ALU Verification Document

By Ashok D’sa EMP ID:6070

Table of Figures

1. Project Overview

1.1 Arithmetic Logic Unit (ALU)

An Arithmetic Logic Unit (ALU) is a core component of a computer's central processing unit (CPU) responsible for carrying out arithmetic operations (such as addition, subtraction, multiplication and division) and logic operations (such as AND, OR, NOT and XOR). It is one of the most essential parts of the CPU, enabling it to process data and make decisions. The performance and efficiency of the ALU directly impact the overall speed and capability of a computer system.

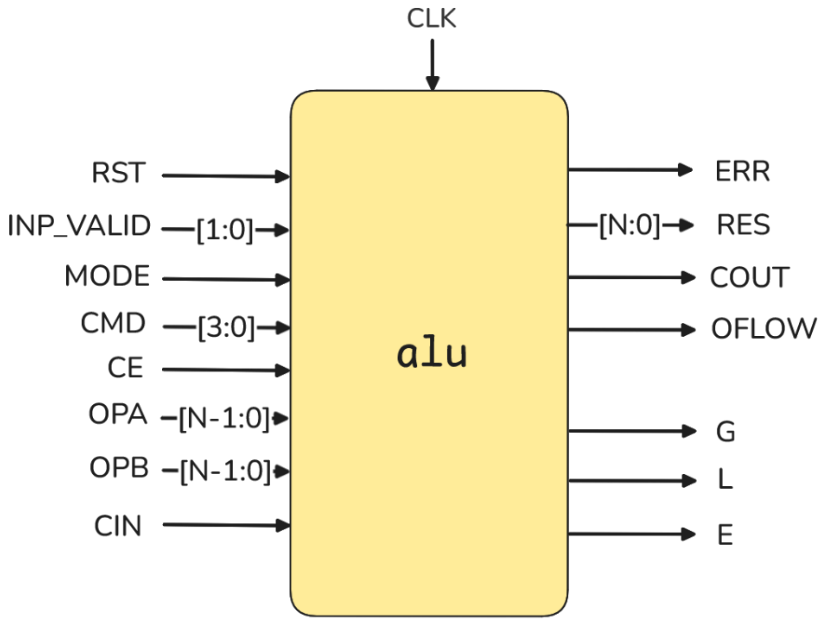

Figure 1: ALU Block

1.2 Advantage of ALU

- Integrates arithmetic and logic functions into a single unit, reducing the need for additional components.
- Facilitates advanced operations needed in scientific, engineering and real-time applications.
- Acts as the core computational unit, enhancing the speed and capability of the processor.

1.3 Disadvantage of ALU

- ALUs typically perform only basic arithmetic and logic operations, more complex tasks require additional hardware or processing units.
- ALU’s rely on registers and memory units for input and output handling.
- An ALU needs instructions from the control unit to operate, making it incapable of functioning independently.
- ALUs are general purpose, they may not be optimized for specialized tasks.

1.4 Use cases of ALU

- Used in CPUs to execute basic arithmetic operations like addition, subtraction, multiplication and division.
- Decision Making in Programs using logical operations (e.g., AND, OR, NOT) essential for conditional branching and comparisons in software.
- Found in microcontrollers used in appliances, automobiles and industrial machines for real-time control and processing.
- Supports calculations needed for rendering graphics, physics simulations and game logic.
- Used in digital signal processors (DSPs) for fast, repetitive arithmetic operations on audio, video, or image data.

2. Verification Objectives

- Review design documentation: analyze functional requirements, supported operations, inputs/outputs, operating modes and edge-case behaviors.
- Construct the verification plan.
- Develop functional coverage and assertion plan.
- Frame the testbench architecture for the alu design.
- Creation of template codes for testbench components
- Implement and enhance testbench for coverage, integrating the functional coverage and SystemVerilog assertions.
- Validating the functional correctness of the ALU, considering the corner cases as well.
- Validating the timing of the operations.
- Checking its robustness against errors.

