DESIGN DOCUMENT

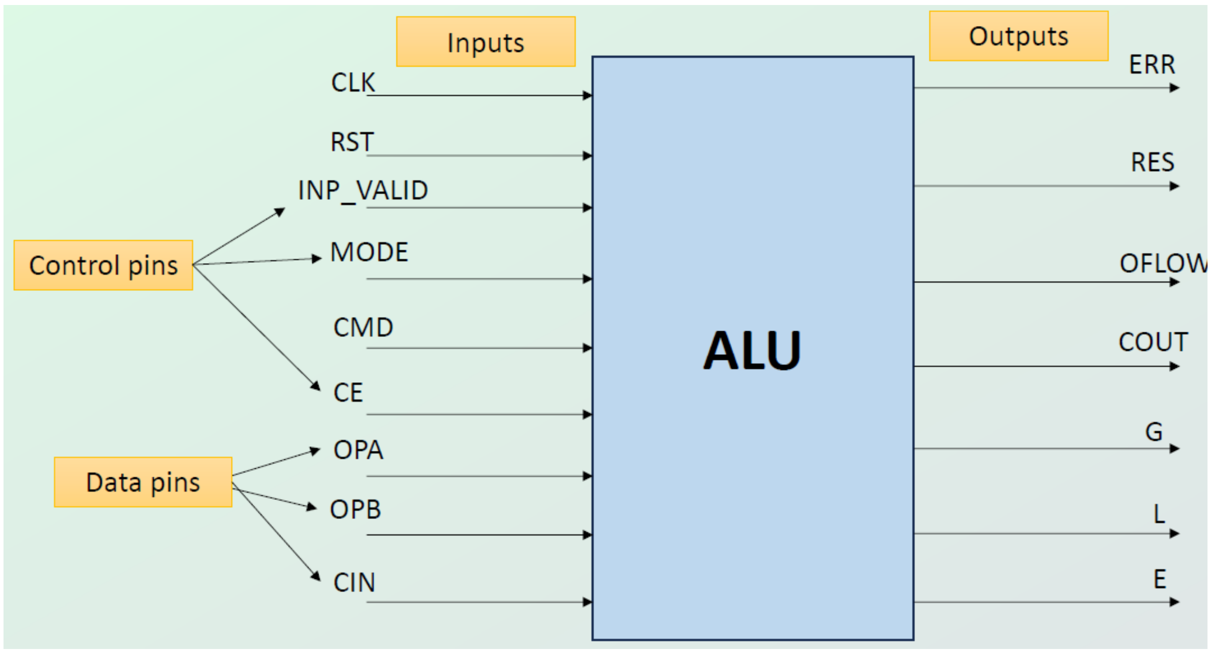
ALU Project

YAGNADATT SARANGI
EMP ID: 6124
02.06.2025

Introduction:

The Arithmetic Logic Unit (ALU) is a fundamental combinational circuit that performs essential arithmetic and logical operations within a digital system. In this design, the ALU is implemented using Verilog, a hardware description language widely used for modelling digital systems.

Pin-Out diagram of ALU



This pin-out diagram provides a structural overview of how the ALU interfaces with external components. The design features clearly categorized inputs and outputs to enable smooth data flow and operation control. Control pins manage the operation mode and synchronization, while data pins provide the operands required for computation. The outputs reflect the results of the operations along with relevant status indicators such as comparison and error flags.

By structuring the ALU in this way, the design ensures modularity, reusability, and ease of integration into larger digital systems such as CPUs and signal processors.

Objectives:

