ALU VERIFICATION

PROJECT OVERVIEW:

This project details the comprehensive verification of a parameterized Arithmetic Logic Unit (ALU) using a modern, class-based SystemVerilog testbench. The primary objective is to ensure the functional and timing correctness of the ALU design through a structured and reusable verification environment.

VERIFICATION OBJECTIVES:

FUNCTIONAL CORRECTNESS:

Arithmetic Operations: Verify that all arithmetic commands (`ADD`, `SUB`, `ADD_CIN`, `SUB_CIN`, `INC_A`, `DEC_A`, `CMP`, multiplication, etc.) produce the correct values for `RES`, `COUT`, `OFLOW`, and the comparator flags (`G`, `L`, `E`).
Logical Operations: Verify that all logical commands (`AND`, `OR`, `XOR`, `NOT`, `NAND`, `NOR`, etc.) produce the correct bitwise results including shift and rotate operations.

TIMING CORRECTNESS:

Latency : Verify that standard operations complete in the specified 2 clock cycles, and that multi-cycle instructions, like multiplication, complete 3 clock cycles.

ERROR HANDLING:

Invalid Command Errors: Verify that the ERR signal is asserted correctly when an invalid operation is attempted, such as a rotate command where `OPB[7:4]` is non-zero.

Operand Timeout: Verify that the design correctly triggers the 16-cycle timeout and asserts the ERR signal if a second required operand is not valid within 16 clk cycles.

TEST,COVERAGE AND ASSERTION PLAN:

Test Plan Completion: To successfully execute all tests outlined in the Test Plan with zero failures, errors, or scoreboard mismatches.

Functional Coverage: Achieve 100% of the coverage goals defined in the Functional Coverage Plan. This includes hitting all defined coverpoints and crosses for every command, mode, and range of input operands.

Assertion Coverage: Evaluate all assertions in the Assertion Plan, ensuring all properties have been active and tested without failure during simulation.

DUT INTERFACES:

The ALU (Design Under Test) interface is composed of three primary categories of pins: data pins for the operands, control pins to manage the ALU's operation and timing, and output pins to present the result and status

flags. A clear understanding of these interfaces is fundamental to creating an effective verification environment.

- **Data Pins:** These include the two primary parameterized operands, `OPA` and `OPB`, along with a 1-bit carry-in signal, `CIN`, used for arithmetic operations.
- **Control Pins:** These signals manage the ALU's state and behavior. They include the standard `CLK` and active-high asynchronous `RST`, a clock enable `CE`, an `INP_VALID` bus to signal when operands are ready, a `MODE` pin to switch between arithmetic and logical operations, and a `CMD` bus to select the specific instruction.
- **Output Pins:** The ALU's primary output is the `RES` bus, which is one bit wider than the operands to include a carry-out. Additional outputs provide critical status information, including
  `COUT` (carry-out) ,
  `OFLOW` (overflow) , comparator flags (`G`, `L`, `E`) , and an `ERR` signal for illegal operations or timeouts.

