ALU Verification Plan

Submitted By,

Name: RAKSHITH S SHETTY

Emp. ID:6100

TABLE OF CONTENT

LIST OF FIGURES

CHAPTER 1
PROJECT OVERVIEW AND SPECIFICATION

1.ALU INTRODUCTION:

The Arithmetic Logic Unit is at the core of digital systems and is able to handle multiple arithmetic or logical operations simultaneously. This project centers around the design and verification of a parameterized synchronous ALU supporting both arithmetic and logic functions with configurable bit widths in 16, 32, 64, and 128 bits. The ALU has two inputs with one control interface containing a mode select and a 4-bit opcode to select one of the operations.

It performs unsigned addition, subtraction, increment, decrement, comparison, and several types of multiplication, as well as logical operations including AND, OR, XOR, NOT, and their respective complementary operations: NAND, NOR, and XNOR. Furthermore, it supports shift and rotate operations of both directions with variable shift amounts.

To make the ALU both reliable and efficient, the design includes several advanced features. It supports synchronous operation with clock enable, allows for an asynchronous reset, and includes built-in mechanisms to detect errors during operation. One key feature is a 16-cycle timeout system that keeps track of how long the ALU waits for input operands. If the operands don’t arrive in time, the system automatically uses the most recent values and raises an error signal, helping to prevent unexpected behavior.

Overall, this ALU is built to be flexible and scalable, making it easy to integrate into larger digital systems like embedded platforms, custom processors, or other hardware projects that require a reliable computation unit.

1.2 Advantages of ALU:

- **Scalable Architecture:**

  The ALU is designed to scale effortlessly across different bit-widths — whether it's 8, 16, 32, 64 bits, or more — without needing to modify the core logic. This saves development time and reduces the risk of introducing new bugs.

- **Extensive Operation Support:**

  With 11 arithmetic and 14 logical operations built-in, the ALU handles everything from basic addition to advanced functions like rotate. This reduces the need for extra components or processing elsewhere.

- **Smart Input Management:**

  The ALU uses an INP_VALID signal to manage inputs that arrive at different times. A built-in timeout mechanism ensures the system doesn't hang if inputs are delayed or missing.

- **Informative Status Flags:**

  After each operation, the ALU provides useful flags — such as carry, overflow, greater-than, less-than, equal, and error — giving the system helpful insights for decision-making.

- **Power Efficiency:**

  A clock enable (CE) feature allows the ALU to shut down when not in use, making it ideal for low-power or battery-operated devices.

- **Built-in Error Detection:**

  The design actively detects invalid operations, especially in cases like rotate, where incorrect inputs can lead to unexpected behavior — helping ensure robust and predictable performance.

1.3 Disadvantages of ALU:

- **Fixed Timeout Limit:**

  The ALU always waits exactly 16 clock cycles for input operands—no more, no less. This rigid timeout might be too long for fast systems or too short for slower ones, and unfortunately, it’s not configurable.

- **Dual-Purpose Commands Can Be Confusing:**

  The same operation code (CMD) can mean different things depending on the mode. For example, CMD = 0 might trigger ADD in arithmetic mode but AND in logic mode. If you're not paying close attention, it's easy to make mistakes.

- **Generic Error Flag:**

  There's only one error signal, no matter what goes wrong. Whether it's a timeout, an invalid command, or a wrong input format — you won't know the exact cause. This makes debugging and fixing issues more difficult.

- **Overdesigned for Simple Use-Cases:**

  The ALU is packed with features, which is great for complex systems — but it comes at the cost of increased hardware usage. For applications that only need basic arithmetic or logic, this ALU might be more than necessary.

- **Rotate Operation Requires Careful Inputs:**

  The rotate-left and rotate-right commands have strict input requirements. For instance, certain bits in operand B must be zero (like bits [7:4]). If the software doesn't handle this properly, it can easily cause errors during rotation.

1.4 Use cases of ALU:

- **Arithmetic Operations:**

  The ALU handles essential math tasks like addition, subtraction, multiplication, and division. These operations are vital for things like updating counters, calculating memory addresses, or performing basic computations in software.

- **Logical Operations:**

  It performs standard bitwise logic functions such as AND, OR, XOR, and NOT. These are widely used in tasks like comparing values, bit masking, and making logical decisions in digital circuits and processors.

- **Bit Shifting and Rotation:**

  The ALU can shift bits left or right or rotate bits—operations that are common in encryption, data encoding/decoding, and low-level data manipulation.

- **Comparisons:**

  It can compare two values to determine if one is greater than, less than, or equal to the other. These results are crucial for implementing if-else conditions, loops, and branching decisions in both hardware and software.

- **Address Calculations:**

  Used in memory operations, the ALU can compute the next instruction address or the location of data, enabling smooth control flow in processors.

- **Checksum and CRC Computation:**

  The ALU supports operations that help compute checksums and cyclic redundancy checks (CRC)—important tools for error detection in data communication systems.

- **Control Signal Generation:**

  Based on comparison results, the ALU can help generate control signals for Finite State Machines (FSMs) and other control logic in digital systems.

- **Overflow and Carry Detection:**

  During arithmetic operations, the ALU can detect overflow and carry, which is crucial for handling signed and unsigned numbers correctly and ensuring computational accuracy.

- **CPU Instruction Execution:**

  As the core of the execution stage in a CPU, the ALU is responsible for carrying out the actual computations defined by machine instructions — making it a fundamental part of any processing unit.

1.5 Project Overview of ALU:

This project centers around designing a versatile and powerful Arithmetic Logic Unit (ALU) that's suitable for a wide range of digital systems — from simple embedded controllers to high-performance processors. The standout feature of this ALU is its parameterized architecture, which means it can be easily configured to work with different bit-widths like 16, 32, 64, or even 128 bits, depending on the system's needs. This flexibility makes it ideal for both resource-constrained devices and compute-intensive applications.The ALU takes in two input operands and supports two operating modes: arithmetic mode for operations like addition, subtraction, and

multiplication, and logical mode for functions such as AND, OR, and XOR. A compact 4-bit command interface is used to select among 14 supported operations in each mode, making the ALU both powerful and easy to control.

The architecture is designed to handle asynchronous or staggered inputs through an input-valid signal, ensuring smooth data flow even when inputs arrive at different times. A built-in 16-cycle timeout mechanism prevents indefinite stalling by signaling an error if inputs do not arrive in time, which enhances overall system reliability. The ALU also generates useful status signals such as overflow, carry, and comparison results (greater than, less than, or equal), allowing external systems like CPUs or control units to make real-time decisions based on operation outcomes. Advanced error detection features are included to catch invalid scenarios, especially during complex operations like bit rotations, where incorrect operand formats could otherwise lead to silent failures. By flagging these issues, the ALU simplifies debugging and ensures safer integration into larger systems. Overall, the design balances performance, scalability, and robustness, making it a reliable component for modern digital hardware.

1.6 Design Features:

- **Synchronous Operation with Asynchronous Reset**

  The ALU runs on the rising edge of the clock for consistent timing, but it can reset immediately using an asynchronous reset — ideal for clean start-ups or emergency shutdowns.

- **Easily Configurable Bit Width**

  You can change the data width to 16, 32, 64, or even 128 bits just by updating a parameter — no need to rewrite the whole design. This makes the ALU highly reusable across different systems.

- **Smart Input Readiness Handling**

  The 2-bit INP_VALID signal tells you if zero, one, or both operands are ready. This helps the ALU work smoothly even when inputs don’t arrive at the same time, such as in pipelined or asynchronous setups.

- **Advanced Rotate Capabilities with Safety Checks**

  Rotate operations (left and right) are controlled by operand B, allowing flexible shifts. The ALU also checks for invalid values during rotate, helping you catch errors early.

- **Full Comparison Output in One Go**

  Instead of giving just one result, the ALU outputs all three: greater-than (G), less-than (L), and equal (E). This makes control decisions faster and easier for the system.

- **Automatic Overflow Detection**

  Arithmetic operations are checked for overflow automatically. You don't have to add extra logic to catch it — the ALU flags it for you, preventing calculation errors.

- **Built-in Power Saving Feature**

  The ALU has a Clock Enable (CE) input, so you can switch it off when it's not in use. Perfect for saving power in energy-sensitive or battery-powered devices.

1.7 Design Limitation:

- **One Error Bit for All Issues**

  There's just a single ERR flag that turns high when anything goes wrong. But it doesn't tell you what went wrong — was it a timeout, a wrong command, or bad input? You're left guessing.

- **Limited Rotate Range**

  Rotate operations are restricted to only 8 positions. That's fine for small data widths but not enough for wider operations like 32-bit or 64-bit rotates.