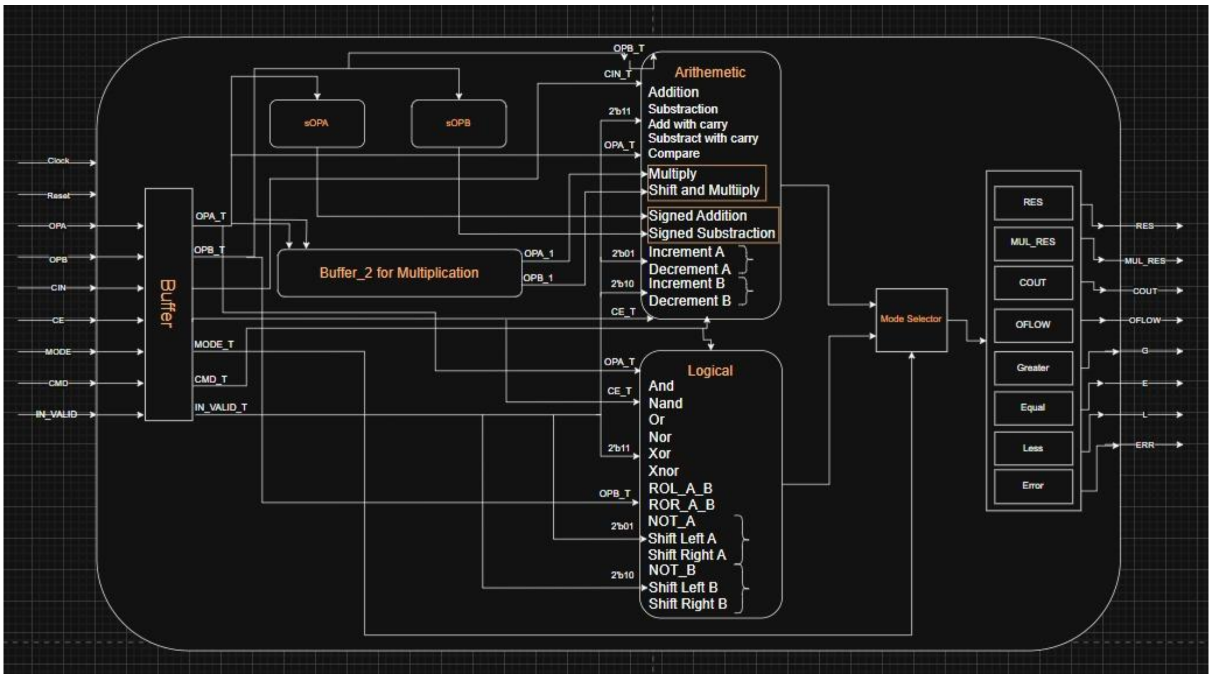
The objective of this project is to design and implement a versatile ALU in Verilog that supports a wide range of arithmetic and logical operations. The ALU is constructed as a combinational circuit with a structured set of inputs and outputs, as illustrated in the pin-out diagram, allowing seamless integration into larger digital systems.

The project aims to enable the ALU to perform operations based on control signals and operand inputs, while generating accurate output results and status flags such as carry-out, overflow, and comparison indicators. It emphasizes modular design, efficient signal handling, and operation mode flexibility.

To ensure the correctness and reliability of the ALU, the project includes an exclusive self-checking testbench. This testbench automatically applies a series of test cases, compares the ALU's outputs with expected results, and flags any mismatches. This approach enables thorough and automated verification of the ALU's functionality across all supported operations.

Architecture:

ALU Architecture



The ALU architecture depicted above illustrates a modular and parameterized combinational logic unit designed for efficient execution of both arithmetic and logical operations. The architecture 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:
    - Basic: Addition, Subtraction, Add/Sub with Carry, Compare
    - Advanced: Multiplication, Shift and Multiply, Signed operations, and Increment/Decrement
    - 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:
    - RES for logic/arithmetic results
    - MUL_RES for multiplication-specific output
    - COUT, OFLOW for carry and overflow detection
    - G, E, L for compare operations
    - ERR for detecting invalid operations, control mismatches, or illegal inputs

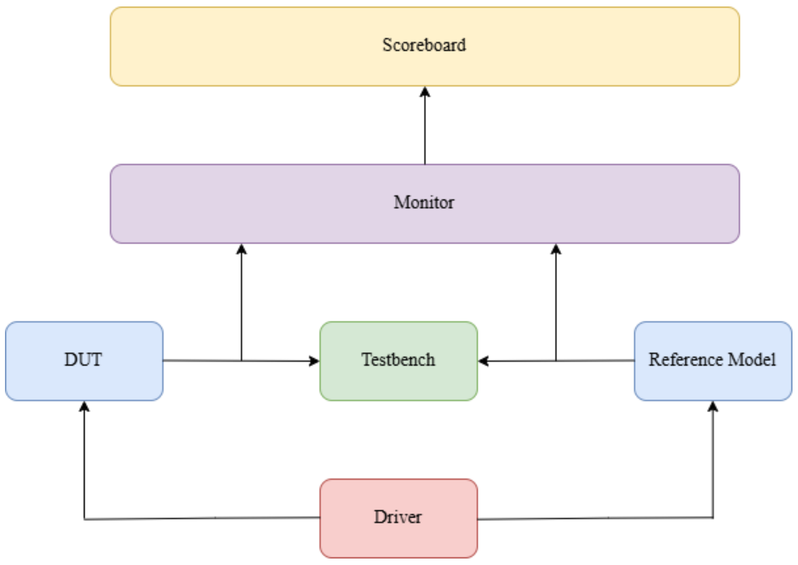
• Support for signed/unsigned operations through operand conditioning.

