# ALU Verification Plan

By Nishanth Gowda CJ   EMP ID 6089

| SL.NO | CONTENTS | PG.NO |
|---|---|---|

# LIST OF FIGURES

# CHAPTER 1

# PROJECT OVERVIEW AND SPECIFICATIONS

## 1.1 ALU introduction:

The Arithmetic Logic Unit (ALU) acts as the brain of any digital processing system, executing core computational functions with both arithmetic and logical operations. In this project, the ALU is implemented as a flexible, parameterized module, allowing customization of bit-widths to suit different design needs, making it ideal for various embedded or processor-based applications.

This ALU design stand out is its comprehensive functionality , it not only performs basic operations like addition, subtraction, and logical comparisons, but also includes features like bitwise rotation, input validation, and error detection. Designed for synchronous operation, it uses proper clock and reset logic to ensure stable performance in real-time digital system

## 1.2 Advantages of ALU:

- **Customizable Design**
  With parameterization, the bit-width can be adjusted to meet specific system requirements, enhancing flexibility and reuse.

- **Supports Both Arithmetic and Logical Operations**
  Capable of executing a wide range of tasks, from simple addition to bitwise logical functions, making it a core component in digital systems.

- **Efficient Computation**
  Performs operations quickly, often within a single clock cycle, ensuring fast data processing in real-time applications.

- **Synchronous Operation**
  Proper integration with clock and reset signals ensures timing reliability and predictable behavior in synchronous digital systems.

- **Input Validation and Error Detection**

  Includes mechanisms to check input validity and detect illegal or unexpected conditions, improving the overall robustness of the system.

- **Scalable and Reusable**

  The modular and parameterized nature allows the ALU to be easily reused across different projects with varying complexity.

## 1.3 Disadvantages of ALU:

- **Limited Instruction Set**

  The ALU can only perform operations that are explicitly defined in the design. Complex functions like floating-point arithmetic require separate units.
- **No Memory Capability**

  The ALU operates purely on inputs and doesn't store data; it must rely on external registers or memory for data storage.
- **Increased Complexity with More Bits**

  As the data width increases (e.g., from 4-bit to 32-bit), the design becomes more complex, potentially affecting performance and power consumption.
- **Fixed Operation Control**

  The set of operations is usually hardcoded; modifying them later requires changes in the HDL code and re-synthesis.
- **Propagation Delay in Certain Operations**

  Some operations, especially those involving carry propagation (like addition or multiplication), can introduce delay depending on how the ALU is implemented.

## 1.4 Use cases of ALU:

- **Microprocessors and Microcontrollers**

  ALUs are the core computational units in processors. They handle arithmetic calculations, logical decisions, and flag generation for instruction execution.

- **Digital Signal Processing (DSP)**

  ALUs perform operations like addition, multiplication, and shifting — essential in filtering, encoding, and decoding signals. Efficient ALU design enhances real-time processing in audio, video, and communication systems.

- **Embedded Systems**

  Used in smart devices (IoT, sensors, controllers) for decision-making and control logic. Performs fast calculations in real-time automation tasks.

- **Graphics Processing Units (GPUs)**

  Specialized ALUs in GPUs perform parallel arithmetic operations required for rendering graphics, transformations, and shading

- **Cryptographic Algorithms**

  ALUs are used in bitwise logic and modular arithmetic — essential in implementing encryption and decryption logic in secure systems.

- **Control Units in Robotics**

  ALUs assist in decision-making based on sensor inputs and feedback, enabling autonomous movement and intelligent control.

- **Scientific and Engineering Calculations**

High-performance computing applications use powerful ALUs to perform complex mathematical operations with speed and accuracy.

- **Real-Time Operating Systems**

  ALUs work with schedulers and system cores to evaluate timing conditions and trigger tasks based on logic and arithmetic decision

## 1.5 Project Overview of ALU:

This project involves the design and implementation of a parameterized ALU capable of supporting a wide range of bit-widths and digital operations, making it suitable for both low-end embedded systems and high-performance processors.

- **Parameterized Design**

  Supports variable bit-widths (e.g., 16, 32, 64, 128 bits). Enables integration into systems with diverse computational needs, from simple microcontrollers to complex processors.

❖ **Dual Mode Operation**

Operates in two distinct modes:

- Arithmetic Mode (MODE = 1)
- Logical Mode (MODE = 0)

Each mode includes 14 unique operations, selected via a 4-bit command input.

❖ **Supported Operations**

- Arithmetic Mode: ADD, SUB, MUL, INC, DEC, etc.
- Logical Mode: AND, OR, XOR, NAND, NOR, NOT, bitwise rotations, etc.

❖ **Input Control Mechanism**

INP_VALID signal ensures inputs are synchronized, especially important in asynchronous or pipelined systems.

Timeout logic prevents indefinite waiting for missing inputs, improving system responsiveness.

❖ **Comprehensive Status Reporting**

Generates flags for:

- Overflow detection
- Carry-out monitoring
- Comparison results (Equal, Greater Than, Less Than)

Useful for decision-making in CPUs, control units, and FSMs.

❖ **Built-In Error Handling**

Detects and flags invalid operations (e.g., improper bit-rotation commands or out-of-range inputs).

Enhances debuggability and system integration by avoiding silent failures.

❖ **High Reusability and Modularity**

Designed to be easily integrated into larger digital systems.

Modular and scalable for use in both educational projects and industry-level SoC architectures.

## 1.6 Design Features:

- **Parameterized Bit-Width Support**

  Supports multiple data widths: 16-bit, 32-bit, 64-bit, and 128-bit

  Enhances reusability for low-end to high-end applications

- **Dual Mode Operation**

  MODE = 1: Arithmetic operations, MODE = 0: Logical operations

  Clear separation between arithmetic and logic functionality

- **4-bit Command Selection System**

  Enables up to 16 distinct operations

  Commands are reused efficiently based on the selected mode

- **Input Validity Handling (INP_VALID)**

  Ensures operations execute only when valid inputs are received

  Useful in pipelined or asynchronous environments

- **Timeout Mechanism**

  Prevents the system from waiting indefinitely for inputs

  Improves reliability in unpredictable data arrival scenarios

- **Status Flags for Control Logic**

  Overflow flag

  Carry-out flag

  Comparison outputs: Greater than, Less than, Equal

  Helps CPU or controller units in decision-making (e.g., branching)

- **Error Detection and Handling**

  Flags invalid operations (e.g., undefined rotation patterns)

  Improves debugging and integration in larger systems

- **Modular and Scalable Design**

  Easily extendable for more operations or features

  Clean separation between datapath and control logic

❖ **Support for Complex Operations**

Includes rotate left/right, shift, AND/OR/XOR, etc.

Enables broader use cases like encryption, CRC, and encoding

❖ **Robust Output Interface**

Generates output data (RESULT)

Sets all relevant status signals on completion

## 1.7 Design Limitation:

❖ **Fixed Timeout Duration**

The 16-cycle timeout is hardcoded in the design.

