ALU Verification Plan

CONTENTS Page Number

Project Overview

- This project involves verifying a parameterized ALU design supporting arithmetic and logical operations.
- The ALU processes 8-bit operands (OPA, OPB) with scalability to 16/32/64/128 bits.
- It features two operational modes: Arithmetic (MODE=1) and Logical (MODE=0).
- Fourteen commands are implemented through a 4-bit CMD input for diverse operations.
- Arithmetic functions include addition, subtraction, and special multiplication variants.
- Logical operations cover all standard bitwise functions (AND, OR, XOR, etc.).
- Unique rotation commands (ROL/ROR) use operand B to control rotation steps.
- A 2-bit INP_VALID signal validates operand inputs with four possible states.
- The design includes a 16-cycle timeout mechanism for operand synchronization.
- Outputs include a 9-bit result (RES) with carry bit and overflow detection.
- Three comparator flags (G/E/L) indicate greater-than, equal-to, or less-than results.
- An ERR flag detects illegal rotation commands and timeout conditions.
- Clock enable (CE) and asynchronous reset (RST) control operational states.
- All operations are synchronized to the positive edge of the CLK signal.
- The design handles both single-operand and dual-operand commands.
- Special functions include increment-then-multiply and shift-then-multiply.
- Input validation ensures proper operand pairing with commands.
- The ALU implements error detection for invalid rotation parameters.
- Output flags provide comprehensive status information (carry, overflow, comparison).
- The verification covers all operational modes, commands, and edge cases.

Verification Objectives

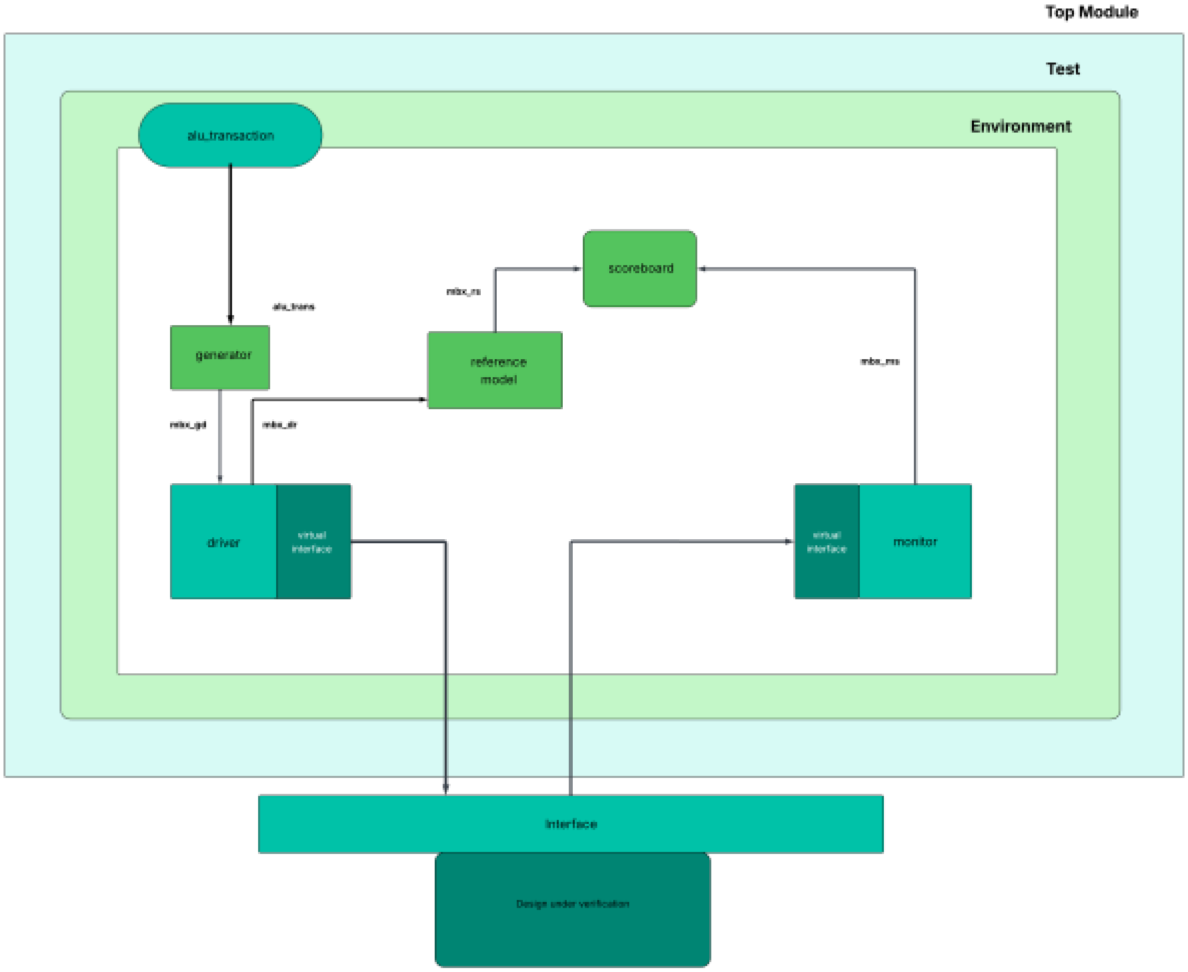
- Verify all arithmetic operations (ADD, SUB, etc.) produce correct results
- Validate logical operations (AND, OR, XOR, etc.) yield accurate outputs
- Confirm INC_MUL command correctly increments operands before multiplying
- Verify SHIFT_MUL properly left-shifts OPA before multiplication
- Test ROL/ROR operations with all valid rotation counts (0-7)
- Validate error flag (ERR) triggers for invalid rotation commands
- Confirm comparator flags (G/E/L) work for all input combinations
- Verify overflow detection (OFLOW) functions for arithmetic operations
- Test carry output (COUT) for addition/subtraction operations
- Validate INP_VALID combinations control operand processing correctly
- Verify 16-cycle timeout mechanism triggers ERR when activated
- Confirm RST asynchronously resets all outputs and internal states
- Test CE functionality to enable/disable clock operations
- Verify MODE switching between arithmetic and logical operations
- Validate all single-operand commands work with proper INP_VALID
- Confirm dual-operand commands require both inputs valid
- Test boundary conditions for maximum/minimum input values
- Verify operations with zero-value operands
- Confirm bit-width scalability (8→128 bits) maintains functionality
- Validate all outputs remain stable during invalid/no-operation states