• 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. Any mismatch is flagged, helping in pinpointing incorrect behaviour in the DUT.

This architecture enables automation, reusability, and scalability while ensuring accurate and efficient 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:
  - MODE = 1: Arithmetic operation
  - MODE = 0: Logical operation
- The INP_VALID signal determines the validity of the operands:
  - 00: No operand is valid
  - 01: Only Operand A is valid
  - 10: Only Operand B is valid
  - 11: Both operands A and B are valid

Command Execution and Timing:

- All inputs must be correctly driven based on the CMD:
  - 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.
  - For operations requiring one operand, only that operand and its corresponding valid bit should be driven. Incorrect IN_VALID values will also raise the ERR flag.
- **Execution Latency**:
  - For multiplication commands (CMD = 9 or 10) under MODE = 1, the result is produced after three clock cycles.
  - 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:
  - G is set if A > B
  - L is set if A < B
  - 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.

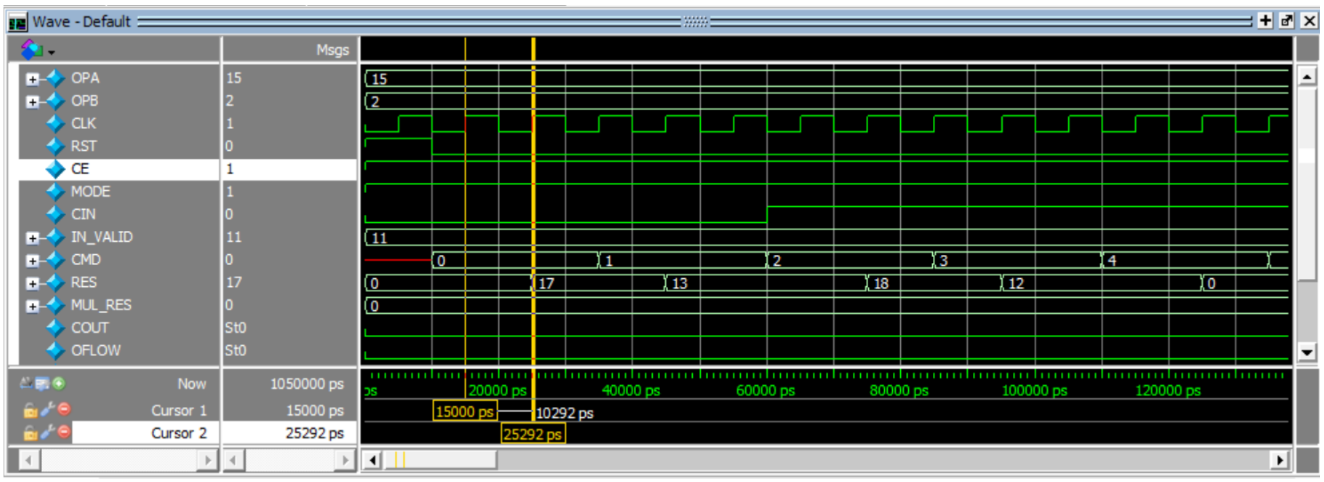
Monitor:
The monitor captures the outputs from both the DUT and the Reference Model. It collects the resulting values (such as RES, ERR, flags like COUT, OFLOW, etc.) and packages them for validation.