Modifying the wait period requires RTL changes, reducing flexibility for integration in different systems.

❖ **Mode-Based Command Overlap**

The same command code performs different operations depending on the mode (arithmetic vs. logical).

Increases the risk of software misconfiguration or unintended operation

❖ **Single Error Flag for Multiple Issues**

Only one `ERR` signal is used to indicate all types of faults (e.g., timeout, invalid command, input errors).

Debugging becomes difficult as the source of the error cannot be distinguished directly.

❖ **Limited Rotate Operation Range**

Rotate Left and Rotate Right operations support a maximum of 8 positions.

This limits applicability when scaling to wider data paths (e.g., 16-bit or 32-bit systems).

❖ **Fixed Operand Selection on Timeout**

In case of timeout, the ALU automatically chooses the latest operand.

This behavior might not align with expected logic in systems that require previous operand retention.

❖ **Lack of Pipelining**

The ALU executes one operation at a time without pipelining.

This can be a bottleneck in high-speed or parallel processing environments, limiting throughput

## 1.8 Design diagram with interface signals:

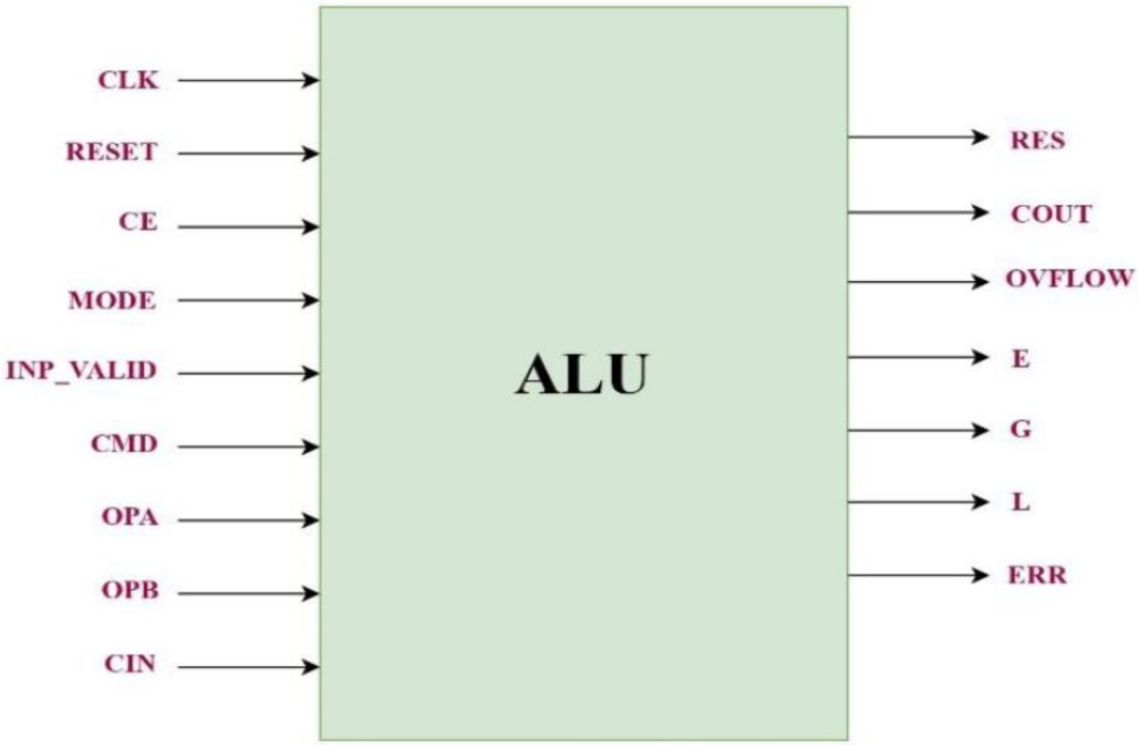



Figure 1.1 ALU block diagram

**Input ports:**

| Signal | Direction | Size(bits) | Description |
| --- | --- | --- | --- |
| INP_VALID | INPUT | 2 | Shows the Validity of the Operands (active high). MSB shows the validity of OPB and LSB shows the validity for OPA |
| MODE | INPUT | 1 | If the value is 1 the ALU is in Arithmetic Mode else it is in Logical Mode |
| CMD | INPUT | 4 | Commands for the Operation |
| OPA | INPUT | Parameterized | Operand A – first arithmetic/logic operand |
| OPB | INPUT | Parameterized | Operand B – second arithmetic/logic operand |

| Signal | Direction | Size(bits) | Description |
|---|---|---|---|
| CIN | INPUT | 1 | Carry In – input carry for arithmetic operations |

**Output ports:**

| **Signal** | **Direction** | **Size(bits)** | **Description** |
|---|---|---|---|
| **ERR** | OUTPUT | 1 | Active High Error Signal |
| **RES** | OUTPUT | Parameterized + 1 | Result of the instruction performed by the ALU |
| **OFLOW** | OUTPUT | 1 | Overflow – indicates arithmetic overflow condition |
| **COUT** | OUTPUT | 1 | Carry out signal, updated during Addition/Subtraction |
| **G** | OUTPUT | 1 | Comparator output which indicates that the value of OPA is greater than the value of OPB |
| **L** | OUTPUT | 1 | Comparator output which indicates that the value of OPA is less than the value of OPB |
| **E** | OUTPUT | 1 | Comparator output which indicates that the value of OPA is equal to the value of OPB |