3. DUT Interfaces

These are the signals present in the interface which is to be shared between the Test and Design.

Signal	Direction	Size(bits)	Description
CLK	INPUT	1	Clock Signal
RST	INPUT	1	Active High Asynchronous Reset
INP_VALID	INPUT	2	Shows the Validity of the Operands(active high) MSB shows the validity of the 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
CE	INPUT	1	Active high clock enable signal
OPA	INPUT	Parameterized	Operand 1
OPB	INPUT	Parameterized	Operand 2
CIN	INPUT	1	Carry In signal
ERR	OUTPUT	1	Active High Error Signal
RES	OUTPUT	Parameterized + 1	Result of the instruction performed by the ALU.
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 lesser than the value of OPB
E	OUTPUT	1	Comparator output which indicates that the value of OPA is equal than the value of OPB
OFLOW	OUTPUT	1	Indicates output overflow, during Addition/Subtraction

4. Testbench Architecture

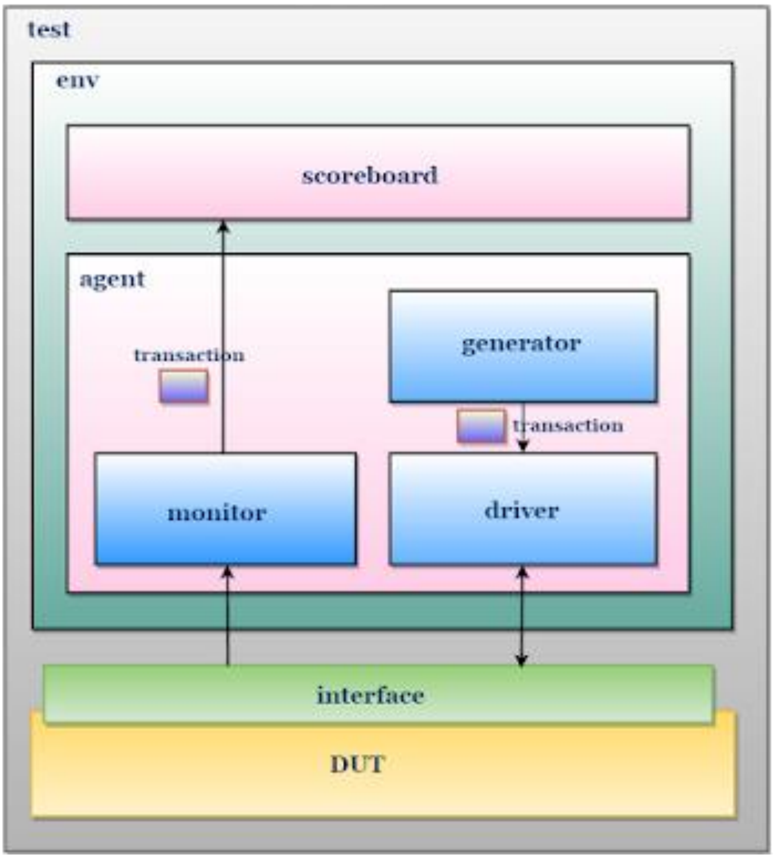