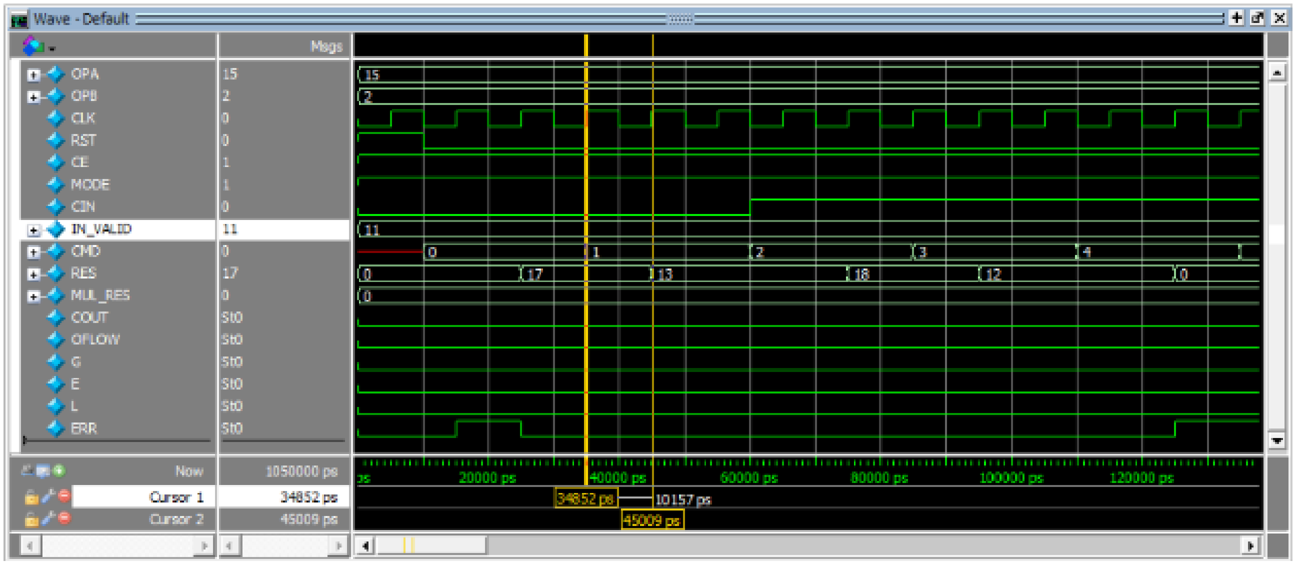
Scoreboard:
The scoreboard is the decision-making component. It compares the results captured by the Monitor from the DUT against the expected results from the Reference Model. Any mismatches are flagged as functional errors, while matches confirm correct behaviour.

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.

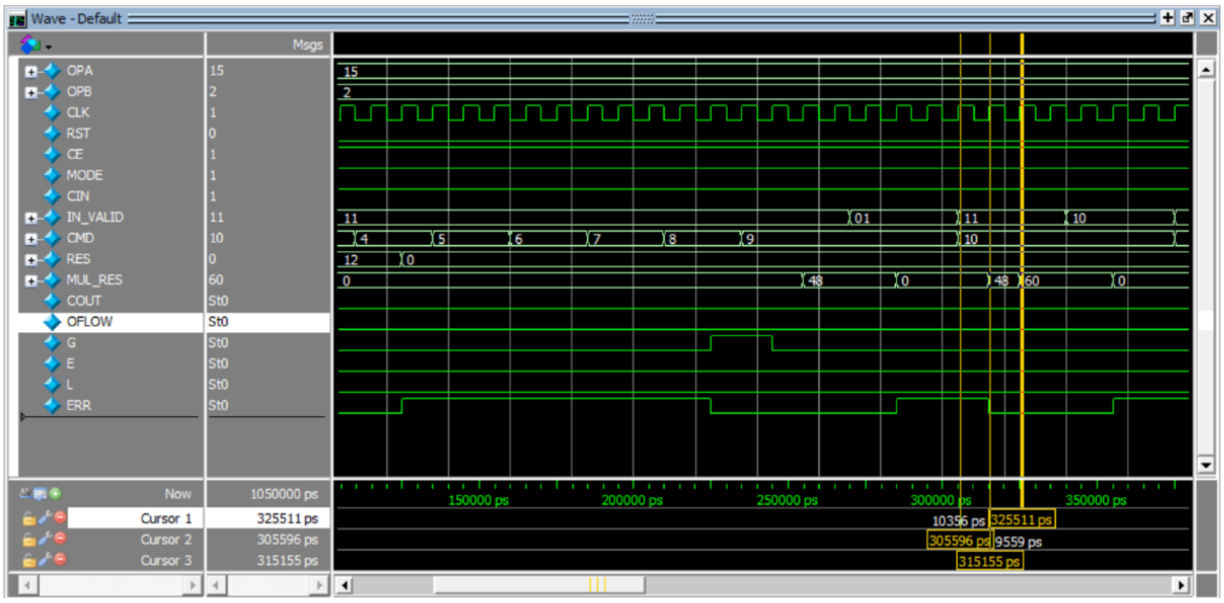
Result:

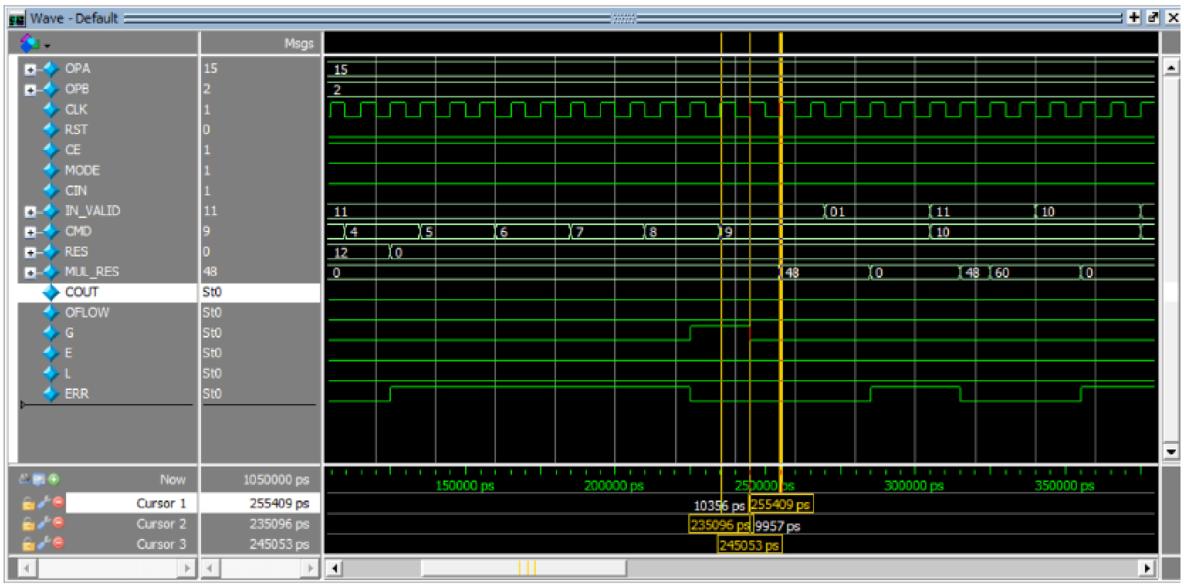
Operations that produce output after two clock cycles