DUT INTERFACE

The ALU interface (`alu_if`) serves as the critical link between the testbench and the DUT, synchronizing all communication to the clock (`clk`) and reset (`rst`) signals. It defines the 8-bit operand inputs `op_a` and `op_b` for arithmetic and logical operations, along with the carry-in bit `cin` for arithmetic functions. The interface includes control signals like the mode selector (`mode`) to switch between arithmetic and logical operations, clock enable (`ce`) to freeze operations, and a 4-bit command input (`cmd`) to select from 14 supported operations. A 2-bit `inp_valid` signal validates the operands, ensuring proper alignment with commands. Outputs consist of a 9-bit result (`res`) combining an 8-bit output with a carry bit, along with status flags for carry (`cout`), overflow (`oflow`), comparison results (`G`, `E`, `L`), and errors (`err`). Three dedicated clocking blocks (`drv_cb`, `mon_cb`, `ref_cb`) manage signal timing: `drv_cb` drives inputs with a 1ns delay to simulate real driver behavior, `mon_cb` samples outputs with zero delay for accurate monitoring, and `ref_cb` provides a synchronization point for the reference model. Built-in assertions automatically check for unknown values on all inputs, generating timestamped error messages if violations occur. Access control is enforced through modports (`DRV`, `MON`, `REF`), restricting drivers to input control while allowing monitors to observe outputs and reset states.