- **Fixed Operand Priority**

  If a timeout happens, the ALU always chooses the most recent operand. While this works in many cases, it might not match the logic expected in all systems — and you can't change this behavior easily.

- **Mode-Based Command Confusion**

  The same command value means different things in arithmetic and logical modes. For example, CMD = 0 might mean ADD in one mode and AND in another. This can easily confuse developers and cause bugs.

- **No Pipelining**

  The ALU handles one operation at a time. There's no pipelining or parallel execution, which means performance might fall short in high-speed or heavily loaded systems.

- **Fixed Timeout**

  The ALU is hardcoded to wait 16 clock cycles for both inputs. If your system runs faster or slower, you can't tune this wait time without editing the design.

- **Wasted Resources**

  All features and operations are implemented in hardware, even if some aren't used. That means more area and power consumption — not ideal for small or low-power chips.

1.8 Design diagram with interface signals:

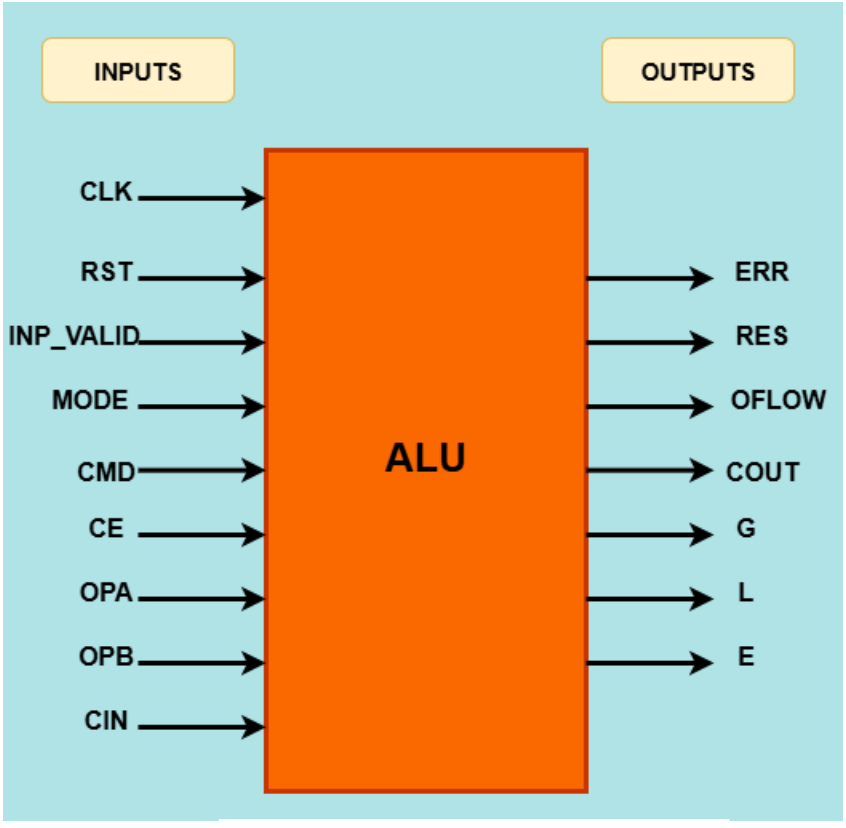


FIGURE 1 : ALU block diagram

Inputs:

Signal Name	Type	Description
MODE	Input	Mode selection signal - determines ALU operation mode
OPA	Input	Operand A (Parameterized)
OPB	Input	Operand B(Parameterized)
CMD	Input	Command signal - specifies the specific ALU operation
INP_VALID	Input	Input valid signal - indicates when input data is valid
CIN	Input	Carry In - input carry for arithmetic operations

Outputs

Signal Name	Type	Description
ERR	Output	Error signal - indicates if an error occurred during operation
RES	Output	Result - the output result of the ALU operation
OFLOW	Output	Overflow - indicates arithmetic overflow condition
COUT	Output	Carry Out - output carry from arithmetic operations
G	Output	Greater than - comparison result flag
L	Output	Less than - comparison result flag
E	Output	Equal - comparison result flag

CHAPTER 2
TESTBENCH ARCHITECTURE AND METHODOLOGY

2.1 Verification Architecture:

The below figure is the general testbench architecture architecture that is followed while writing systemverilog tesbench.

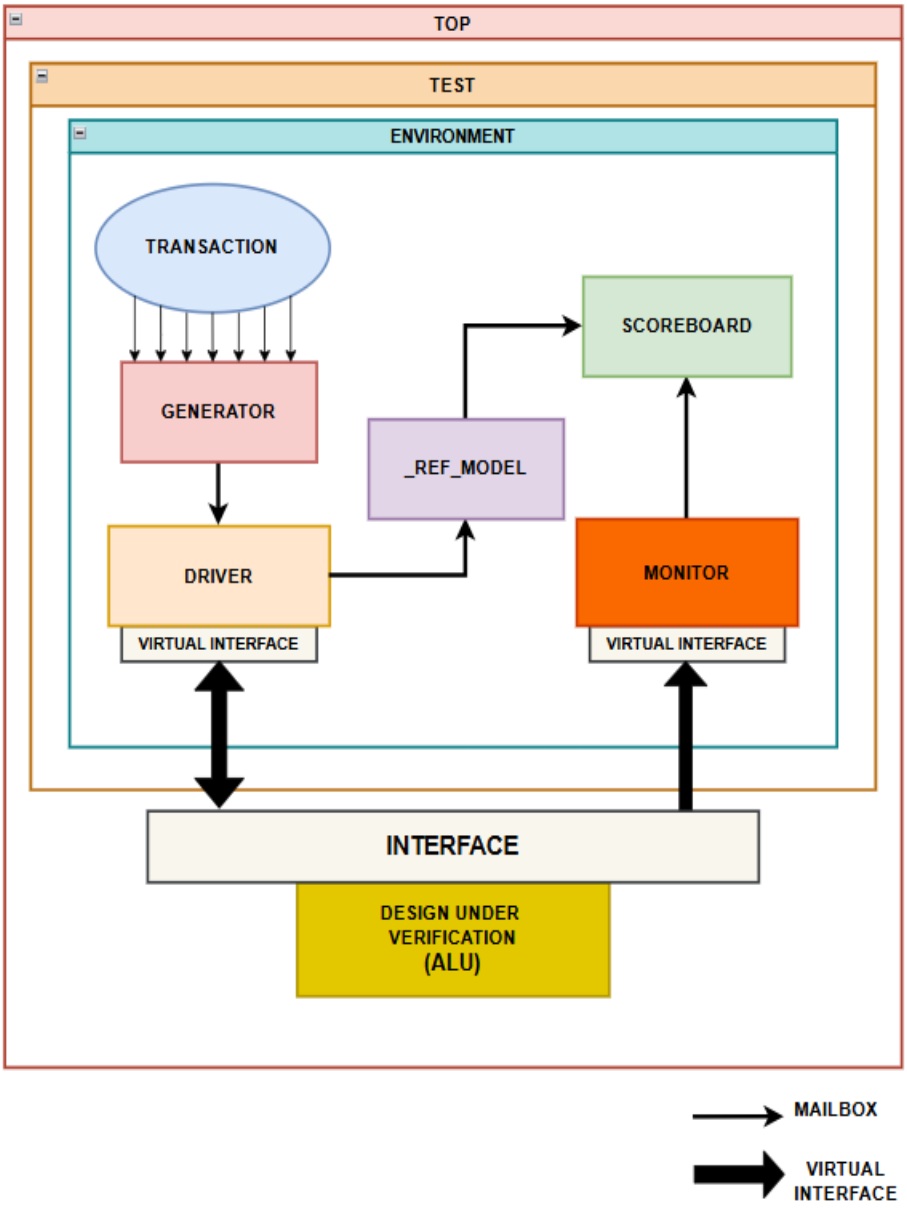


FIGURE 2.1 : Verification architecture

2.2 Verification Architecture for ALU:

● Below is the testbench architecture of ALU.

● The upper most module is TOP, which has interface, DUT(design under test/verification) and TEST.

● TEST(alu_test) has ENVIRONMENT.

● ENVIRONMENT has transaction, generator, driver, monitor, reference model, scoreboard. Driver and monitor use virtual interface to interact with the design.