Figure 2: General SystemVerilog TestBench Architecture

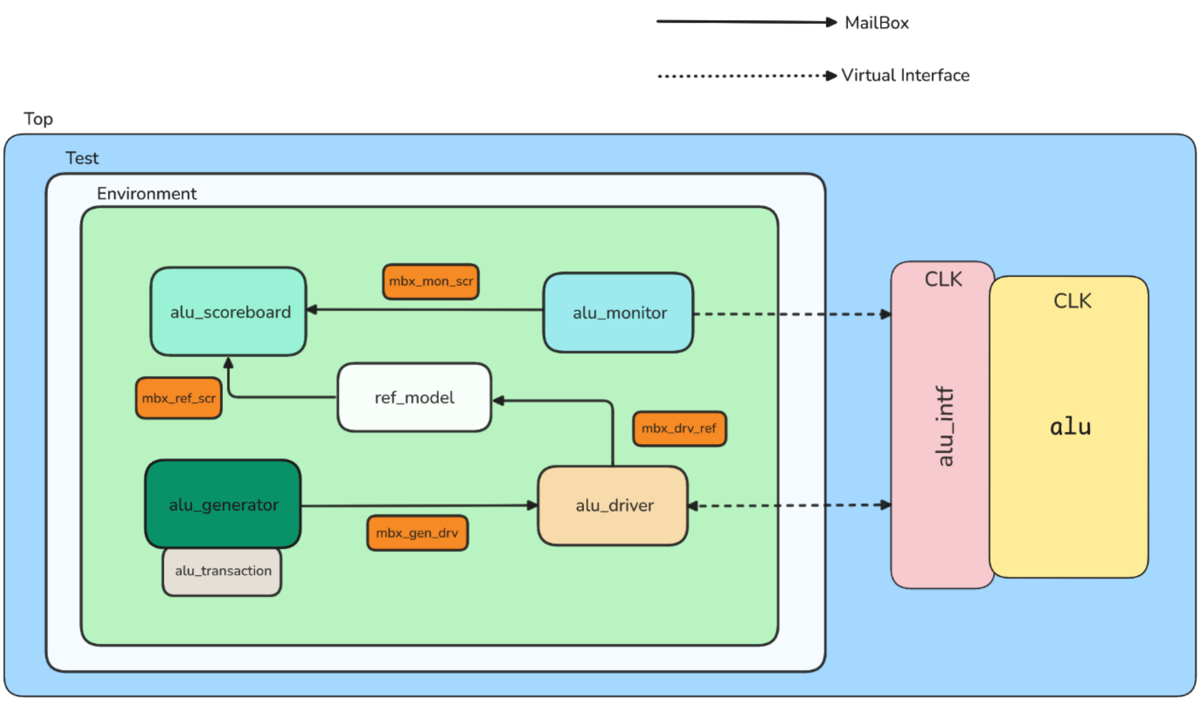

Figure 3: Test Bench Architecture for ALU

4.1 Flow Chart of each component with explanation

Interface

The Interface component encapsulates a bundled set of signals that connect the testbench to the DUT at the pin level. It directly maps to the DUT's input and output ports. This interface is accessed by testbench components such as the driver and monitor using virtual interface handles, enabling structured and reusable connectivity.

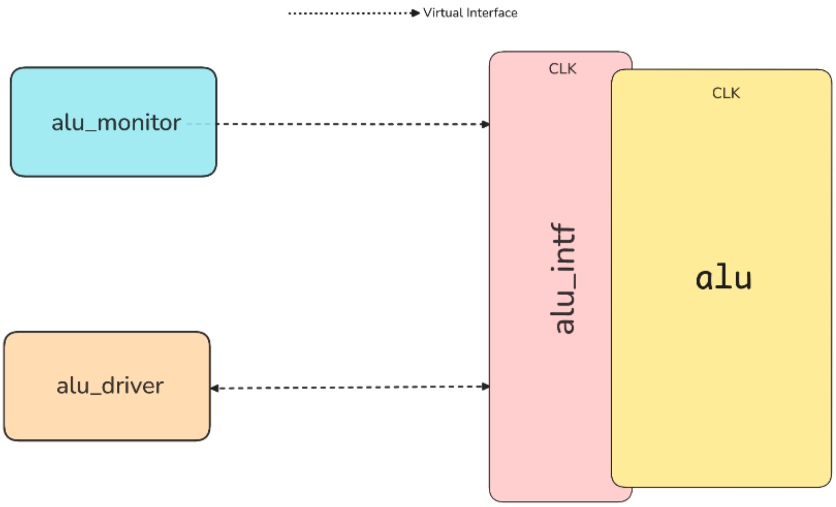

Figure 4: Interface Component

Transaction

Transaction component is an object that encapsulates the stimulus exchanged between testbench components, containing all randomized inputs and non-randomized outputs of the DUT, excluding the clock signal, which is generated separately in the top module. The transaction and can have constraints to target specific test scenarios.