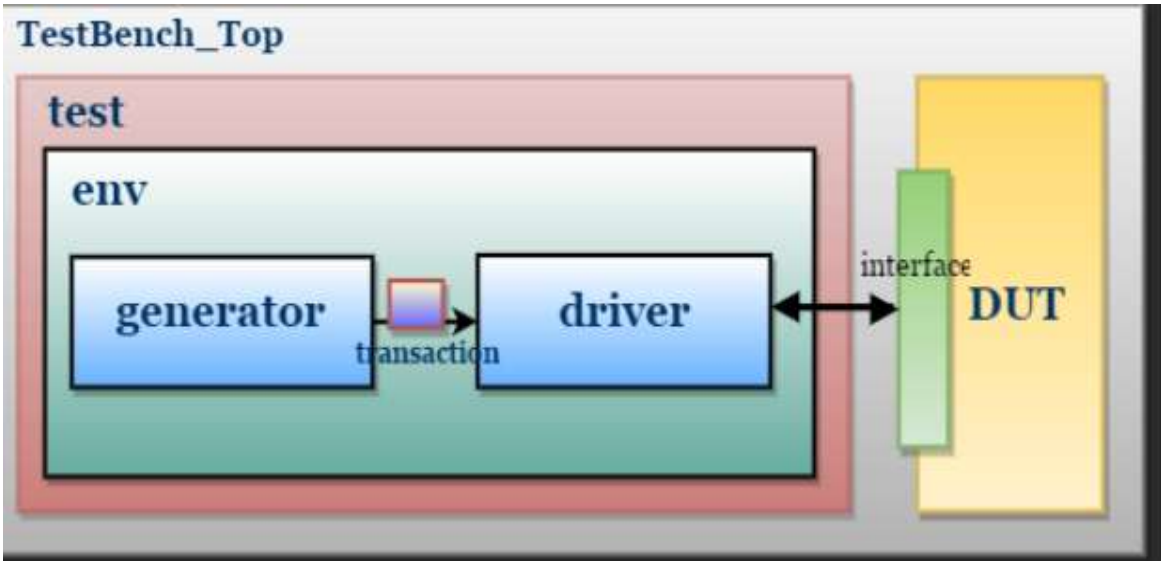
The interface ensures reliable operation through rigorous checks and timing controls. Input assertions act as guardrails, preventing unknown values on `op_a`, `op_b`, `cin`, `ce`, `mode`, `inp_valid`, and `cmd` from propagating into the DUT. These checks enforce protocol rules, such as valid operand-command pairings and proper mode selections. Outputs like `res`, `cout`, and comparison flags update synchronously with clean timing, while `oflow` and `err` flags respond predictably to arithmetic overflows and illegal operations. The clocking blocks isolate timing concerns, with `drv_cb` modeling realistic driver delays and `mon_cb` capturing precise output behavior. Reset dominance is maintained across all modports, ensuring consistent initialization. The interface also supports design scalability through parameterized data widths and enables formal verification via its SystemVerilog assertions.

Testbench Architecture



Transaction class

The code defines a hierarchical transaction class structure for ALU verification, starting with the base alu_transaction class containing randomized inputs (8-bit operands op_a/op_b, control signals cin/ce/mode, 4-bit cmd, and 2-bit inp_valid) and non-randomized outputs (8-bit res with status flags). Key constraints ensure valid configurations: inputs are never zero (inp_valid!=0), reset is inactive (rst==0), clock enable is active (ce==1), and commands are mode-dependent (arithmetic: 0-10, logical: 0-13). Special constraints handle single-operand commands and rotation operations requiring specific operand B patterns. The base class includes a virtual copy() method for deep cloning transactions. Four specialized classes (alu_transaction1 to alu_transaction4) extend this base class, each overriding the copy() method to return their specific type while maintaining identical property copying functionality. This architecture enables flexible test scenario generation with constrained-random verification, while the inheritance structure allows for customized transaction behavior in different test contexts without modifying the base class. The transaction model supports all ALU operations including arithmetic (add/subtract), logical (AND/OR), and special functions (rotations), with built-in validity checks to prevent illegal combinations.



The **generator** is responsible for creating constrained-random stimulus to verify the ALU design. It uses the alu_transaction class hierarchy to generate valid input combinations while adhering to the design specifications. The generator randomizes operands (op_a, op_b), control signals (cin, ce, mode), commands (cmd), and input validity (inp_valid) while following constraints to ensure legal transactions. It supports different test scenarios by leveraging derived transaction classes (alu_transaction1 to alu_transaction4), enabling varied test conditions. The generator interacts with the driver by placing transactions in a mailbox, ensuring synchronization between stimulus generation and application. It can be configured for directed, random, or corner-case testing, making it flexible for different verification goals.

The constraints prevent illegal states, such as invalid command-mode combinations or incorrect operand validity. The generator also logs transactions for debugging and coverage analysis, helping track which test cases have been exercised. By using inheritance, it allows test-specific modifications without altering the base transaction class. The generator's role is crucial in achieving high functional coverage by ensuring diverse input patterns, including edge cases like maximum/minimum operand values, carry-in scenarios, and special operations like rotations. It works in sync with the testbench to apply back-to-back transactions, stress-testing the ALU under different conditions.