● Mailboxes were used for the transfer of data between the different blocks.

mbx_gd : mailbox between generator and driver

mbx_dr : mailbox between driver and reference model

mbx_rs : mailbox between reference model and scoreboard

mbx_ms : mailbox between monitor and scoreboard

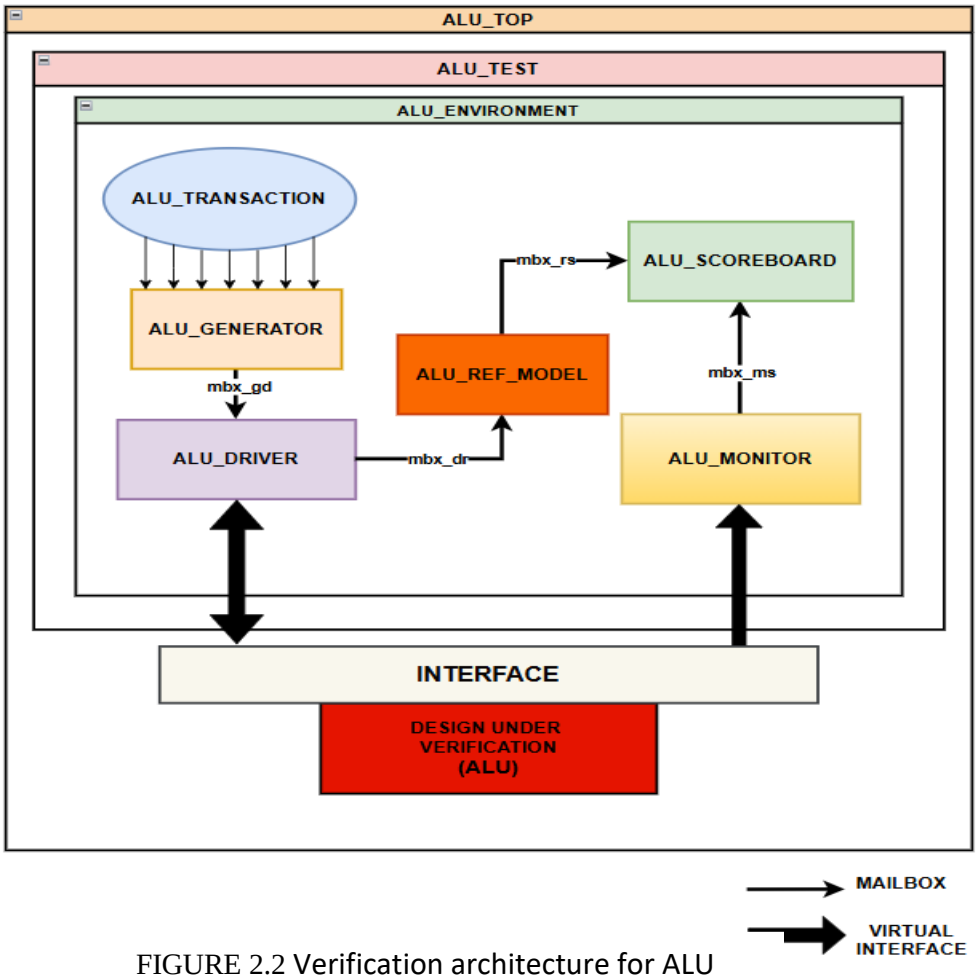


FIGURE 2.2 Verification architecture for ALU

2.3 FLOW CHART OF SV COMPONENTS :

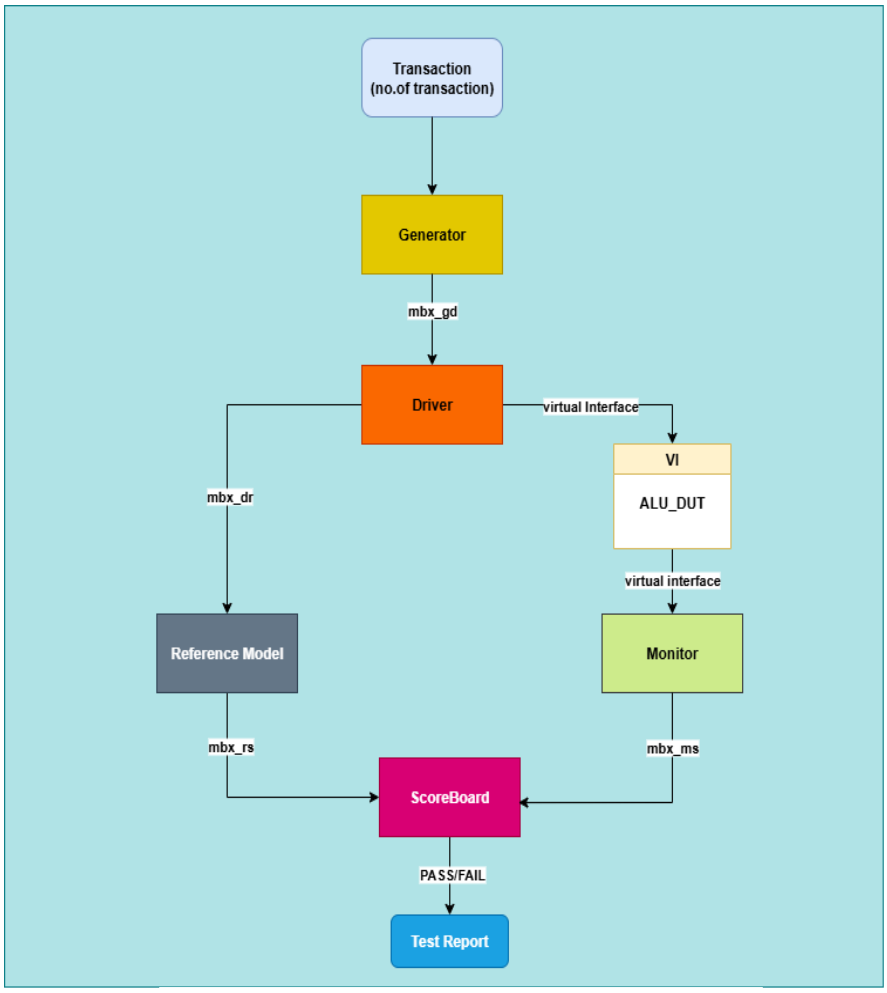


FIGURE 2.3 Flow chart for SV components

2.3.1 INTERFACE

The interface component in SystemVerilog serves as a bridge between the DUT and testbench components. It simplifies signal management and improves the modularity of the verification environment.

- It groups all DUT-related signals in one place, making the design cleaner.This reduces connection complexity and avoids repetitive signal declarations.
- It enables communication between components like driver and monitor.Both access the same interface instance, ensuring synchronized signal handling.

- It supports virtual interfaces, which allow flexible connections at runtime.This helps in reusing the same testbench across different configurations or DUTs.
- It improves reusability and scalability of the verification architecture.Interfaces help maintain modular, parameterizable, and easy-to-update code.

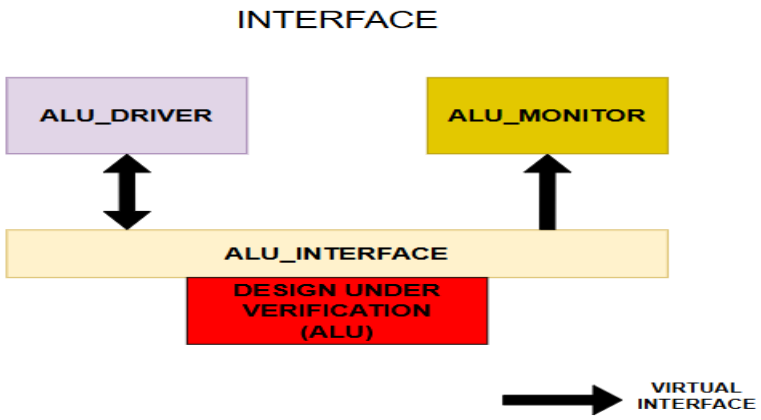


FIGURE 2.3.1 ALU interface component

2.3.2 TRANSACTION

The transaction component represents a single data item or operation to be applied to the DUT during simulation. It encapsulates both inputs and expected outputs, making stimulus generation and checking more structured.

- It contains randomized inputs like OPA, OPB, MODE, CMD, CE, and INP_VALID to generate varied scenarios.These inputs are constrained to maintain valid and meaningful ALU operations.

- The outputs (RES, COUT, OFLOW, G, L, E) store results captured from the DUT after execution.This allows for comparison against reference models or expected behavior.

- The constraint block guides the randomization to ensure only legal combinations are generated.This prevents invalid input conditions and helps in focused testing.

- The copy() method enables deep copying of transactions between components.It is essential for isolating data flow between generator, driver, and scoreboard.