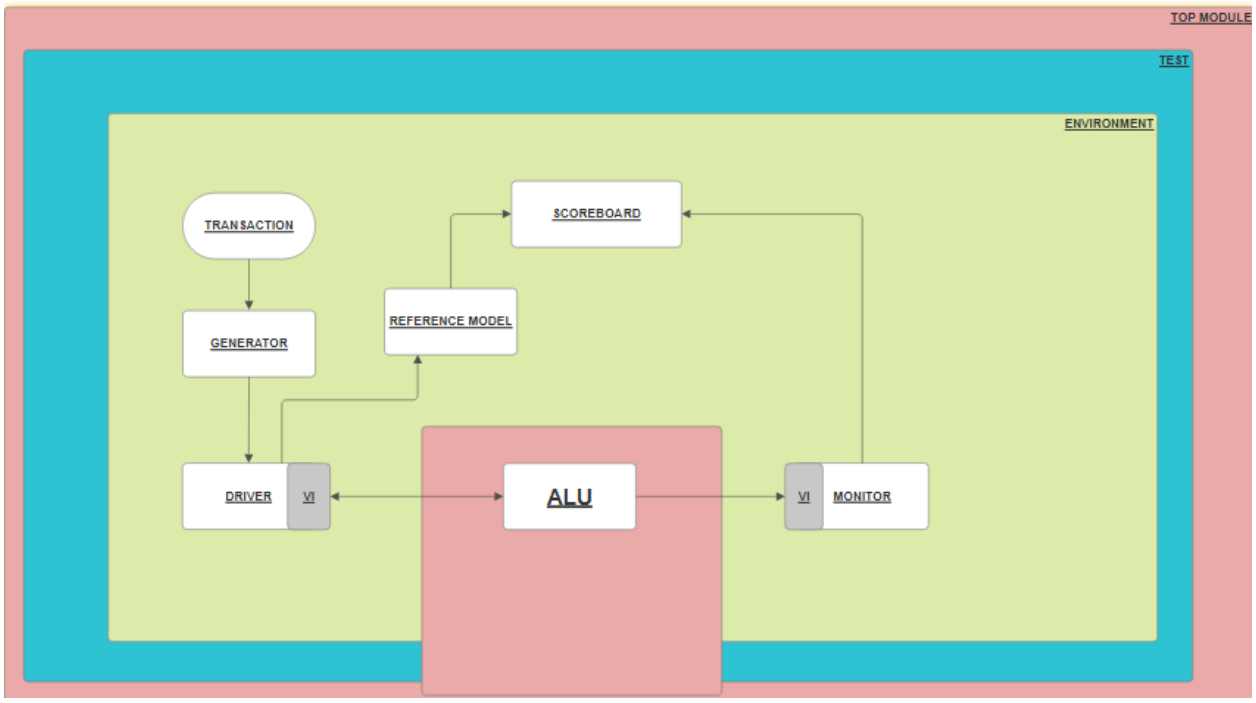
Pin Name	Direction	Width (bits)	Function
OPA	INPUT	Parameterized	The first primary operand for the operation.
OPB	INPUT	Parameterized	The second primary operand for the operation.

`CIN`	**INPUT**	**1**	**Carry-in signal used in certain arithmetic operations.**
`CLK`	**INPUT**	**1**	**The main clock signal for the synchronous design.**
`RST`	**INPUT**	**1**	**Active-high asynchronous reset..**
`CE`	**INPUT**	**1**	**Active-high clock enable signal. Operations are paused when this is low.**
`MODE`	**INPUT**	**1**	**Selects the operation mode: `1` for Arithmetic, `0` for Logical.**
`INP_VALID`	**INPUT**	**2**	**Indicates which operands are currently valid on the data bus.**
`CMD`	**INPUT**	**Parameterized (4)**	**The command code that selects the specific**

			instruction to be executed.
RES	**OUTPUT**	**Parameterized + 1**	**The result of the performed ALU instruction.**
OFLOW	**OUTPUT**	**1**	**Indicates an overflow has occurred during a subtraction operation.**
COUT	**OUTPUT**	**1**	**The carry-out signal generated by addition operations.**
G	**OUTPUT**	**1**	**Comparator output; asserted when OPA is greater than OPB.**
L	**OUTPUT**	**1**	**Comparator output; asserted when OPA is lesser than OPB.**
E	**OUTPUT**	**1**	**Comparator output; asserted when OPA is equal to OPB.**

ERR	OUTPUT	1	Asserts when an illegal operation is commanded or a timeout occurs.

TESTBENCH ARCHITECTURE:



The figure illustrates the Verification Architecture for an Arithmetic Logic Unit (ALU) using a modular testbench environment.

Key Components:

• **Transaction:** Defines the input and output of transactions (e.g., operands and operations) exchanged between components.

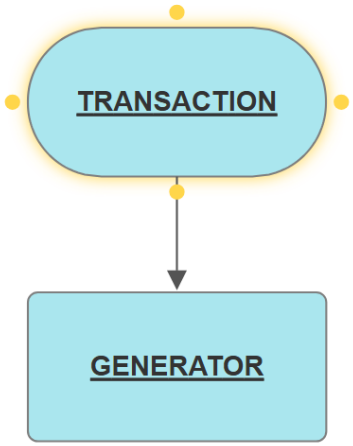
• **Generator:** Randomly creates test scenarios by generating transactions. These are sent to the driver for processing.