# CHAPTER 2
# TESTBENCH ARCHITECTURE AND METHODOLOGY

## 2.1 Verification Architecture :

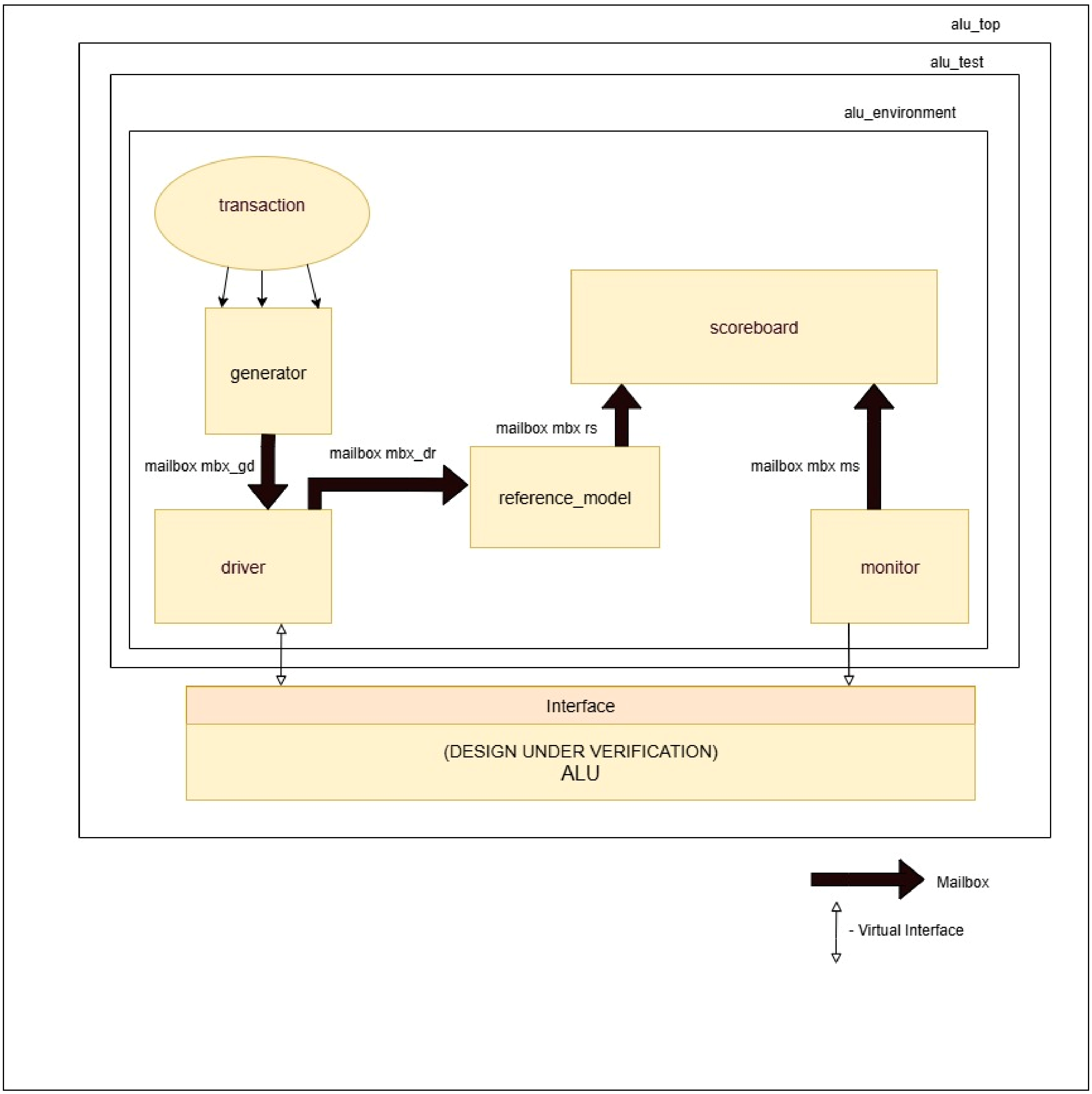



figure 2.1: verification architecture

# 2.2 Verification Architecture for ALU:

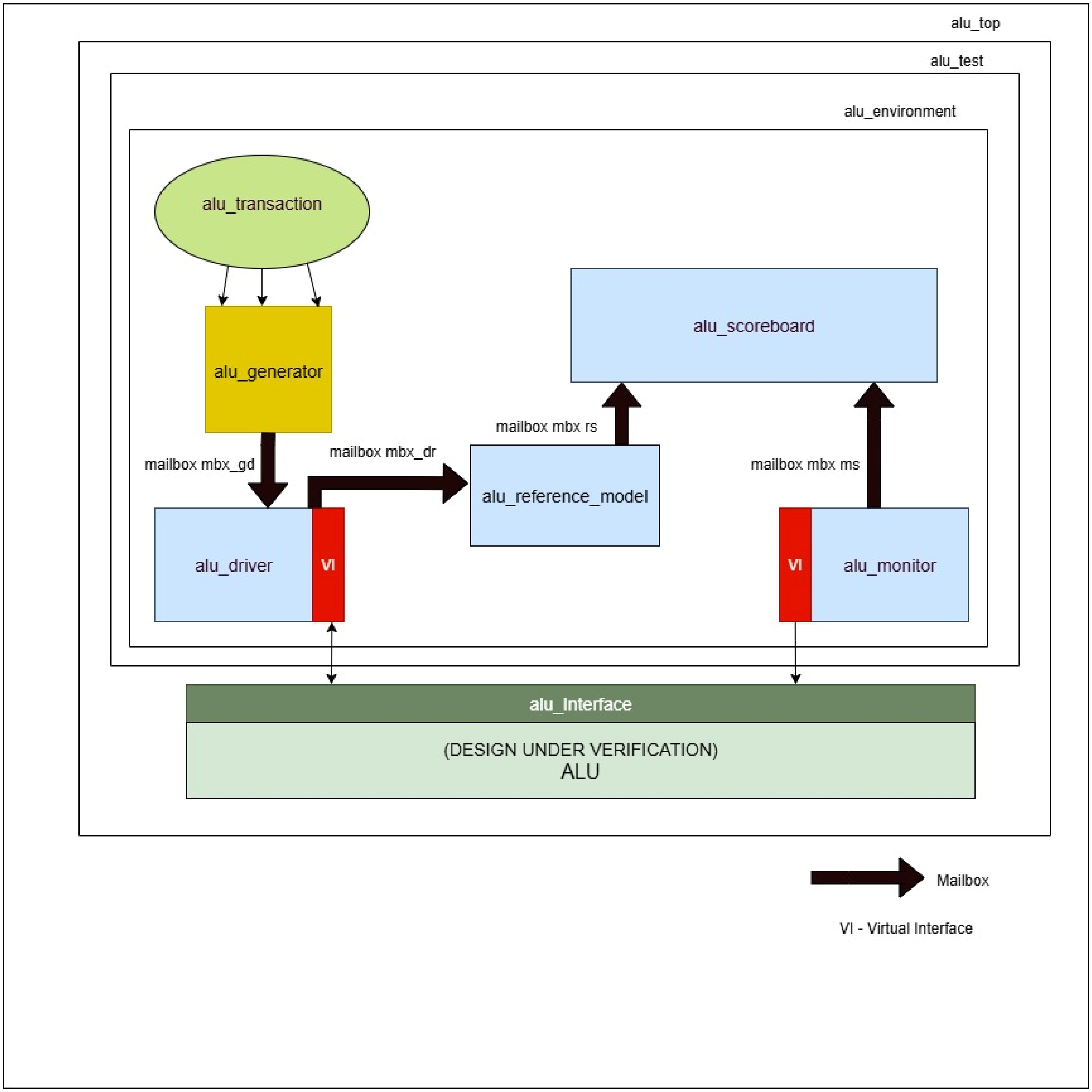



Figure 2.2 :verification architecture for ALU

## 2.3 FLOW CHART OF SV COMPONENTS :

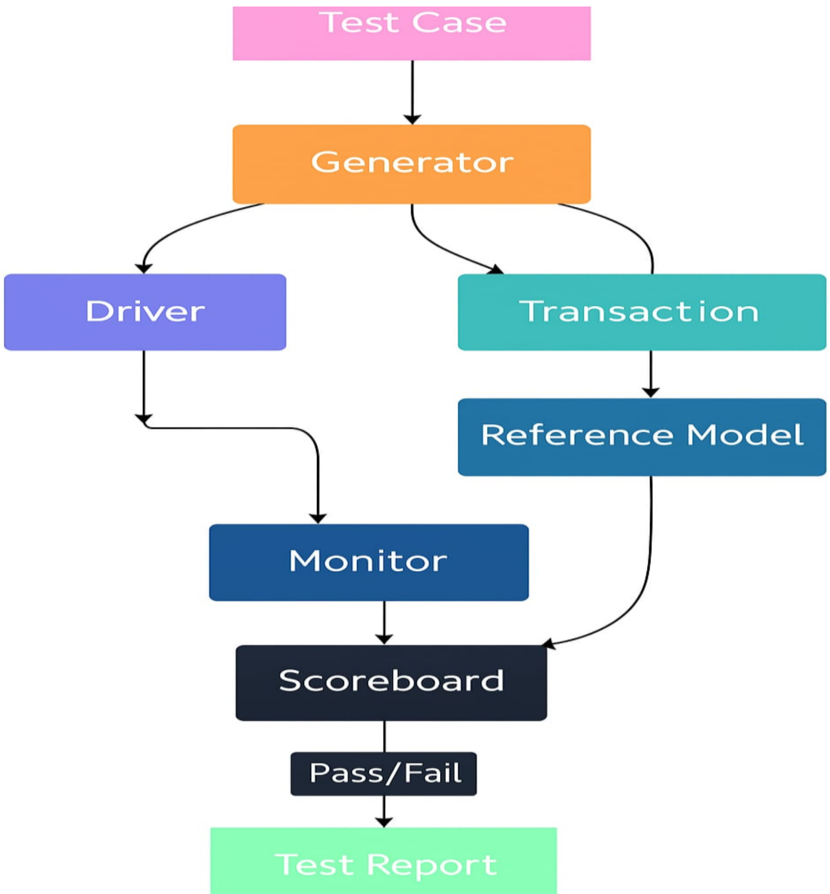



Figure 2.3 :flow chart for SV components

### 2.3.1 INTERFACE COMPONENT

alu_interface:

- A SystemVerilog interface that connects DUT ports to the testbench.
- Enables access to DUT signals in a clean and controlled way.
- The VI (Virtual Interface) is passed to the driver and monitor.

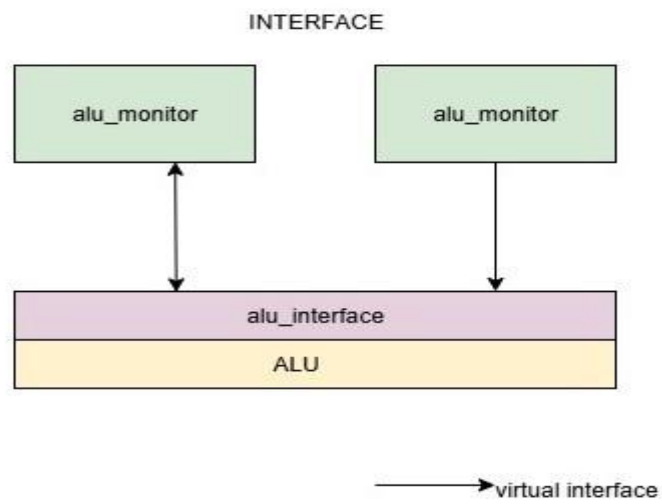



Figure 2.3.1: ALU interface component

### 2.3.2 TRANSACTION COMPONENT

alu_transaction :

- A class that defines the transaction or stimulus data structure.
- Contains all signal fields like:
    - CLK, RST, INP_VALID, MODE
    - CMD, CE, OP_A, OP_B, CIN, ERR