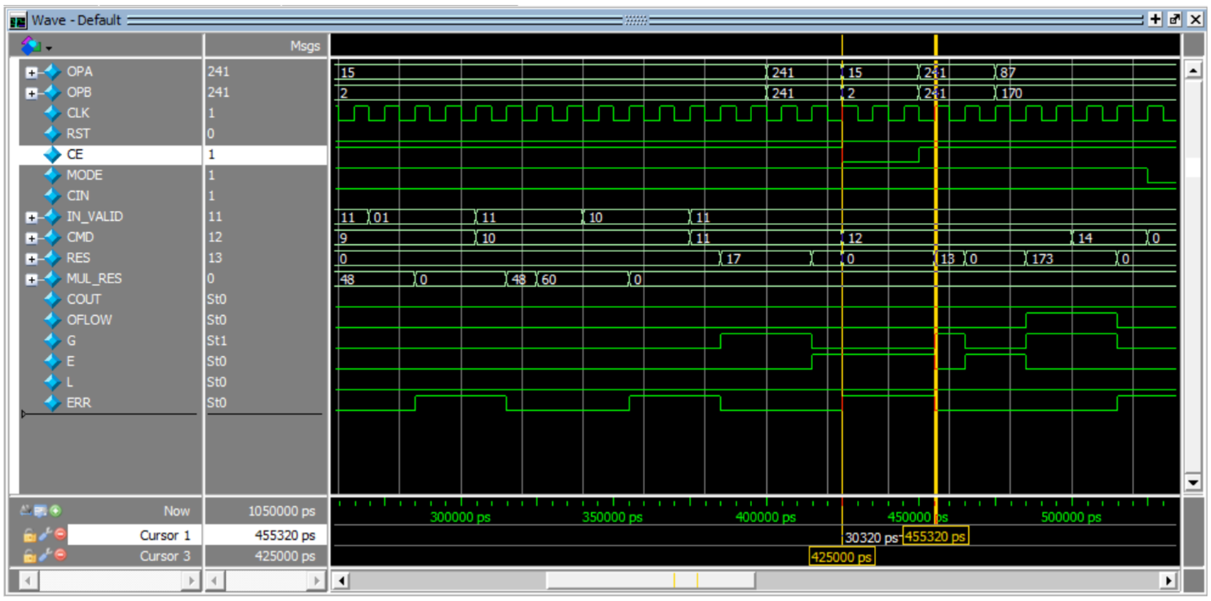


Operations that produce output after three clock cycle



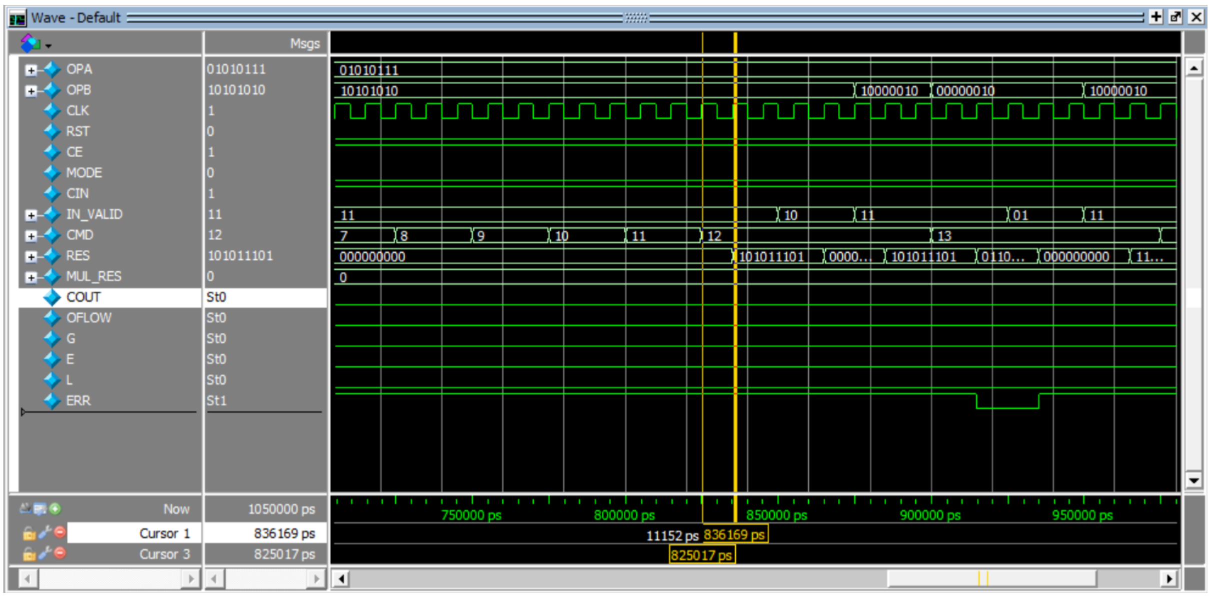


ERR flag being set high when the CE is low


Wave - Default

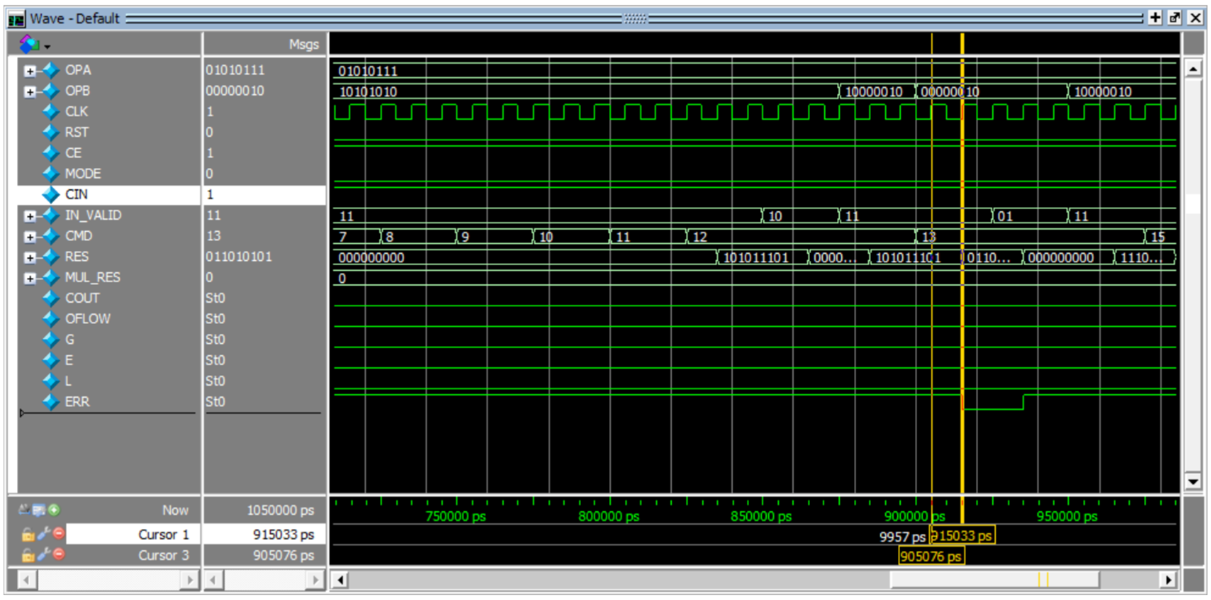
	Msgs
OPA	241
OPB	241
CLK	1
RST	0
CE	1
MODE	1
CIN	1
IN_VALID	11
CMD	12
RES	13
MUL_RES	0
COUT	St0
OFLOW	St0
G	St1
E	St0
L	St0
ERR	St0
Now	1050000 ps