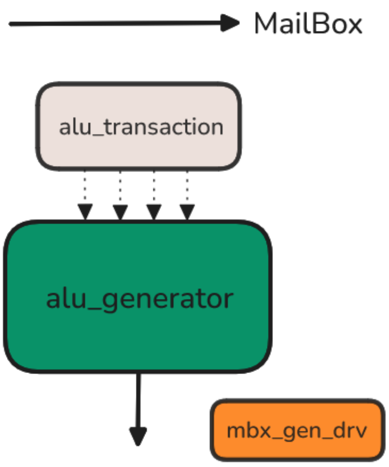

Figure 5: Transaction Component

Generator

Generator component of the testbench which generates constrained random stimuli (transactions) for the DUT. The generator then sends the generated stimuli to the driver through a mailbox(mbx_gen_drv).

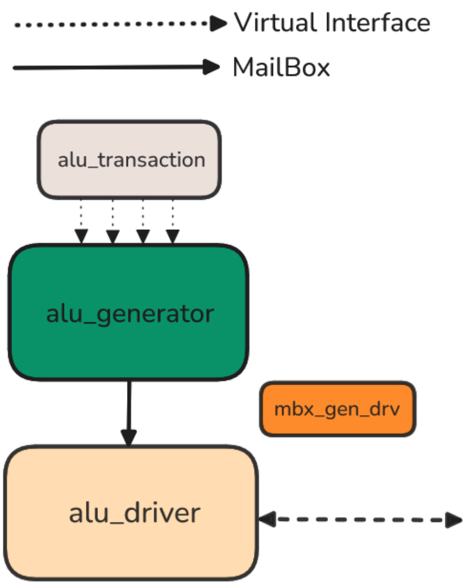

Figure 6: Generator Component

Driver

Driver component converts high-level transactions into pin-level activity at the DUT inputs. It receives transactions from the generator via a mailbox(mbx_gen_drv) and drives them to the DUT using a virtual interface. Also, it forwards the received transactions to the reference model through another mailbox (mbx_drv_ref) for result prediction and comparison.

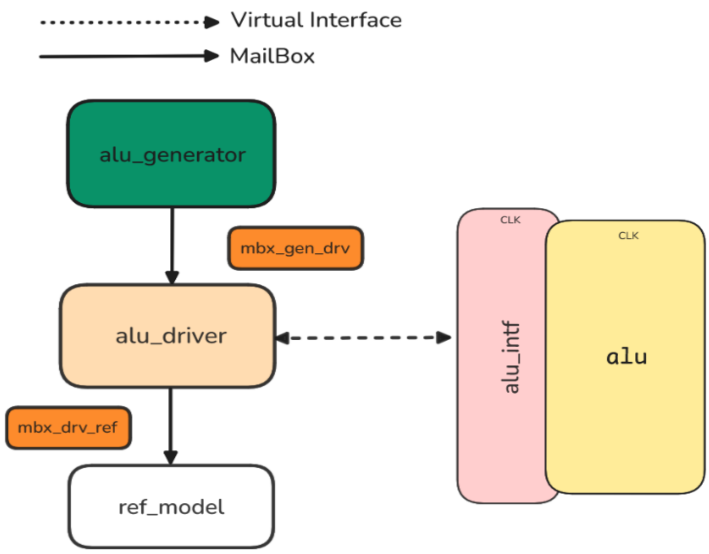

Figure 7: Driver Component

Reference Model

Reference Model serves as a golden implementation/expected output for output prediction, validation and evaluation of the actual output. It is typically non-synthesizable. It is used to validate functionality and evaluate system performance. The model receives input transactions from the driver via a mailbox(mbx_drv_ref), processes them according to the intended functionality, and sends the predicted outputs to the scoreboard through another mailbox(mbx_ref_scr) for comparison.

