demonstrating the robustness and completeness of the verification process.

Future Improvement:-
While the current ALU design already incorporates pipelined stages and supports wider datawidths through parameterization, there are several directions for further enhancement. Futureimprovements may include:

• Expanding the Operation Set: Adding more complex operations such as division,
square root, or trigonometric functions to increase ALU versatility.
• Formal Verification Integration: Incorporating formal verification techniques
alongside simulation to mathematically prove correctness across all possible input
combinations.
• Power and Area Optimization: Refining the logic to reduce dynamic power
consumption and silicon area, making the design more suitable for low-power or
resource-constrained environments.