Cursor 1	455320 ps
Cursor 3	425000 ps

Rotate left operation


Wave - Default

	Msgs
OPA	01010111
OPB	10101010
CLK	1
RST	0
CE	1
MODE	0
CIN	1
IN_VALID	11
CMD	12
RES	101011101
MUL_RES	0
COUT	St0
OFLOW	St0
G	St0
E	St0
L	St0
ERR	St1
Now	1050000 ps
Cursor 1	836169 ps
Cursor 3	825017 ps

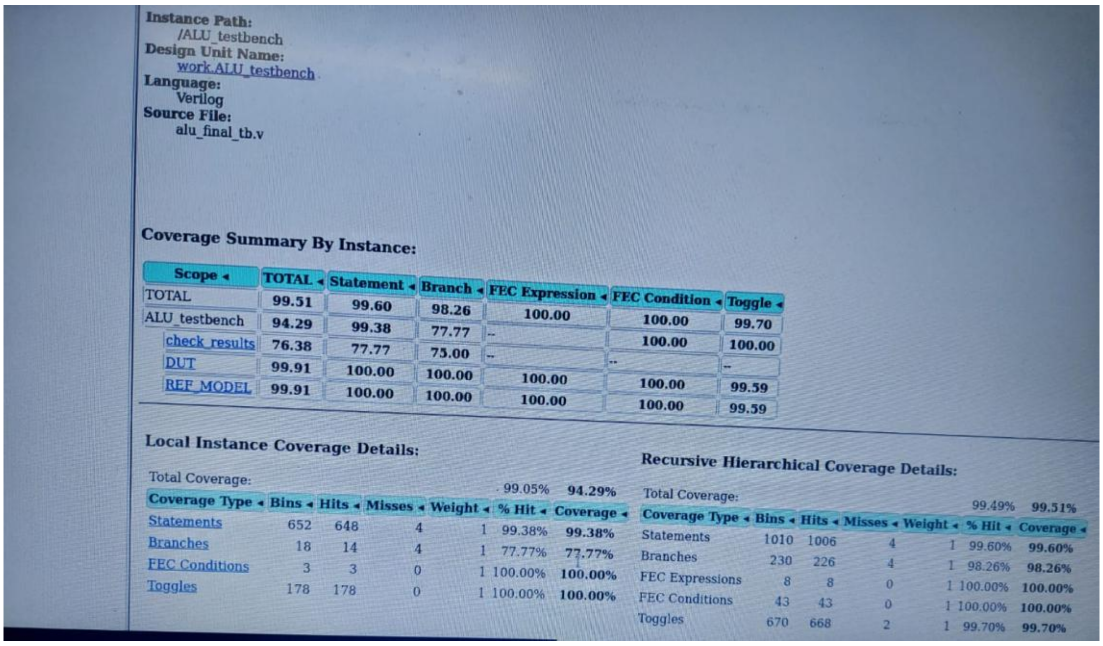
Rotate right operation



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.