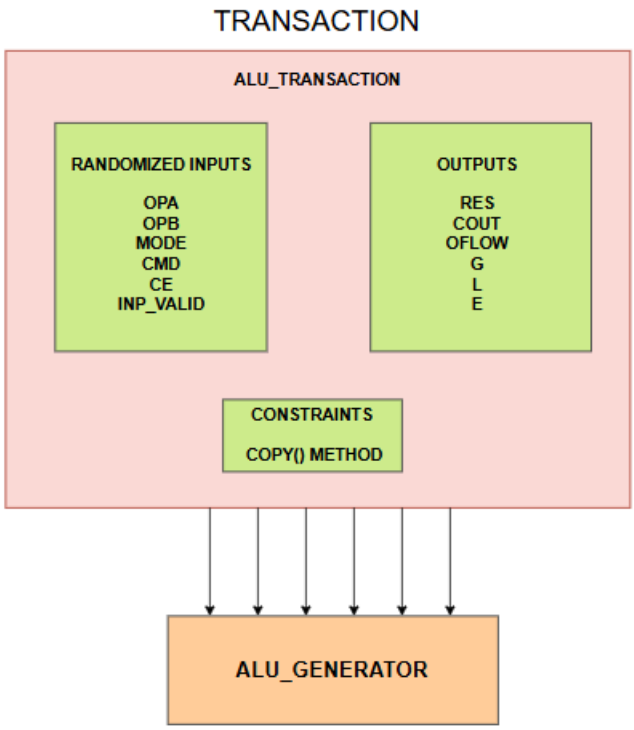


FIGURE 2.3.2 ALU transaction component

2.3.3 GENERATOR

The generator component is responsible for creating randomized transactions and supplying them to the driver for execution. It acts as the stimulus source in the verification environment.

- It uses the ALU_TRANSACTION class to create input scenarios with varied and constrained values.This helps simulate different functional cases of the ALU.
- The ALU_GENERATOR randomizes the transaction and prepares it for the test.It ensures legal and valid inputs are generated before sending them forward.
- A mailbox (mbx_gd) is used to transfer the transaction from generator to driver.This decouples the generation and driving processes, supporting synchronization.

- The generator can support various test strategies like directed, constrained-random, or functional coverage-driven tests.This flexibility helps improve test coverage and uncover corner cases.

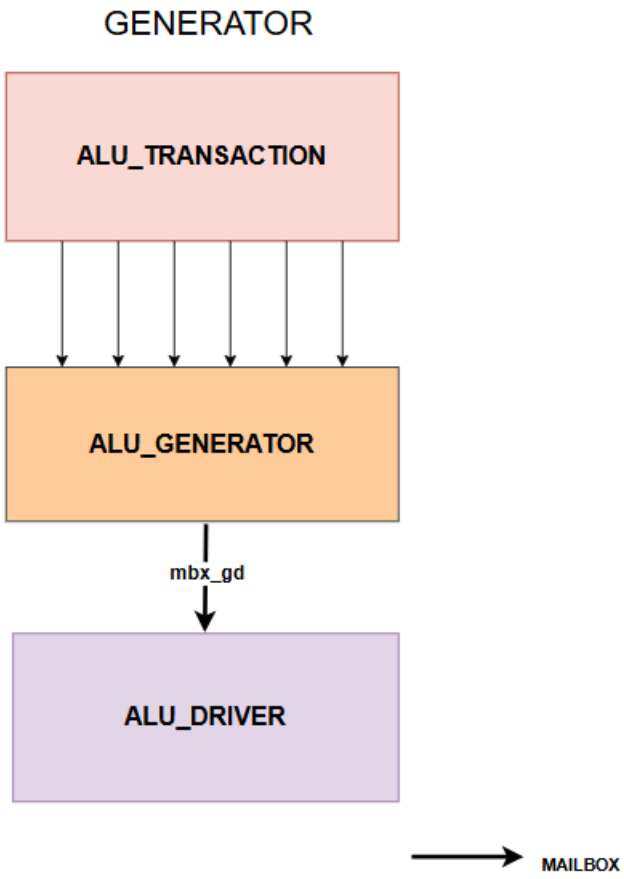


FIGURE 2.3.3 ALU generator component

2.3.4 DRIVER

The driver component is responsible for applying the generated input transactions to the DUT via the interface. It serves as the active element that drives stimulus into the design.

- It receives transactions from the ALU_GENERATOR through a mailbox (mbx_gd).This enables synchronization between transaction generation and driving.
- The ALU_DRIVER interprets the transaction and drives the signals onto the DUT via ALU_INTERFACE.A virtual interface is used to connect to the DUT signals indirectly.

- After driving, the same transaction is forwarded to the ALU_REFERENCE_MODEL using mbx_dr.This enables the reference model to predict expected outputs for scoreboard comparison.
- The driver ensures accurate timing and protocol adherence when applying inputs.It is crucial for reproducing real hardware behavior in a controlled simulation.

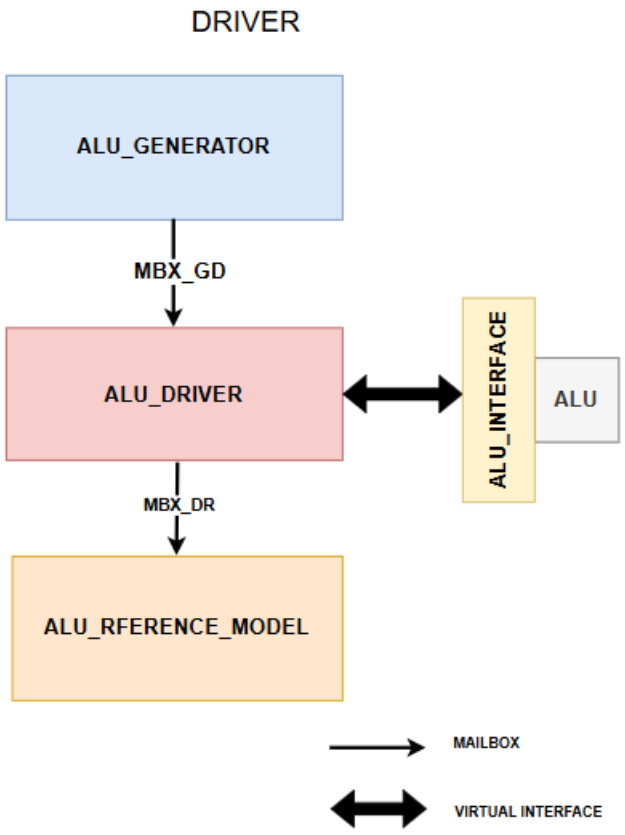


FIGURE 2.3.4 ALU driver component

2.3.5 MONITOR

The monitor component is a passive element in the testbench that observes DUT outputs and sends them for checking. It ensures accurate response tracking without influencing the simulation.

- The ALU_MONITOR connects to the DUT using a virtual interface.It continuously samples DUT outputs through the shared ALU_INTERFACE.

- It extracts results like RES, COUT, OFLOW, G, L, and E during simulation.These observed values are formed into a transaction structure.
- The captured data is sent to the ALU_SCOREBOARD via a mailbox (mbx_ms).This enables comparison with predicted values from the reference model.
- Being passive, the monitor ensures non-intrusive observation of DUT behavior.It plays a key role in functional coverage and output verification.

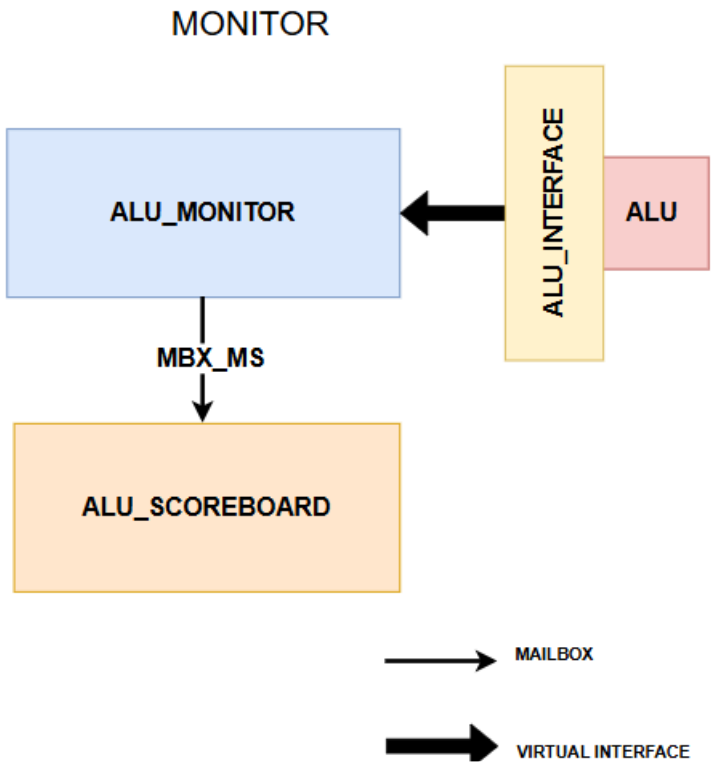


FIGURE 2.3.5 ALU monitor component

2.3.6 REFERENCE MODEL

The reference model component predicts the expected outputs based on the same inputs sent to the DUT. It acts as the golden reference for comparing DUT behavior.