The **driver** receives transactions from the generator via a mailbox and applies them to the ALU DUT through the alu_if interface. It translates high-level transactions into signal-level activity, driving inputs (op_a, op_b, cmd, mode, etc.) at the correct clock edges. The driver uses the drv_cb clocking block to ensure proper timing, with 1ns output delays mimicking real-world driver behavior. It handles reset conditions by forcing signals to default states when rst is active. The driver also checks for protocol violations,.

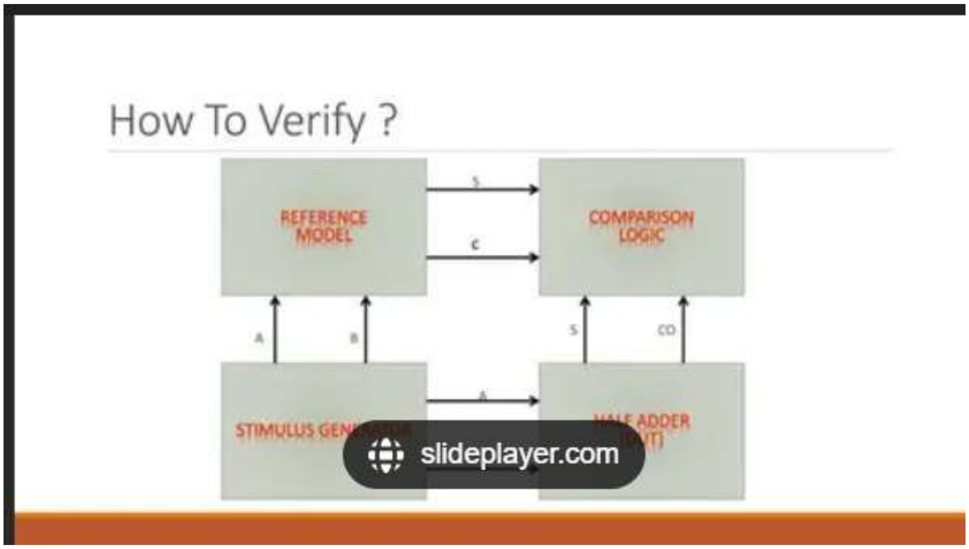
The monitor observes the ALU’s outputs (res, cout, err, flags) via

the mon_cb clocking block, sampling signals at zero delay for accurate timing. It checks for unknown (X/Z) values, protocol violations, and unexpected flag behaviors (e.g., oflow in logical mode). The monitor captures transaction results and forwards them to the scoreboard for checking. It also validates output stability between clock edges, flagging glitches or timing errors. The monitor tracks DUT responses per input command, ensuring operations like ADD, SUB, or ROL produce correct outputs. It detects and reports errors such as incorrect comparison flags (G, E, L) or missed overflow conditions. The monitor synchronizes with the driver to correlate inputs with outputs, enabling cycle-accurate debugging. It logs all activity for waveform analysis and coverage tracking. The monitor plays a key role in assertions, verifying properties like "err must trigger for illegal rotations." It supports both passive monitoring (for normal tests) and active checks (for error injection). By comparing observed results against expected behavior, it helps identify design flaws early.

It supports both single-cycle and multi-cycle operations, such as those requiring 16-clock timeout checks for operand synchronization. The driver can inject errors for negative testing, verifying the ALU’s resilience to illegal inputs. It works closely with the monitor to ensure signal integrity and logs driven values for debugging. The driver’s flexibility allows it to adapt to different test modes, including burst transactions, idle cycles, and backpressure scenarios.