• **Driver:** Receives transactions from the generator and drives them to the DUV and reference model using a virtual interface (VI).
• **DUV (Design Under Verification):** The actual ALU design that receives inputs from the driver and generates outputs for validation.
• **Monitor:** Observes and captures the output signals from the DUV through the virtual interface. It converts them back into transaction format and forwards them to the scoreboard.
• **Reference_model:** Serves as a golden model that receives the same input as the DUV and produces the expected output for comparison.
• **Scoreboard:** Compares the output from the DUV (via the monitor) with the expected output from the reference model. Any mismatches are flagged as functional errors.
• **Mailboxes:** Facilitate communication between generator, driver, monitor, reference model, and scoreboard by passing transactions.

FLOWCHART :

1) TRANSACTION CLASS

The alu_transaction class encapsulates all ALU input stimuli and output responses for verification purposes.



COMPONENTS :

Randomized Stimulus (Inputs)

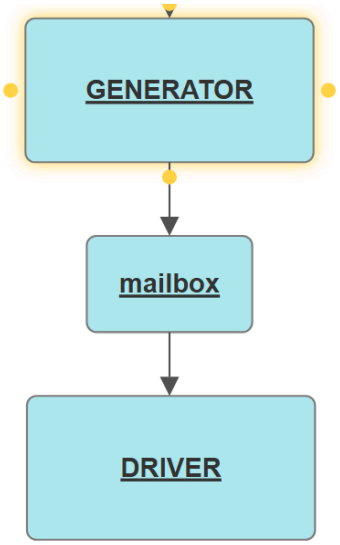
To effectively test the ALU, input signals like **`INP_VALID`**, **`MODE`**, **`CMD`**, **`OPA`**, **`OPB`**, and **`CIN`** are declared with the **`rand`** keyword. This allows the generator component to automatically create a wide range of random, valid test scenarios to drive the design.

Monitored Results (Outputs)

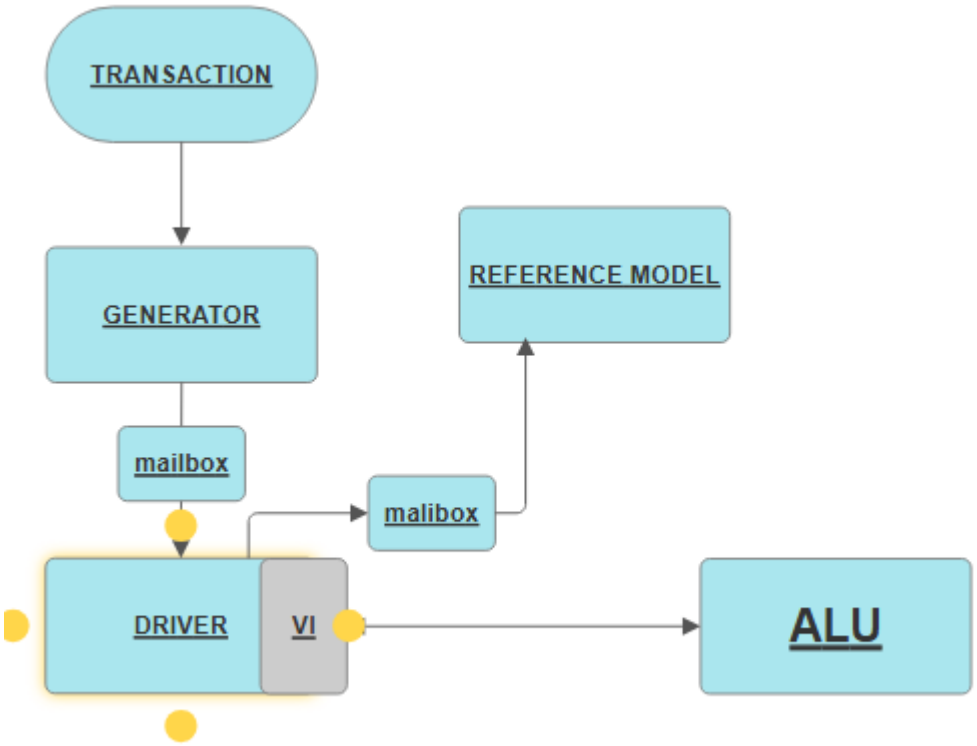
Conversely, the output signals—**`ERR`**, **`RES`**, **`OFLOW`**, **`COUT`**, **`G`**, **`L`**, and **`E`**—are declared as standard variables without the `rand` keyword. These variables don't generate stimulus; instead, they serve as containers to capture the ALU's response.