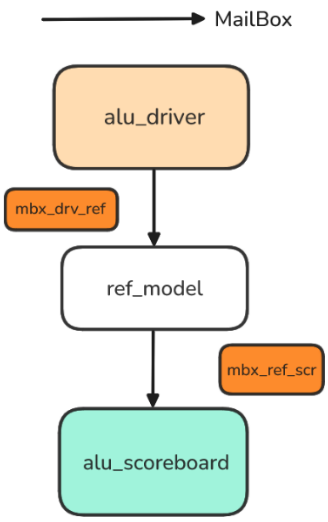

Figure 8: Reference Model

Monitor

Monitor component converts pin-level activity from the DUT outputs into high-level transactions. It captures the DUT's output signals via a virtual interface and packages them into transactions, which are then sent to the scoreboard through a mailbox(mbx_mon_scr) for comparison and analysis.

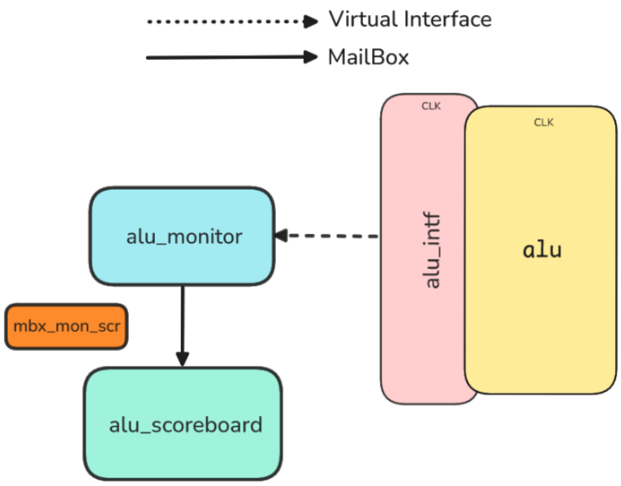

Figure 9: Monitor Component

Scoreboard

Scoreboard component receives the expected transactions from the reference model via one mailbox(mbx_ref_scr) and the actual transactions from the monitor via another(mbx_mon_scr). It compares these transactions to validate functional correctness and generates a report highlighting any mismatches or confirming successful operation.

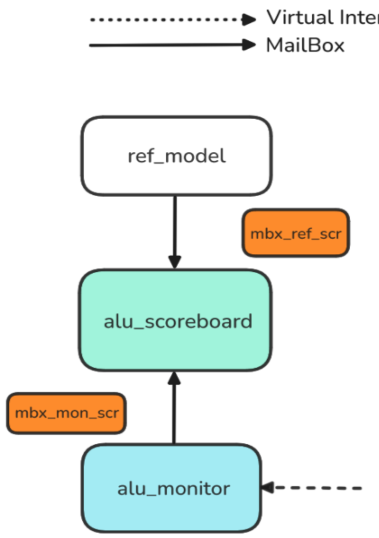

Figure 10: Scoreboard Component

Environment

Environment component serves as the central part of the testbench, responsible for instantiating and connecting all the major sub-components, including the generator, driver, monitor, reference model, and scoreboard. It builds and organizes the testbench, ensuring proper communication and data flow between components for seamless verification.