    - RES_OFLOW, COUT, G, E, L
- These represent the inputs/outputs used to drive the DUT.

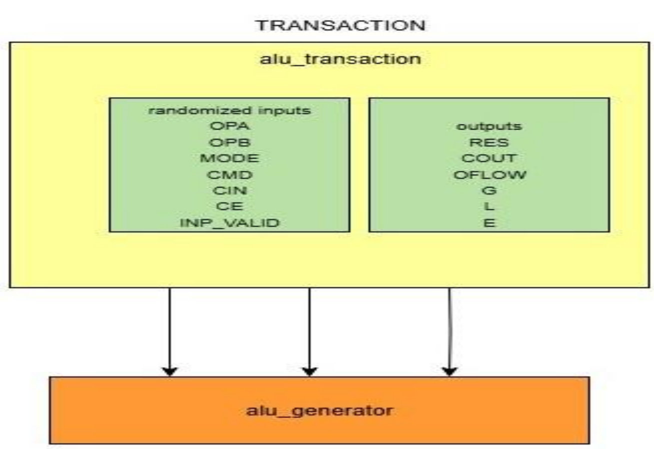



Figure 2.3.2: ALU transaction component

### 2.3.3 GENERATOR COMPONENT

alu_generator:

- Generates randomized or constrained stimulus (transaction objects).
- Sends transactions to alu_driver using mailbox.

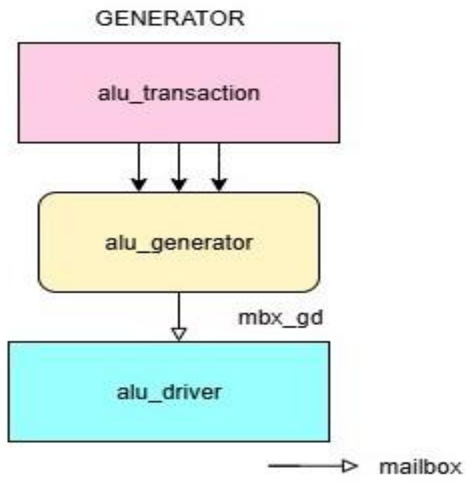



Figure 2.3.3: ALU generator component

### 2.3.4 DRIVER COMPONENT

alu_driver:

- Converts high-level transactions into pin-level signals.
- Drives those onto the DUT via the Virtual Interface (VI).
- Acts like a proxy between transaction world and DUT signal world.

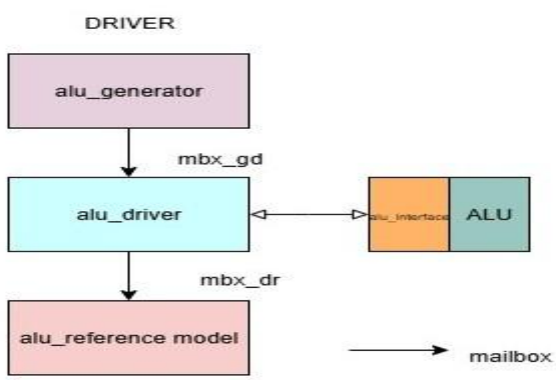



Figure 2.3.4: ALU driver component

### 2.3.5 MONITOR COMPONENT

alu_monitor:

- Monitors outputs from the DUT through the Virtual Interface (VI).
- Converts low-level DUT output signals back into high-level transaction objects.
- Sends these to the alu_scoreboard for checking.