- The ALU_REFERENCE_MODEL receives transactions from the driver through the mbx_dr mailbox.These transactions include the exact inputs driven to the DUT.
- It computes the expected output using the same ALU logic implemented in high-level code.This ensures functional correctness by mirroring the DUT's behavior.

- The predicted result is then sent to the ALU_SCOREBOARD using mbx_rs.This allows direct comparison with DUT results captured by the monitor.
- The reference model isolates functional bugs by providing an ideal output path.It is crucial for verifying the DUT output accuracy under various scenarios.

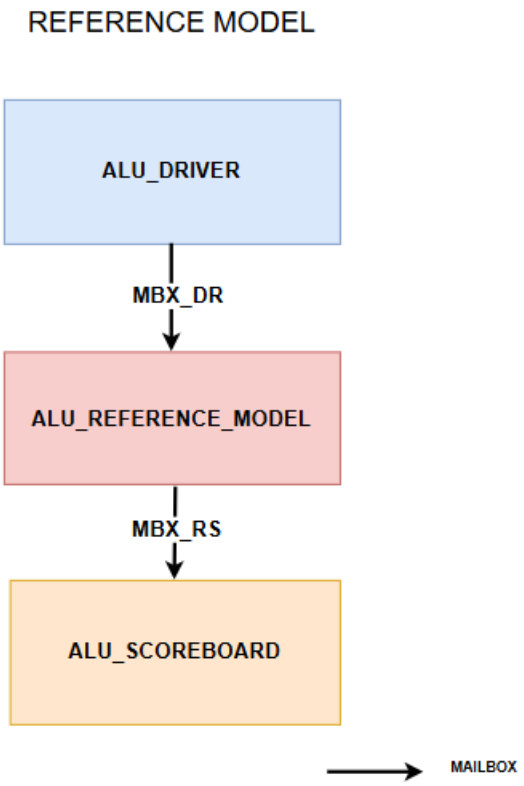


FIGURE 2.3.6 ALU reference model

2.3.7 SCOREBOARD

The scoreboard component is responsible for comparing the DUT outputs with the expected results from the reference model. It ensures functional correctness and flags mismatches during simulation.

- The ALU_SCOREBOARD receives predicted outputs from the reference model via mbx_rs.These represent the golden results computed based on input transactions.
- It also collects actual DUT outputs from the monitor through mbx_ms.These are the observed values after the transaction is applied to the DUT.

- The scoreboard compares both sets of data field-by-field for mismatches.If any difference is found, it logs an error or assertion failure.
- It helps in verifying DUT accuracy, detecting bugs, and improving coverage.The scoreboard is essential for automated result checking and debug efficiency.

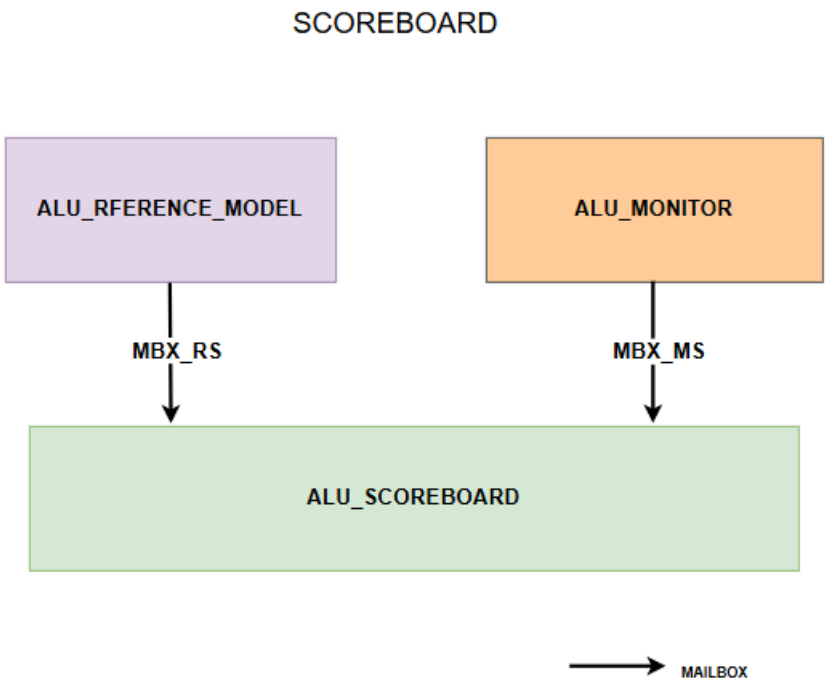


FIGURE 2.3.7 ALU scoreboard component

2.3.8 ENVIRONMENT

The environment component integrates all verification components into a single container called ALU_ENVIRONMENT. It orchestrates data flow, connections, and execution in the testbench.

- It encapsulates all subcomponents like generator, driver, monitor, reference model, scoreboard, and transaction class.This modular structure makes the testbench reusable and maintainable.
- Mailboxes (mbx_gd, mbx_dr, mbx_rs, mbx_ms) are used to communicate between components.They ensure orderly data transfer and synchronization during simulation.

- Each component performs its specific task and passes data forward for processing or checking.This separation of concerns improves clarity and debugging.

- The environment provides the foundation for building tests by connecting everything in one place.It enables full automation of stimulus generation, DUT driving, monitoring, and result checking.

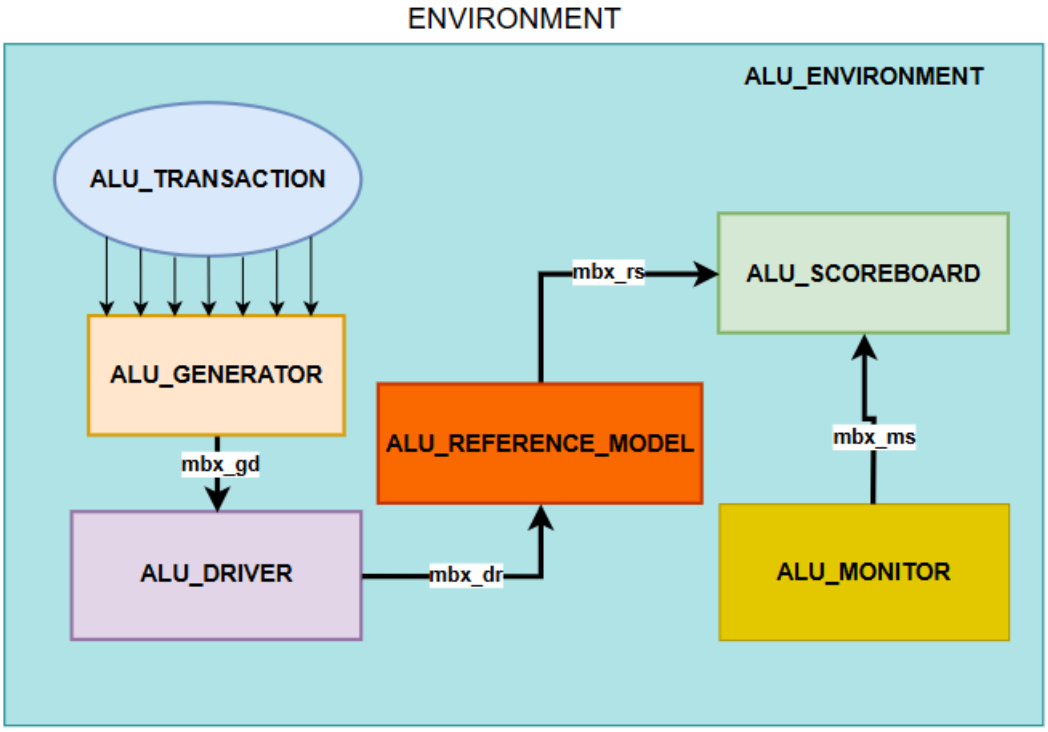


FIGURE 2.3.8 ALU environment component

2.3.9 TEST

The test component is the top-level module that controls and configures the entire verification process. It creates the environment, starts simulation activities, and defines test scenarios.