Reference Model

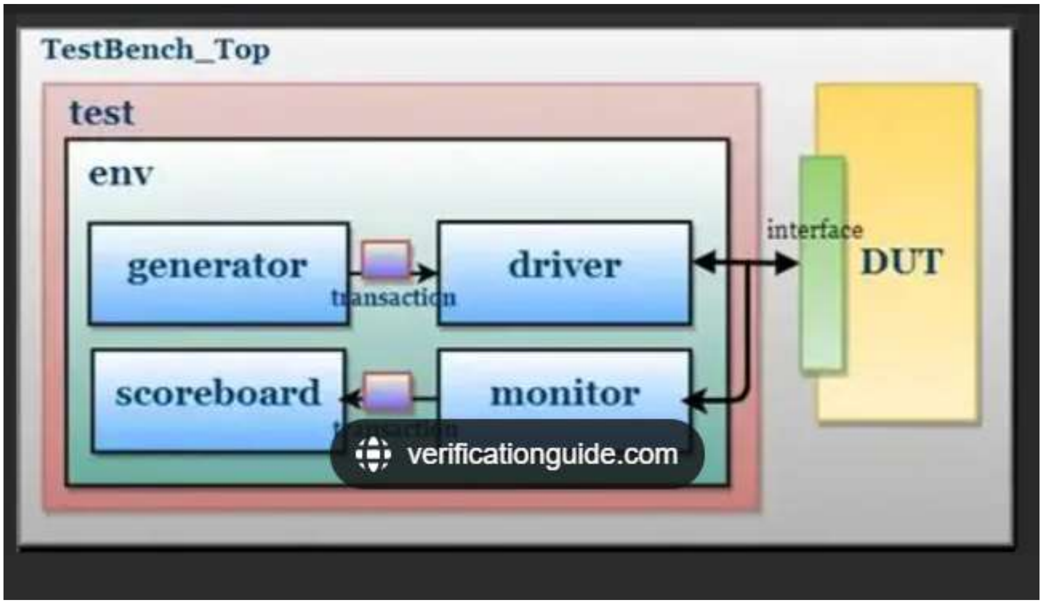
The reference model emulates the ALU's ideal behavior, calculating expected results for each transaction. It processes inputs (op_a, op_b, cmd, etc.) identically to the DUT, generating predictions for res, cout, flags, and err. The model handles arithmetic (ADD/SUB), logical (AND/OR), and special operations (ROL/SHIFT_MUL) with precise bit-level accuracy. It checks operand validity (inp_valid) and mode (mode) to mirror the DUT's decision logic. The model flags errors for illegal rotations (OPB[7:4] ≠ 0) and timeouts (16-cycle rule). It synchronizes with the testbench via the ref_cb clocking block, ensuring alignment with DUT timing. The reference model's outputs are sent to the scoreboard for comparison against DUT responses. It supports all parameterized widths (8-bit to 128-bit) for scalability. The model also logs its computations, aiding in debugging mismatches. By providing a golden reference, it ensures the DUT's correctness across all test scenarios, including corner cases like zero operands or maximum overflows.



How To Verify ?

Scoreboard

The scoreboard compares DUT outputs (from the monitor) with expected results (from the reference model), flagging discrepancies as errors. It checks for matching res, cout, oflow, and flags (G, E, L) on a per-transaction basis. The scoreboard handles synchronization delays, ensuring comparisons account for pipeline effects. It categorizes errors by type (e.g., arithmetic miscalculation, missed overflow) for targeted debugging. The scoreboard maintains coverage statistics, tracking which operations and edge cases have been tested. It supports configurable tolerance for partial matches (e.g., don't-care bits in logical ops). The scoreboard logs all comparisons, generating pass/fail reports and trend analysis. It integrates with functional coverage to identify untested scenarios. By correlating inputs to outputs, it helps isolate root causes of failures. The scoreboard also supports error injection tests, verifying the DUT's handling of illegal states. Its role is critical in achieving verification closure by ensuring the ALU behaves correctly across all modes, commands, and operands.