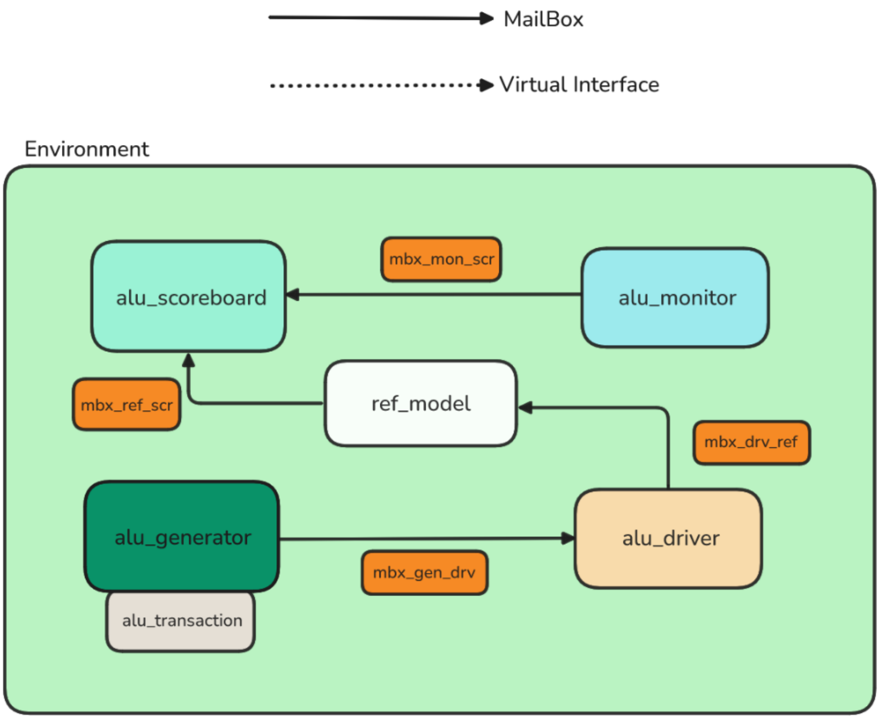

Figure 11: Environment Component

Test

Test component is responsible for defining and executing various test cases. It instantiates and builds the verification environment, configuring it as needed to apply specific test scenarios and stimuli to the DUT.

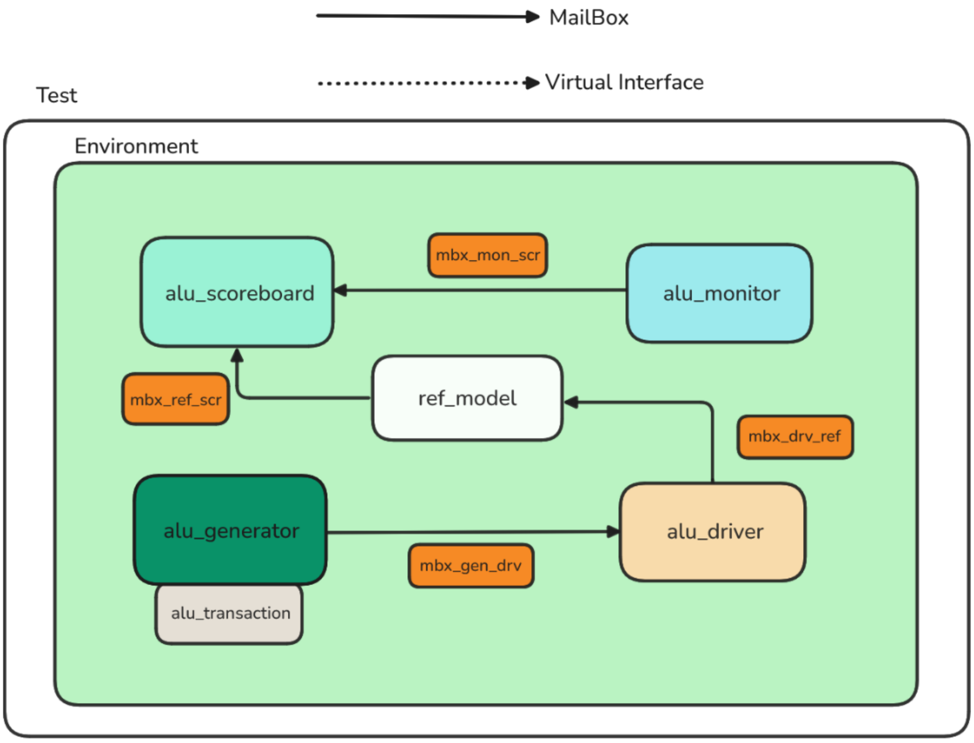

Figure 12: Test Component

Top

Top module is the component that instantiates all components (DUT, interface and test) and is responsible for the clock generation.