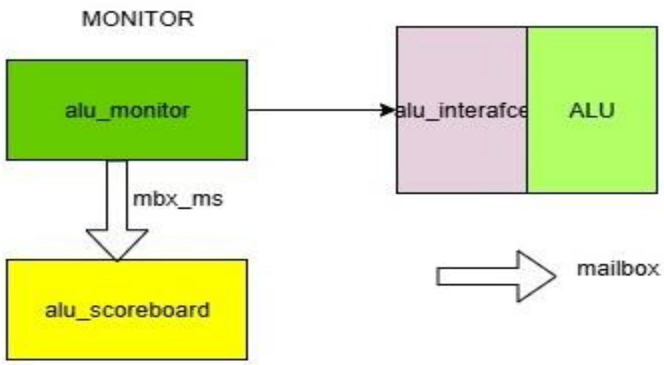



Figure 2.3.5: ALU monitor component

### 2.3.6 REFERENCE MODEL COMPONENT

alu_reference_model:

•A golden model of the ALU that predicts the expected output.

•Receives the same transaction as the DUT (from generator).

•Sends the predicted output to the alu_scoreboard

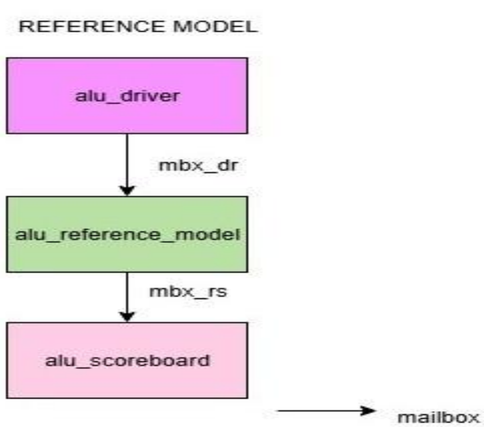



figure 2.3.1: ALU reference model component

### 2.3.7 SCOREBOARD COMPONENT

alu_scoreboard:

- Compares outputs from the DUT (via alu_monitor) and the reference model.
- Logs mismatches or confirms correctness.
- Verifies functional accuracy.

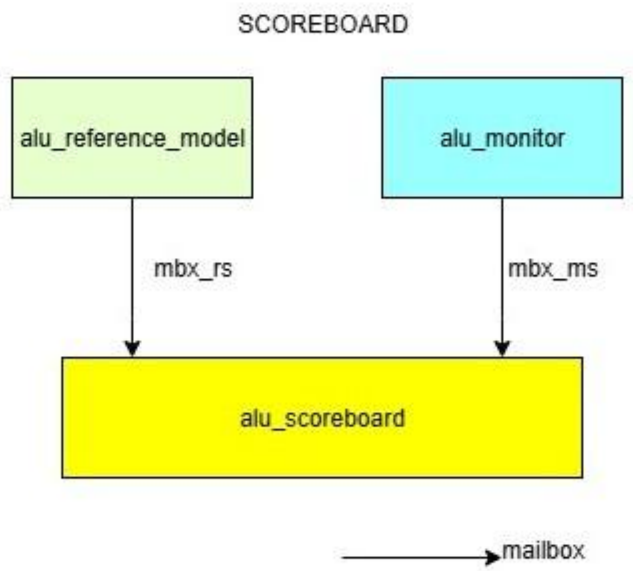



Figure 2.3.7: ALU scoreboard component

### 2.3.8 ENVIRONMENT COMPONENT

The ALU environment instantiates and connects all testbench components (generator, driver, monitor, reference model, scoreboard) into a unified verification system. It manages inter-component communication through mailboxes and distributes virtual interface handles for DUT access. The environment serves as the top-level orchestrator that controls test execution and coordinates the complete verification flow.

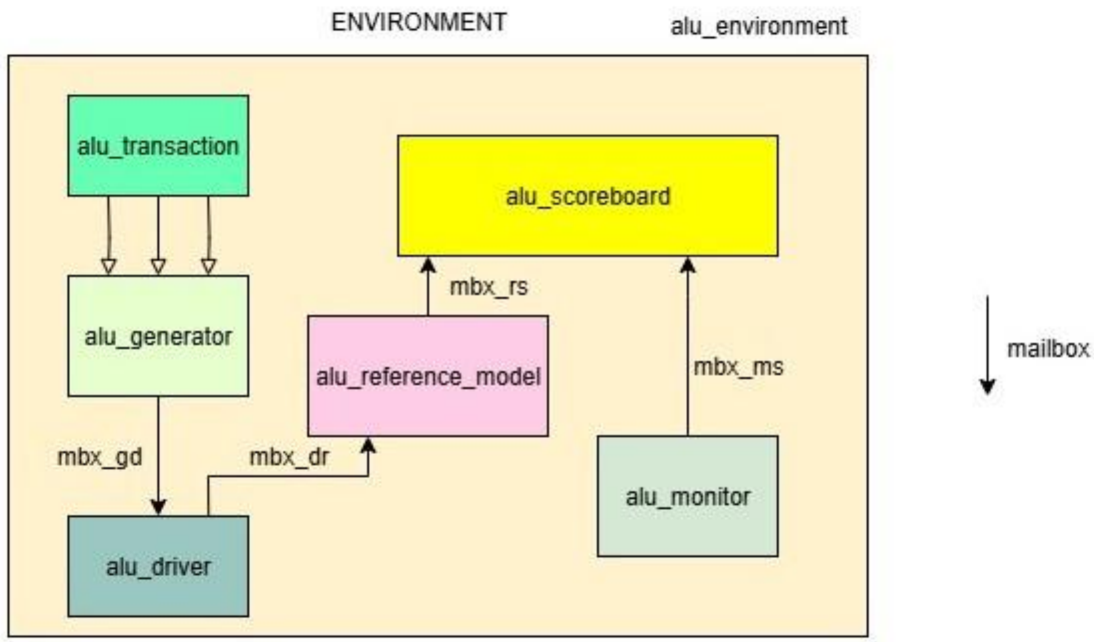



Figure 2.3.1: ALU environment component

### 2.3.9 TEST COMPONENT

The ALU test serves as the configuration and control layer that instantiates the test environment and defines specific test scenarios. It configures test parameters, initializes the ALU environment with appropriate settings, and controls the stimulus generation through predefined test sequences. The test class acts as the entry point for executing targeted verification scenarios and managing the overall test flow.