- It instantiates the ALU_ENVIRONMENT and connects all lower-level components.This includes setting up mailboxes, virtual interfaces, and configuration objects if needed.

- The test triggers phases like build, run, and report to coordinate simulation flow.In the run phase, it starts the generator and controls simulation duration.

- Different tests can extend the base test to apply varied constraints or stimulus patterns.This enables regression testing with multiple coverage-driven or directed test cases.
- The test ensures that the environment behaves correctly and gathers coverage or pass/fail status.It acts as the main control unit to manage and execute the entire verification plan.

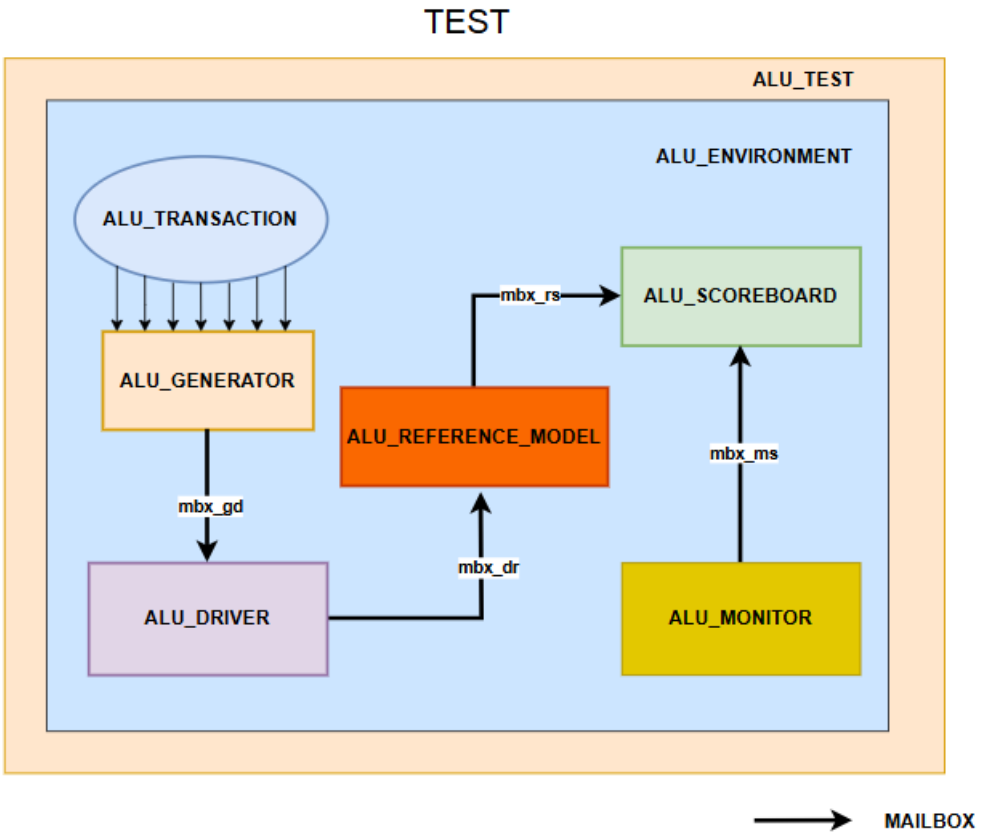


FIGURE 2.3.9 ALU test component

2.3.10 TOP

The top component is the highest level in the verification hierarchy that connects the DUT and the testbench. It is typically written as a Verilog module and is used to simulate the entire system.

- It instantiates the Design Under Test (DUT) and the testbench test class.This ensures both functional hardware and verification logic are present in simulation.

- The top module also includes the interface and binds it to the DUT and testbench using virtual interfaces.This allows components like driver and monitor to interact with DUT signals.
- It handles clock and reset generation, and any other signal initialization needed before starting the test.This setup ensures the DUT operates in a realistic simulation environment.

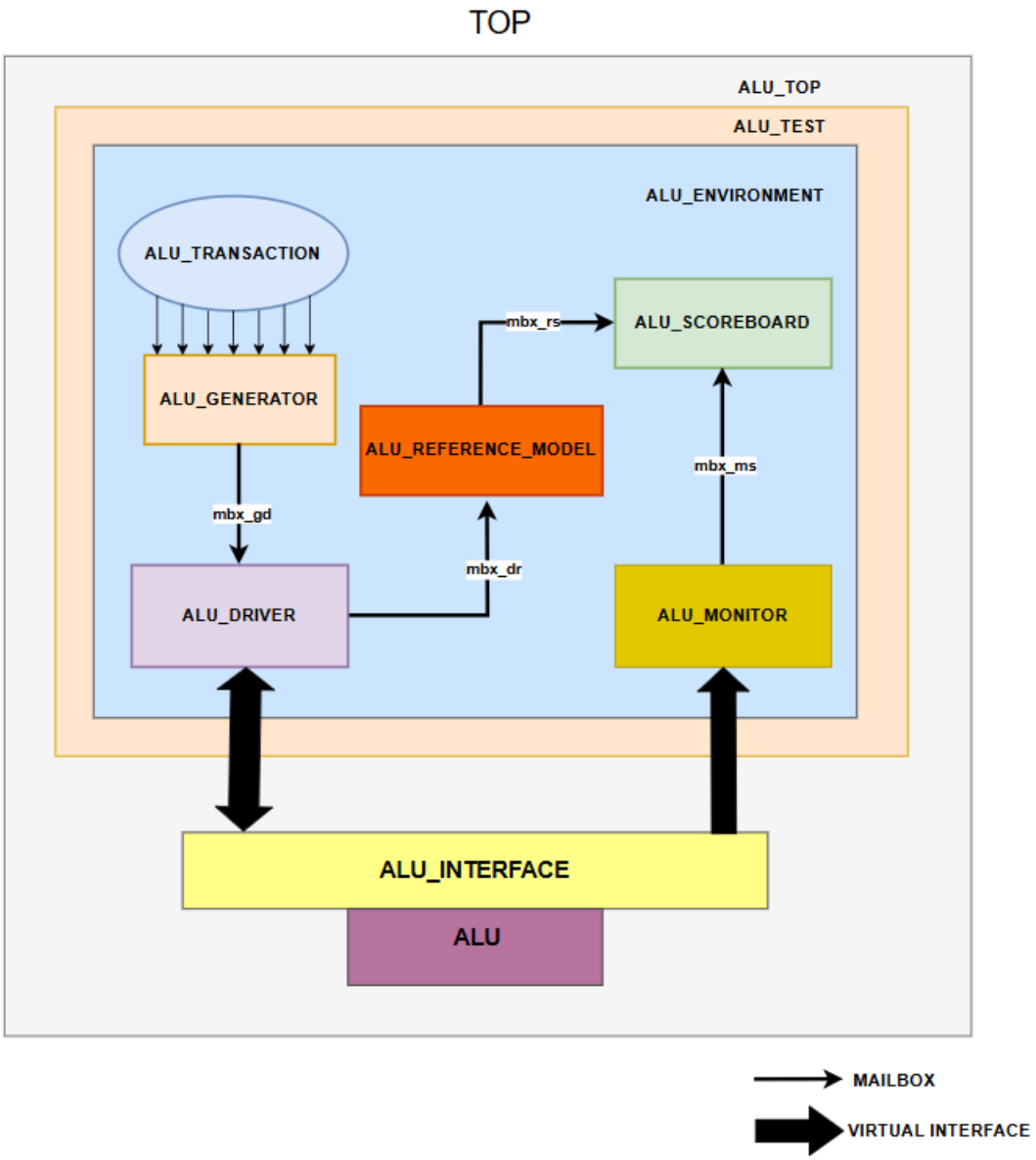


FIGURE 2.3.10 ALU top component

TEST PLAN

- TEST PLAN: https://docs.google.com/spreadsheets/d/1Hb9BDjKUWGku7YgYGu4m2-MxOdKwIwqMFIhu1x-F5Y4/edit?gid=0#gid=0

- COVERAGE PLAN : https://docs.google.com/spreadsheets/d/1Hb9BDjKUWGku7YgYGu4m2-MxOdKwIwqMFIhu1x-F5Y4/edit?gid=820551267#gid=820551267

- ASSERTION PLAN : https://docs.google.com/spreadsheets/d/1Hb9BDjKUWGku7YgYGu4m2-MxOdKwIwqMFIhu1x-F5Y4/edit?gid=1490673837#gid=1490673837