Instance Path:
/ALU_testbench
Design Unit Name:
work.ALU_testbench
Language:
Verilog
Source File:
alu_final_tb.v

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Expression	FEC Condition	Toggle
TOTAL	99.51	99.60	98.26	100.00	100.00	99.70
ALU_testbench	94.29	99.38	77.77	--	100.00	100.00
check_results	76.38	77.77	75.00	--	--	--
DUT	99.91	100.00	100.00	100.00	100.00	99.59
REF_MODEL	99.91	100.00	100.00	100.00	100.00	99.59

Local Instance Coverage Details:

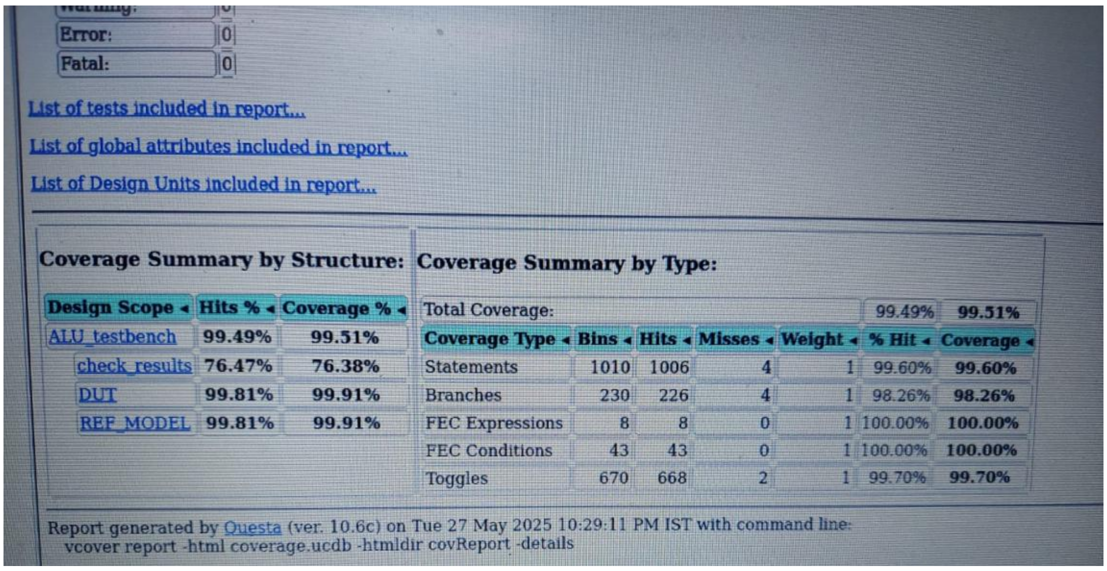
Total Coverage: 99.05% 94.29%

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	652	648	4	1	99.38%	99.38%
Branches	18	14	4	1	77.77%	77.77%
FEC Conditions	3	3	0	1	100.00%	100.00%
Toggles	178	178	0	1	100.00%	100.00%

Recursive Hierarchical Coverage Details:

Total Coverage: 99.49% 99.51%

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Statements	1010	1006	4	1	99.60%	99.60%
Branches	230	226	4	1	98.26%	98.26%
FEC Expressions	8	8	0	1	100.00%	100.00%
FEC Conditions	43	43	0	1	100.00%	100.00%
Toggles	670	668	2	1	99.70%	99.70%



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 %
ALU_testbench	99.49%	99.51%
check_results	76.47%	76.38%
DUT	99.81%	99.91%
REF_MODEL	99.81%	99.91%

Coverage Summary by Type:

Total Coverage:					99.49%	99.51%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	1010	1006	4	1	99.60%	99.60%
Branches	230	226	4	1	98.26%	98.26%
FEC Expressions	8	8	0	1	100.00%	100.00%
FEC Conditions	43	43	0	1	100.00%	100.00%
Toggles	670	668	2	1	99.70%	99.70%

Report generated by Questa (ver. 10.6c) on Tue 27 May 2025 10:29:11 PM IST with command line:

```
vcover report -html coverage.ucdb -htmldir covReport -details
```

Future Improvement:

While the current ALU design already incorporates pipelined stages and supports wider data widths through parameterization, there are several directions for further enhancement. Future improvements 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.