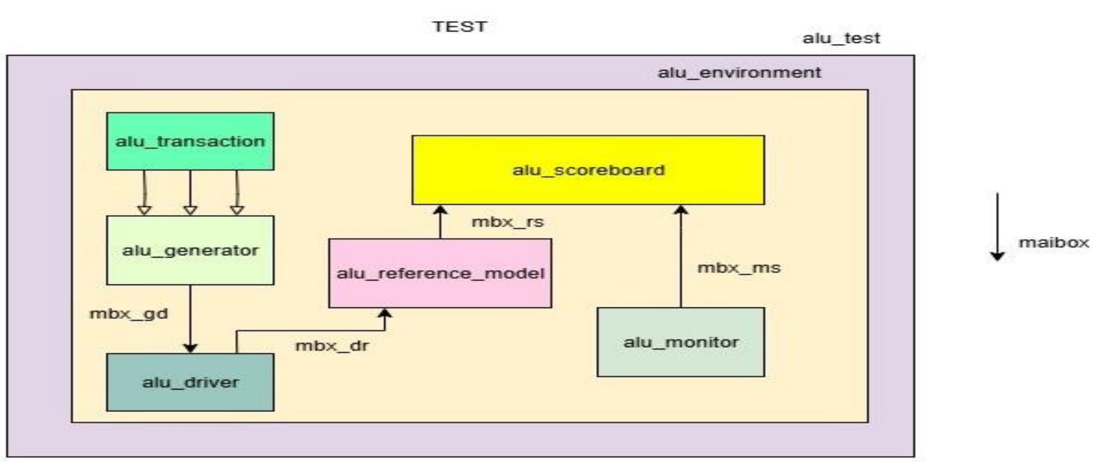



Figure 2.3.9: ALU test component

### 2.3.10 TOP COMPONENT

alu_top (Outer Box):

•Top-level testbench module that instantiates everything needed to verify the ALU.

•Includes the alu_test and the alu_interface, which connects the design to the testbench.