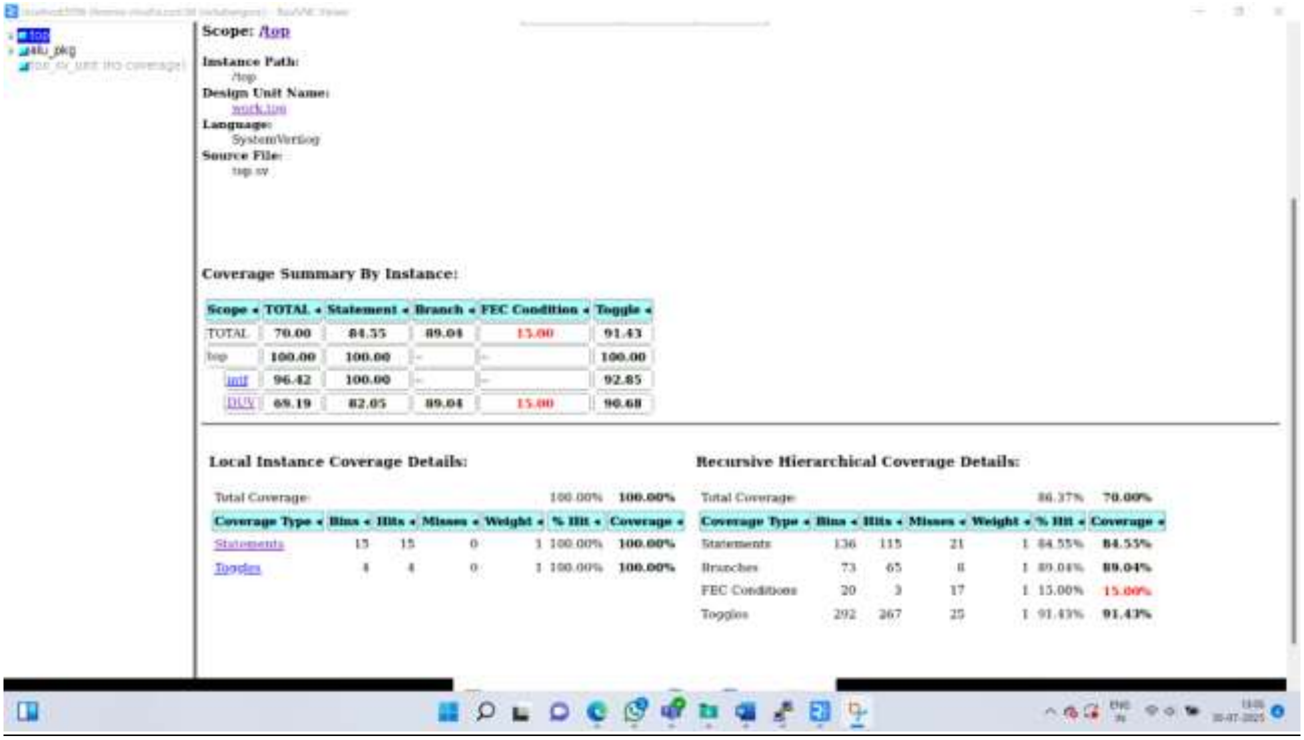


VERIFICATION RESULTS AND ANALYSIS

Error in DUT

The key issue in this simulation is **low Finite State Machine (FSM) coverage (15%)**, indicating that the testbench fails to exercise critical state transitions in the ALU. Despite achieving **100% output functional coverage**, the design's internal FSM paths remain largely untested, risking hidden bugs. The testbench only validates normal operations but misses error states, reset transitions, and illegal command handling. Additionally, 8 warnings suggest potential RTL or testbench issues (e.g., uninitialized signals, dead code). The **input coverage (50.13%)** is also suboptimal, implying insufficient randomization of operands and control signals To fix this, inject invalid commands, test reset scenarios, and add directed tests for edge cases. Without these, the ALU may malfunction in real-world use.

Coverage Report



Scope: /top

Instance Path:
/top
Design Unit Name:
work.top
Language:
SystemVerilog
Source File:
top.sv

Coverage Summary By Instance:

Scope	TOTAL	Statement	Branch	FEC Condition	Toggle
TOTAL	70.00	84.55	89.04	15.00	91.43
top	100.00	100.00	--	--	100.00
intf	96.42	100.00	--	--	92.85
DUV	69.19	82.05	89.04	15.00	90.68