CHAPTER 3

VERIFICATION RESULTS AND ANALYSIS

3.1 ERRORS IN THE DUT

SPECIFICATION	BUGS DESCRIPTION
ADD_IN	In ADD_IN operation, when OPA = OPB and CIN = 1, Then,there is no COUT in this condition.
SUB_IN	In SUB_IN operation, when OPA = OPB and CIN = 1, result is -1. There is no overflow in this condition.
INC_A	In INC_A operation (CMD = 4'b0100), RES = OPA is assigned without increment.This is a bug , it should be RES = OPA + 1 to perform increment correctly.
INC_B	As per spec,CMD 6 is INC_B but in design it is DEC_B
DEC_B	As per spec,CMD 7 is DEC_B but in design it is INC_B
OR	As per spec,CMD 2 is OR operation,but in design it is logical AND operation
SHR1_A	As per spec ,CMD=8 ,MODE=0, shift right operation,but in design its res=opa
SHR1_B	As per spec ,CMD=10 ,MODE=0, shift right operation,but in design it’s a left shift
ROR	In CMD = 4'b1101, when any of oprd2[4] to oprd2[7] are high, ERR is set to 0.This is a bug — as per specification, ERR should be set to 1 to indicate error
ADD_IN	In ADD_IN operation, the design performs addition but does not assign COUT. This is a bug — COUT should reflect the carry-out from the MSB of the sum.
MUL_S	In CMD = 4'b1010, the design performs RES = (oprd1 << 1) - oprd2, which is incorrect. It should perform multiplication — RES = (oprd1 << 1) * oprd2 as per specification.

INP_VALID	When INP_VALID = 2'b00, no operation should occur and ERR should be set to 1. In the current design, ERR is not asserted, which violates input validity
CLK WAITING	During the 16-cycle wait state, if INP_VALID transitions from '01' to '10' in the next cycle, the design incorrectly takes both inputs as valid and performs the operation, which leads to erroneous output

3.2 CODE COVERAGE

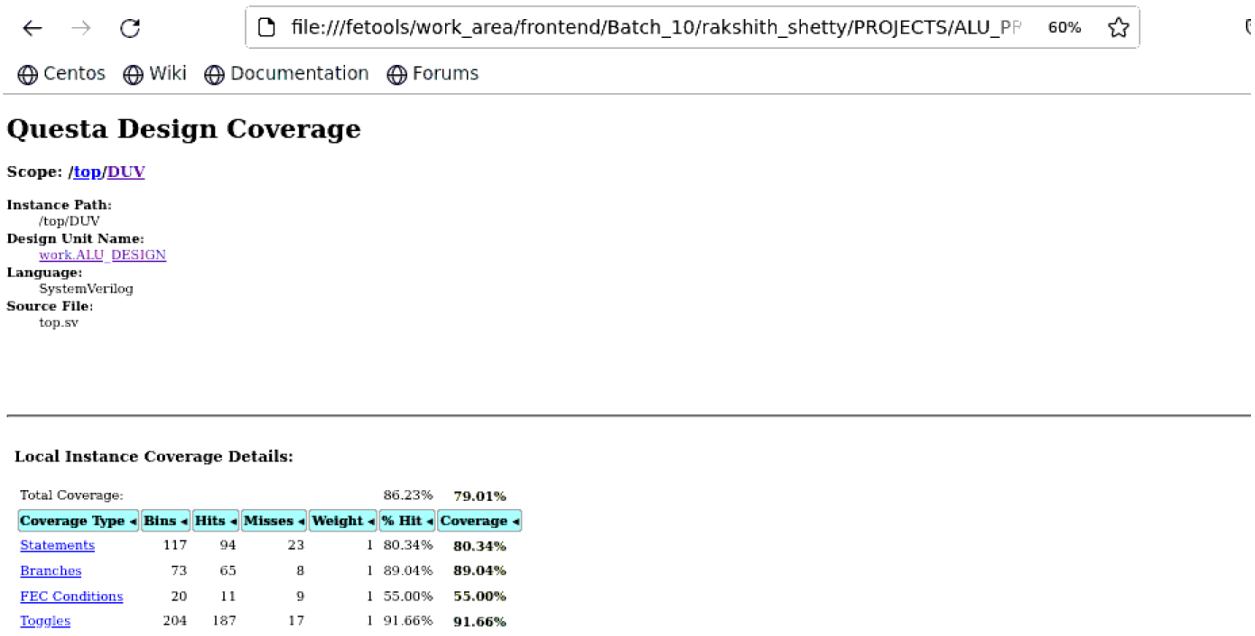


FIGURE 3.1 Code coverage

3.3 INPUT FUNCTIONAL COVERAGE

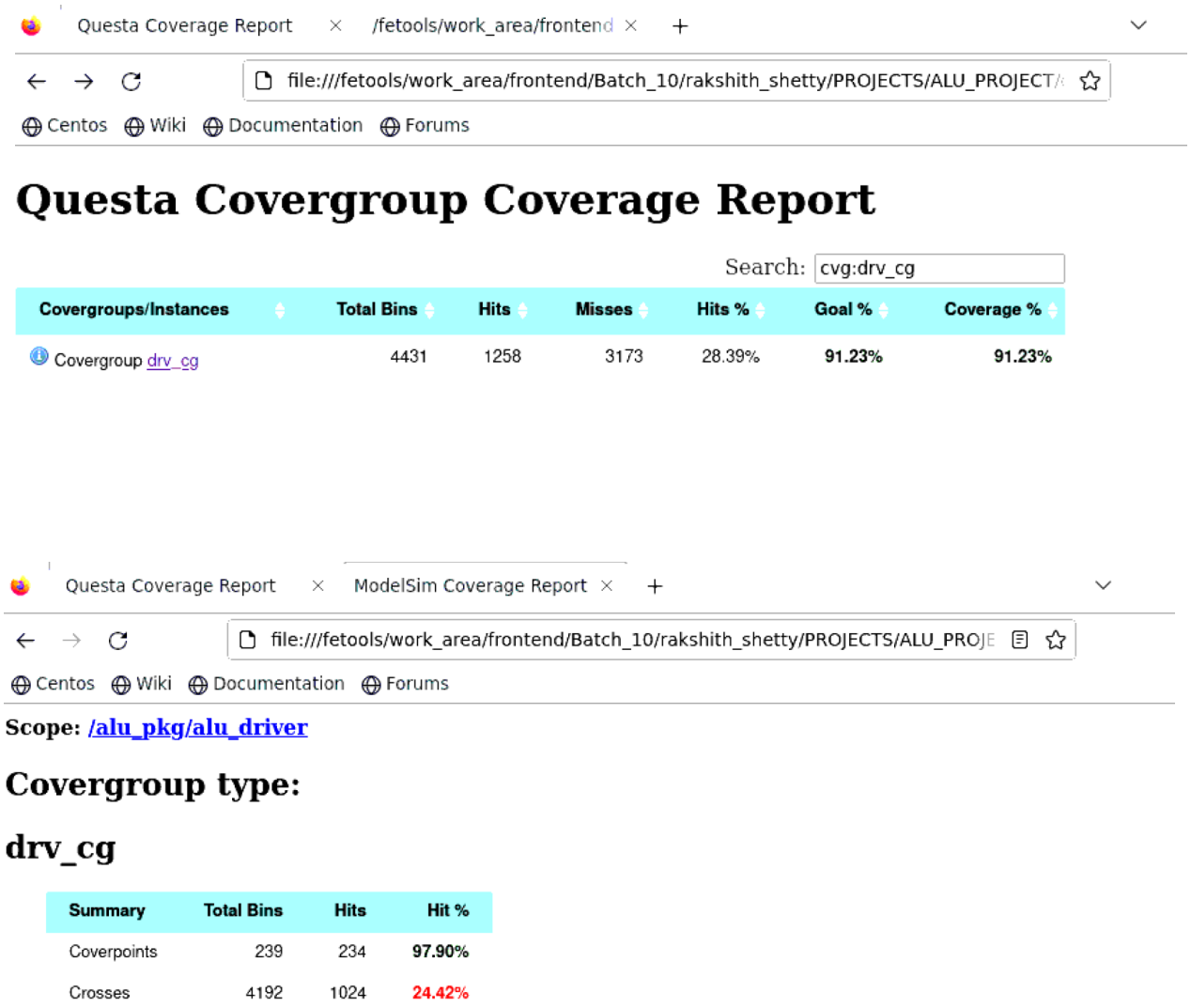


FIGURE 3.2 Input functional coverage

3.4 OUTPUT FUNCTIONAL COVERAGE

Covergroup type:

cg_monitor

Summary	Total Bins	Hits	Hit %
Coverpoints	264	245	92.80%
Crosses	0	0	0.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CARR_OUT	2	2	0	100.00%	100.00%	100.00%
EQUAL	1	0	1	0.00%	0.00%	0.00%
ERROR	1	0	1	0.00%	0.00%	0.00%
GREATER	1	1	0	100.00%	100.00%	100.00%
LESSER	1	1	0	100.00%	100.00%	100.00%
OVERFLOW	2	2	0	100.00%	100.00%	100.00%
RESULT_CHECK	256	239	17	93.35%	93.35%	93.35%