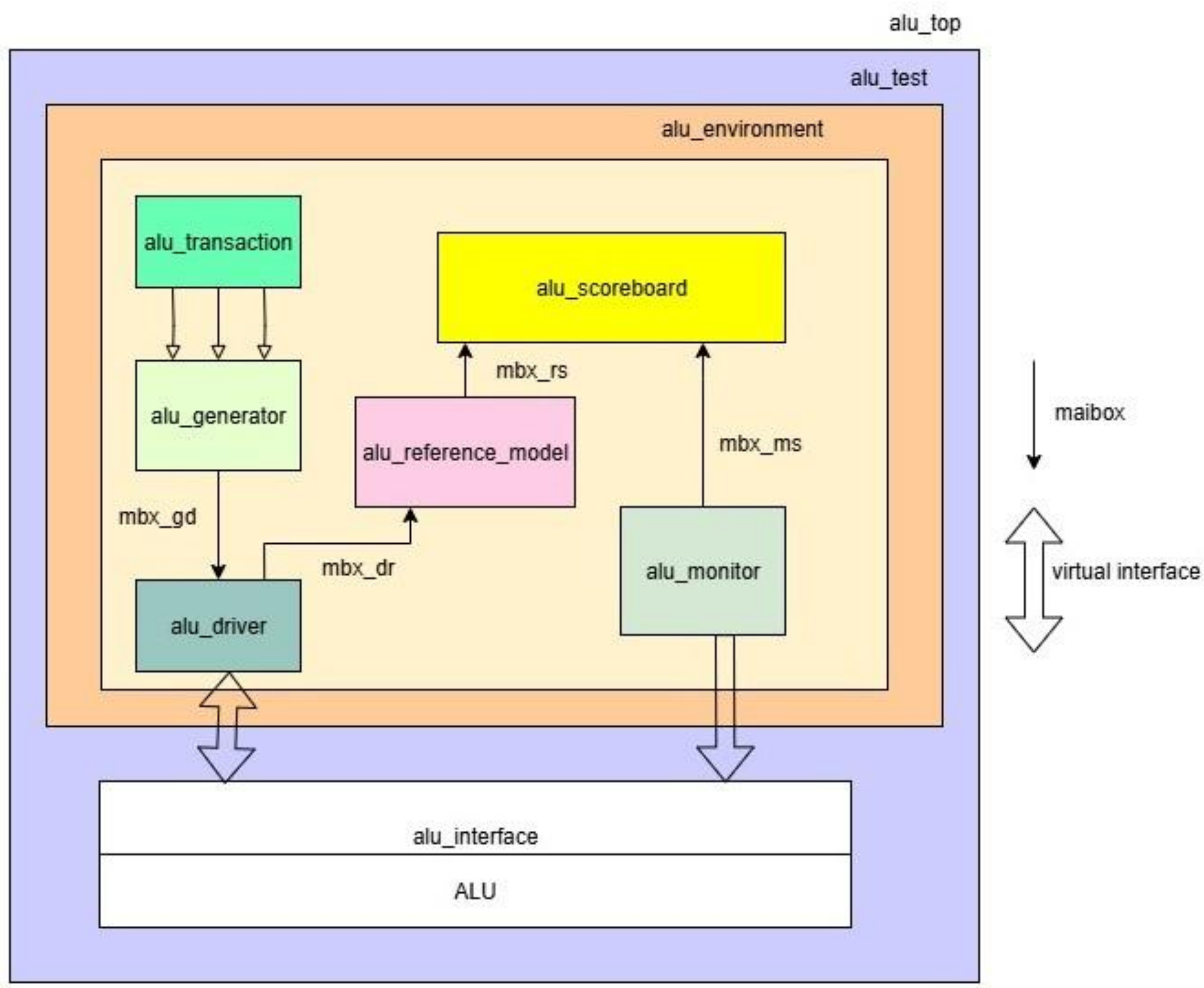



Figure 2.3.10: ALU top component

**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

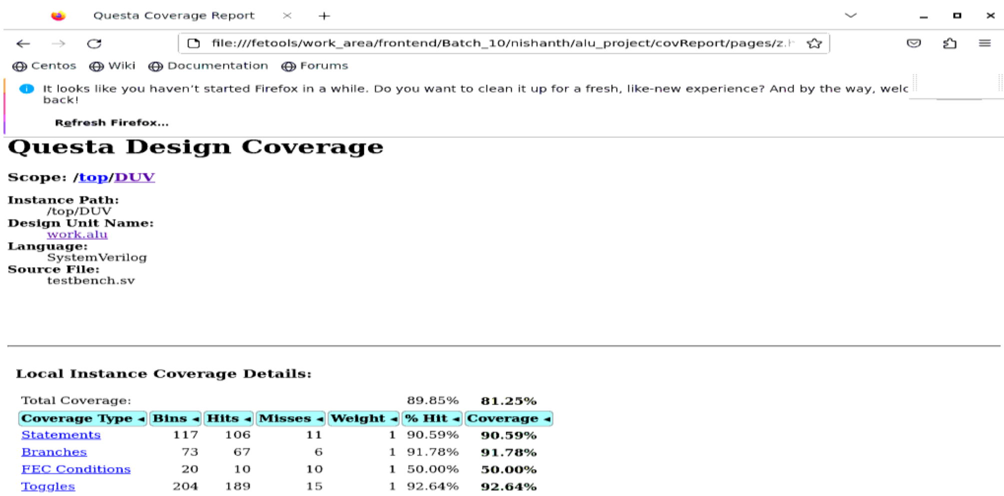
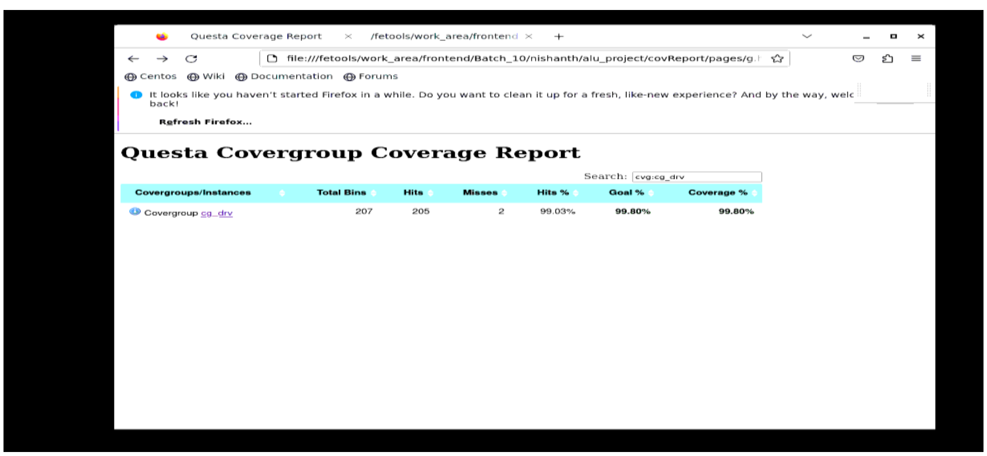
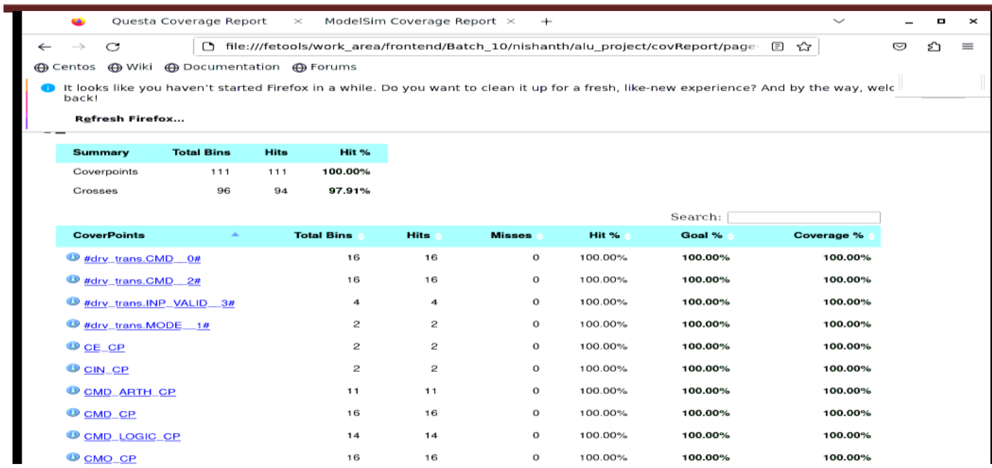



Figure 3.2: code coverage

## 3.3 INPUT FUNCTIONAL COVERAGE



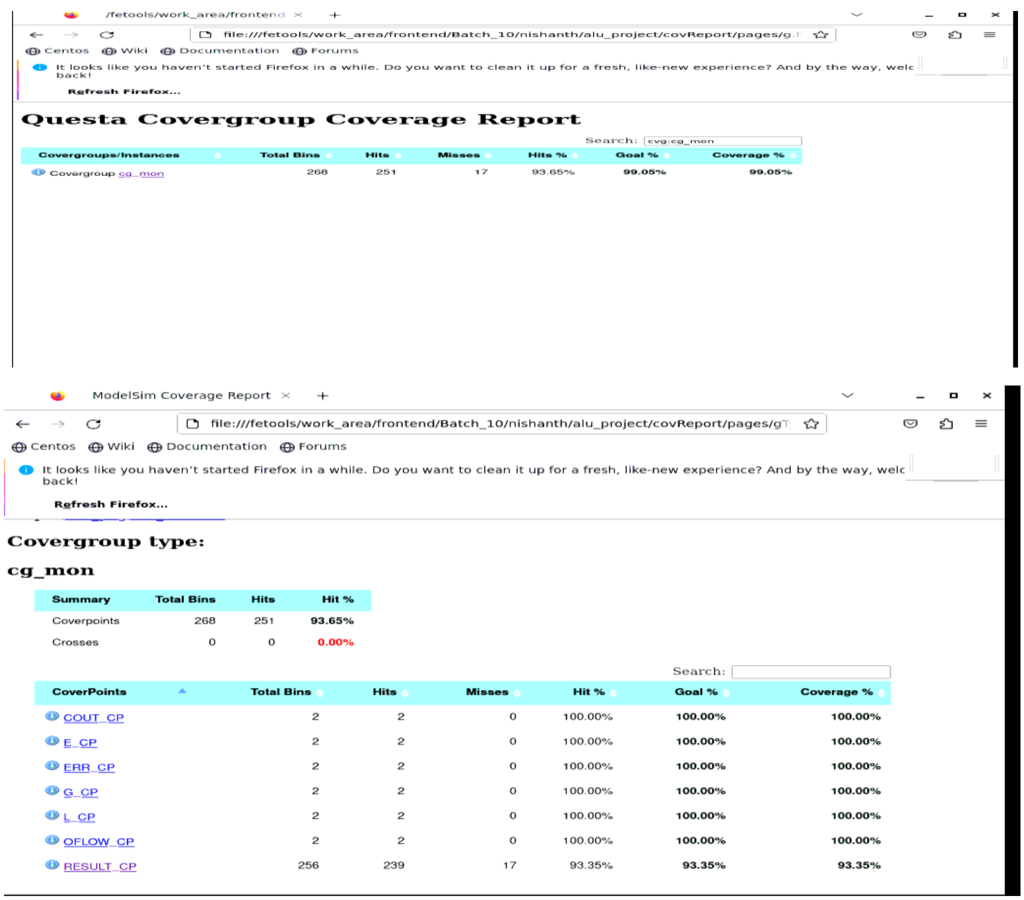


| Summary | Total Bins | Hits | Hit % |
|---|---|---|---|
| Coverpoints | 111 | 111 | 100.00% |
| Crosses | 96 | 94 | 97.91% |

| CoverPoints | Total Bins | Hits | Misses | Hit % | Goal % | Coverage % |
|---|---|---|---|---|---|---|
| #drv_trans.CMD__0# | 16 | 16 | 0 | 100.00% | 100.00% | 100.00% |
| #drv_trans.CMD__2# | 16 | 16 | 0 | 100.00% | 100.00% | 100.00% |
| #drv_trans.INP_VALID__3# | 4 | 4 | 0 | 100.00% | 100.00% | 100.00% |
| #drv_trans.MODE__1# | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| CE_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| CIN_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| CMD_ARTH_CP | 11 | 11 | 0 | 100.00% | 100.00% | 100.00% |
| CMD_CP | 16 | 16 | 0 | 100.00% | 100.00% | 100.00% |
| CMD_LOGIC_CP | 14 | 14 | 0 | 100.00% | 100.00% | 100.00% |
| CMO_CP | 16 | 16 | 0 | 100.00% | 100.00% | 100.00% |