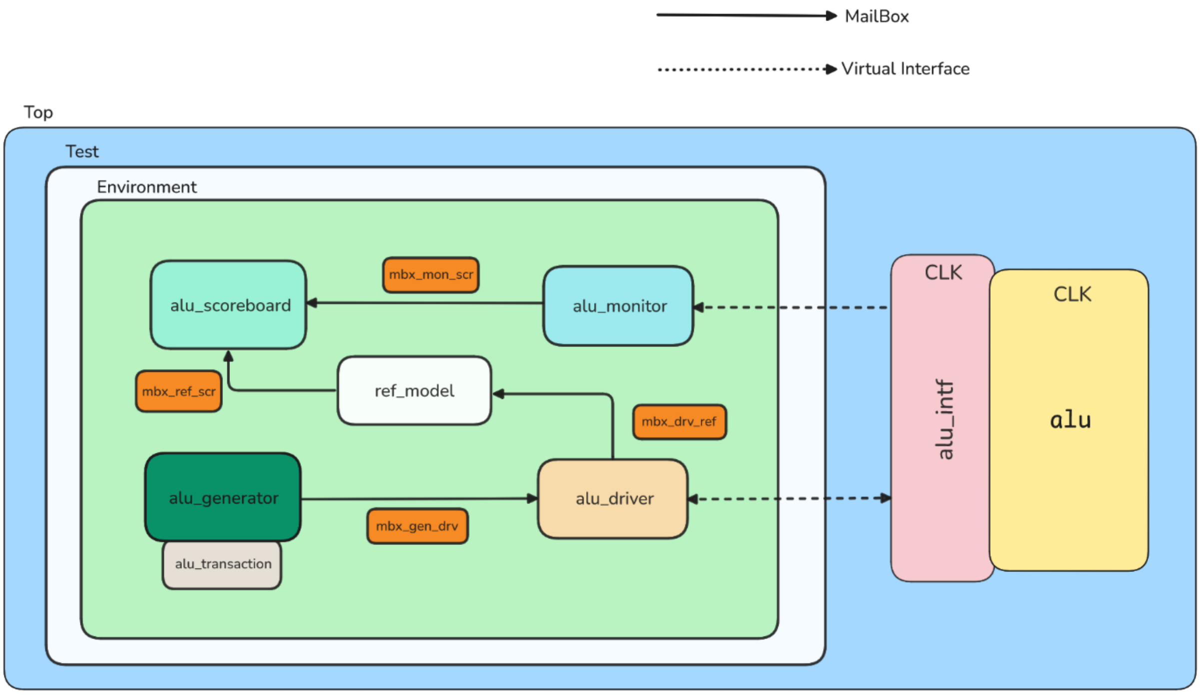

Figure 13: Top

3. Test Plan

[GitHub Link]

3.1 Test Scenarios

[Test Plan]

3.2 Functional Coverage Plan

[Functional Coverage Plan]

3.3 Assertions

[Assertion Plan]

4. Bugs Identified

4.1 Incorrect Output for Multiple Commands

Several ALU commands are failing to provide correct results. The affected commands are:

- OR
- CMP (Compare)
- DEC_B (Decrement B)
- INC_B (Increment B)
- INC_A (Increment A)
- ADD_CIN (Addition with Carry Input)
- ADD
- SH_MUL (Shift Multiply)

For each of these operations, the output generated does not match the expected results as specified in the ALU's functional requirements.

4.2 Error Detection Failure in Rotate Command

The Rotate command fails to trigger its designated error condition. Ideally, when a rotate operation enters an abnormal or unsupported state, the error flag or condition should activate preventing erroneous rotate operations to go undetected in simulation.

4.3 Input Valid Signal Not Handling Errors

When the inp_valid signal is set to 2'b00 (00 in binary), the ALU does not properly trigger its error indication mechanism. According to design requirements, this setting represents an invalid or undefined input state and should raise an error flag.

4.4. Output Width Truncation in Certain Commands

For these commands, there is an output width bug:

- INC_A
- SHL1_B (Shift Left by 1, B)
- SHL1_A (Shift Left by 1, A)

The result is updated and propagated only for the lower 8 bits rather than the required 9 bits.