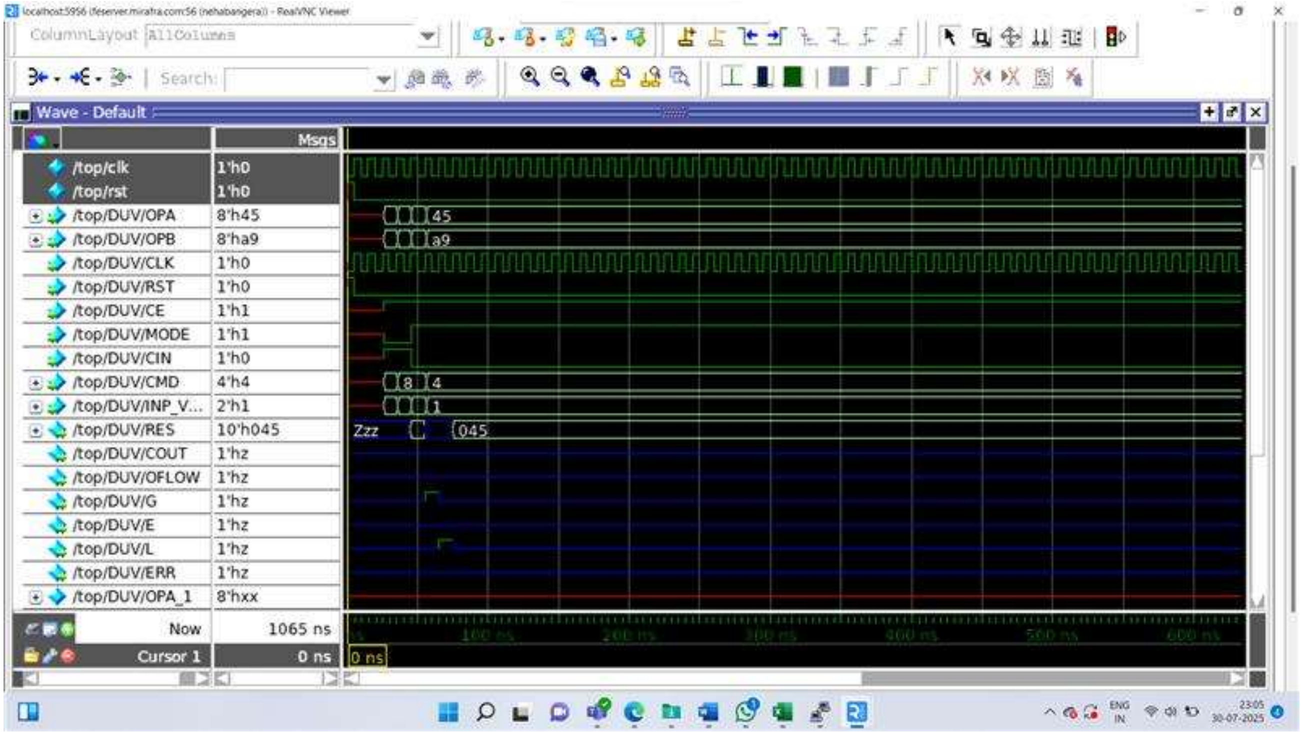
Local Instance Coverage Details:

Total Coverage:					100.00%	100.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	15	15	0	1	100.00%	100.00%
Toggles	4	4	0	1	100.00%	100.00%

Recursive Hierarchical Coverage Details:

Total Coverage:					86.37%	70.00%
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	136	115	21	1	84.55%	84.55%
Branches	73	65	8	1	89.04%	89.04%
FEC Conditions	20	3	17	1	15.00%	15.00%
Toggles	292	267	25	1	91.43%	91.43%

The **100% output functional coverage** confirms all expected ALU operations (like ADD, SUB, etc.) were verified correctly, and the scoreboard matches prove correct results. However, **low FSM (15%) and input (50%) coverage** indicate untested internal states and corner cases, risking hidden bugs. Full reliability requires improving **FSM, toggle, and input coverage** checks.



localhost:5956 (feserver.mirafra.com:56 (nehabangera)) - RealVNC Viewer
ColumnLayout AllColumns
Search:
Wave - Default

	Msgs
/top/clk	1'h0
/top/rst	1'h0
/top/DUV/OPA	8'h45
/top/DUV/OPB	8'ha9
/top/DUV/CLK	1'h0
/top/DUV/RST	1'h0
/top/DUV/CE	1'h1
/top/DUV/MODE	1'h1
/top/DUV/CIN	1'h0
/top/DUV/CMD	4'h4
/top/DUV/INP_V...	2'h1
/top/DUV/RES	10'h045
/top/DUV/COUT	1'hz
/top/DUV/OFLOW	1'hz
/top/DUV/G	1'hz
/top/DUV/E	1'hz
/top/DUV/L	1'hz
/top/DUV/ERR	1'hz
/top/DUV/OPA_1	8'hxx
Now	1065 ns
Cursor 1	0 ns