FIGURE 3.3 Output functional coverage

3.5 ASSERTION COVERAGE

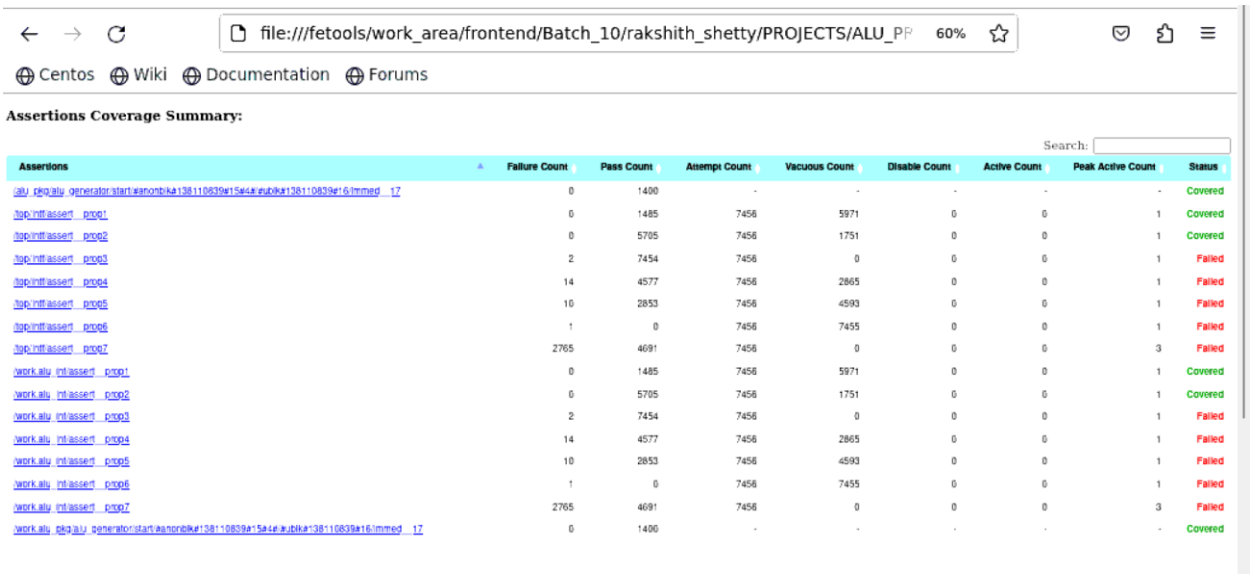
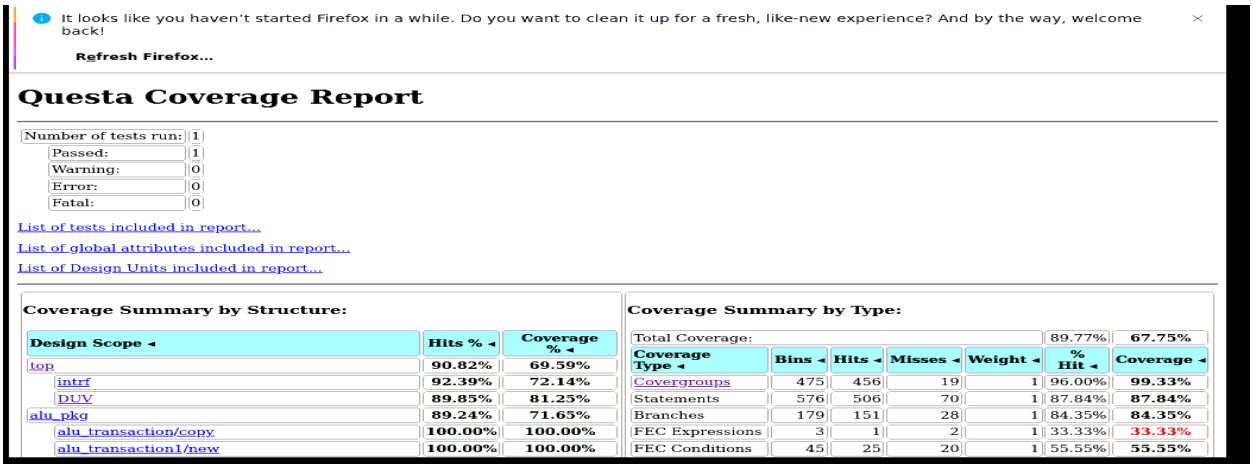


FIGURE 3.4 Assertion coverage

3.6 OVERALL COVERAGE



It looks like you haven't started Firefox in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welcome back!

Refresh Firefox...

Questa Coverage Report

Number of tests run:	1
Passed:	1
Warning:	0
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 %
top	90.82%	69.59%
intrf	92.39%	72.14%
DUV	89.85%	81.25%
alu_pkg	89.24%	71.65%
alu_transaction/copy	100.00%	100.00%
alu_transaction1/new	100.00%	100.00%

Coverage Summary by Type:

Total Coverage:					89.77%	67.75%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Covergroups	475	456	19	1	96.00%	99.33%
Statements	576	506	70	1	87.84%	87.84%
Branches	179	151	28	1	84.35%	84.35%
FEC Expressions	3	1	2	1	33.33%	33.33%
FEC Conditions	45	25	20	1	55.55%	55.55%

FIGURE 3.5 Overall code coverage

3.7 OUTPUT WAVEFORM

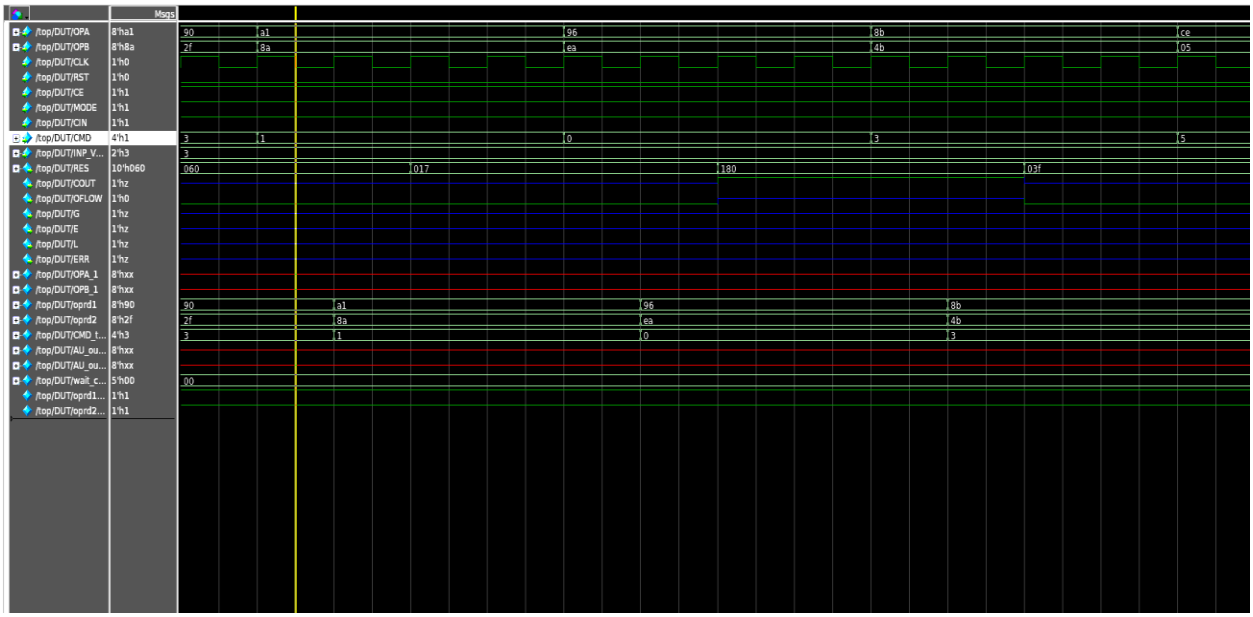


FIGURE 3.6 Output waveform