This truncation leads to incomplete data and may affect upper bit calculations which rely on the full output width.

4.4 Output

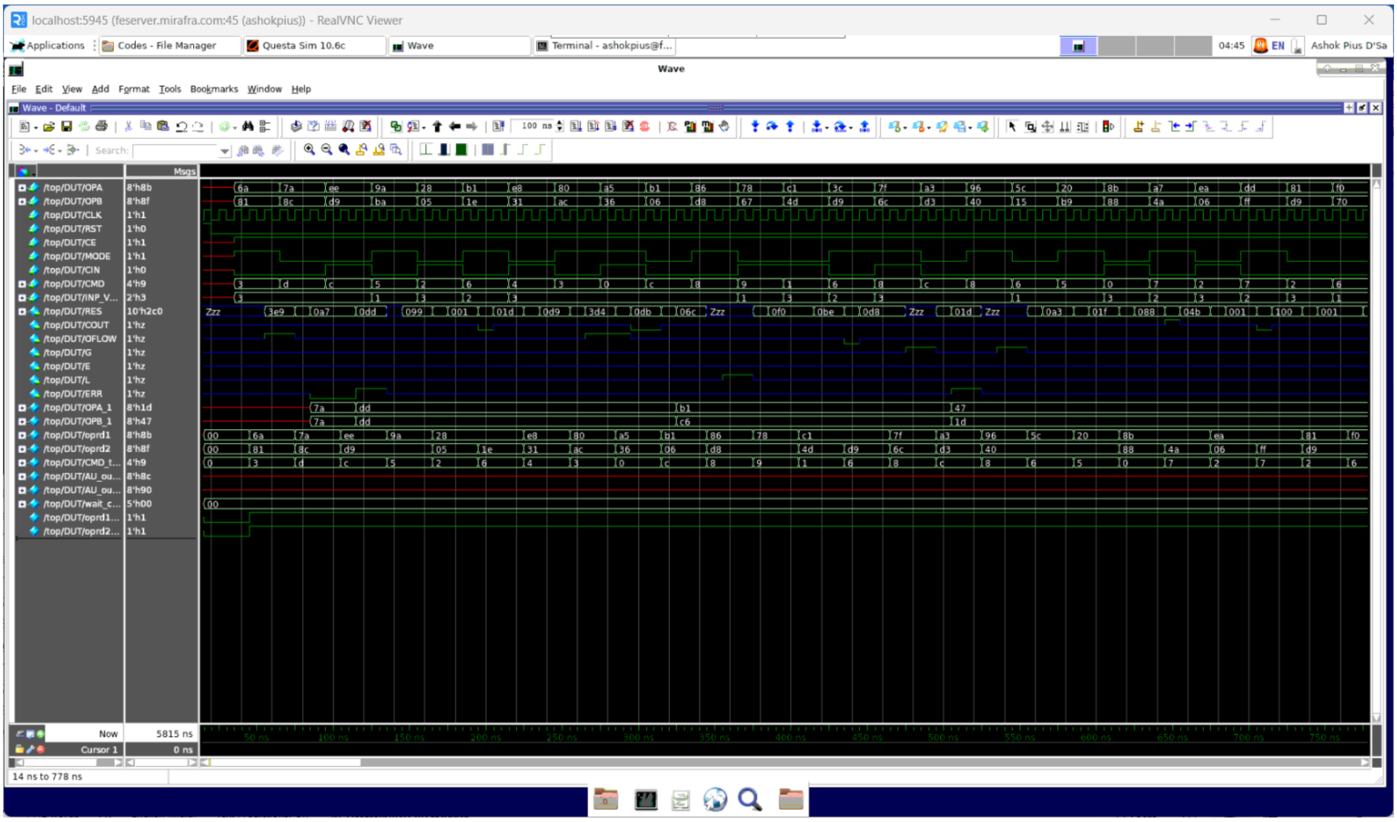

Figure 14: Running Regression Test