Figure 3.3: input functional coverage

## 3.4 OUTPUT FUNCTIONAL COVERAGE



Questa Covergroup Coverage Report

| Covergroups/Instances | Total Bins | Hits | Misses | Hits % | Goal % | Coverage % |
|---|---|---|---|---|---|---|
| Covergroup cg_mon | 268 | 251 | 17 | 93.65% | 99.05% | 99.05% |

Covergroup type:

cg_mon

| Summary | Total Bins | Hits | Hit % |
|---|---|---|---|
| Coverpoints | 268 | 251 | 93.65% |
| Crosses | 0 | 0 | 0.00% |

| CoverPoints | Total Bins | Hits | Misses | Hit % | Goal % | Coverage % |
|---|---|---|---|---|---|---|
| COUT_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| E_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| ERR_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| G_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| L_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| OFLOW_CP | 2 | 2 | 0 | 100.00% | 100.00% | 100.00% |
| RESULT_CP | 256 | 239 | 17 | 93.35% | 93.35% | 93.35% |

Figure 3.4: output functional coverage

## 3.5 ASSERTION COVERAGE

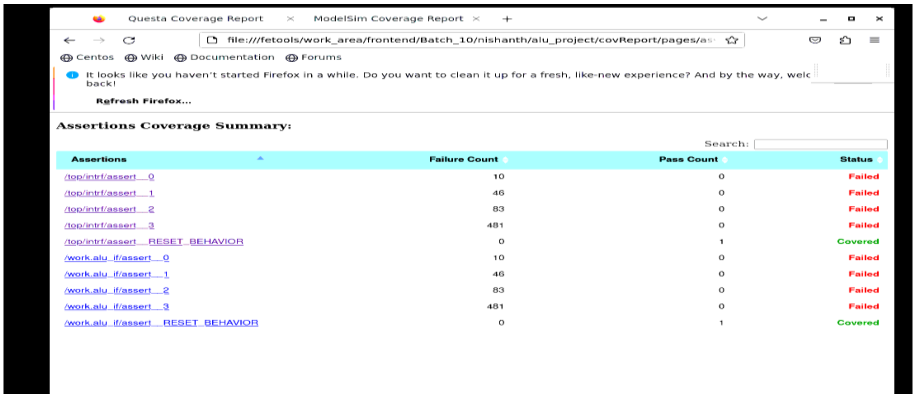



Figure 3.5: assertion coverage

## 3.6 OVERALL COVERAGE

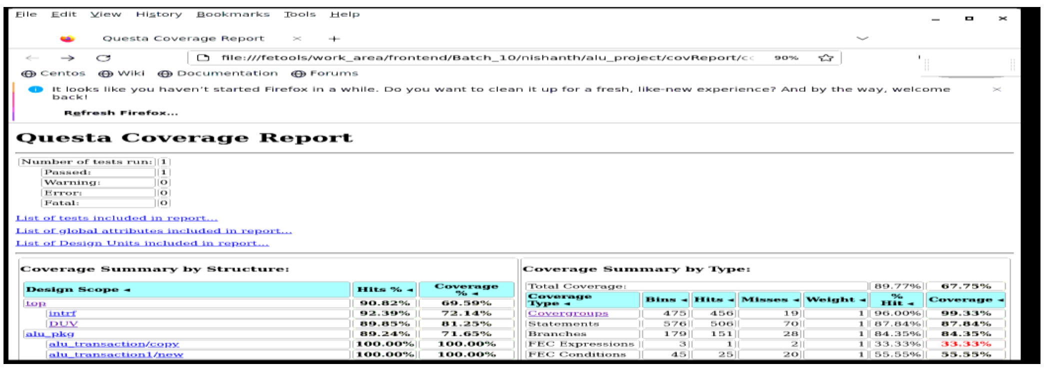



Figure 3.6: overall coverage

## 3.7 OUTPUT WAVEFORM

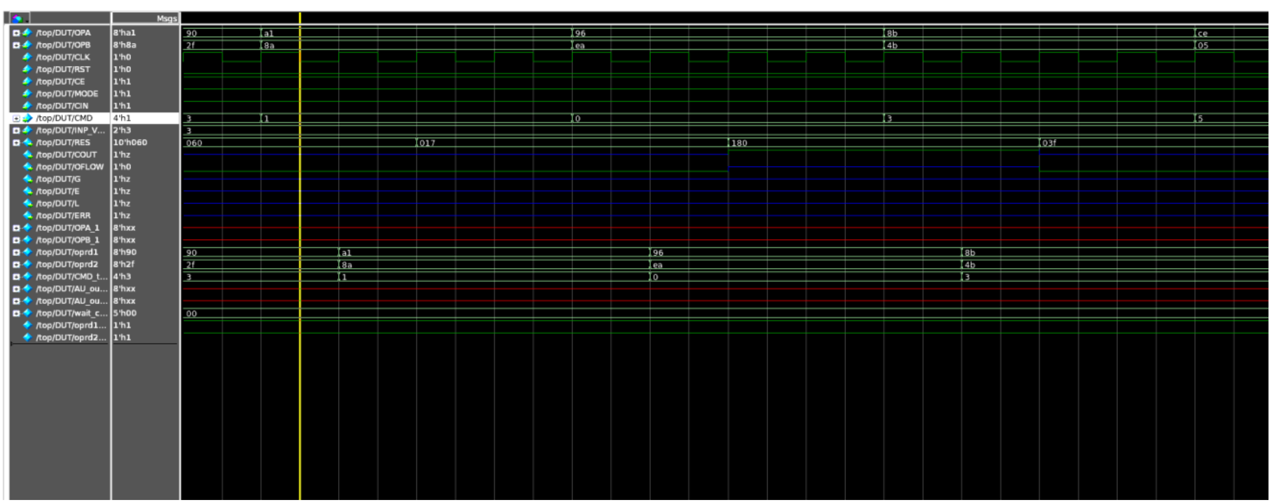


Figure 3.7: output waveform