2) GENERATOR:

The generator contains the ALU transaction class handle and the mailbox handle, which is connected to the driver. The start() task randomizes and sends these transactions to the driver via the mailbox.



3) DRIVER:



Receives Transactions: The Driver's process starts by receiving an abstract transaction packet from the GENERATOR through a `mailbox`. This packet contains the data for one ALU operation.

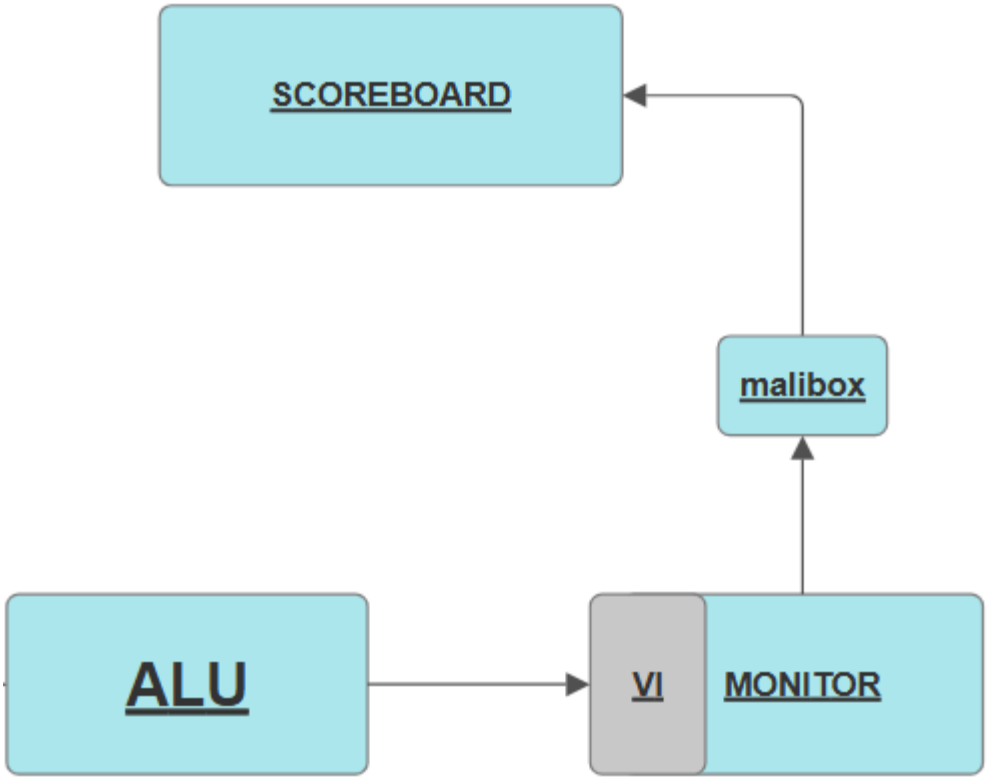
Drives the ALU: Its main responsibility is to translate this transaction into real, timed signals to drive the physical `ALU`. As the diagram shows, it uses a **Virtual Interface (VI)** to drive inputs to the ALU.
Forwards to Reference Model: At the same time, the Driver sends a copy of the transaction to the `REFERENCE MODEL` through a second `mailbox`. This ensures the Reference Model receives the exact same stimulus as the DUT, allowing it to calculate the expected result for later comparison.

4) MONITOR:

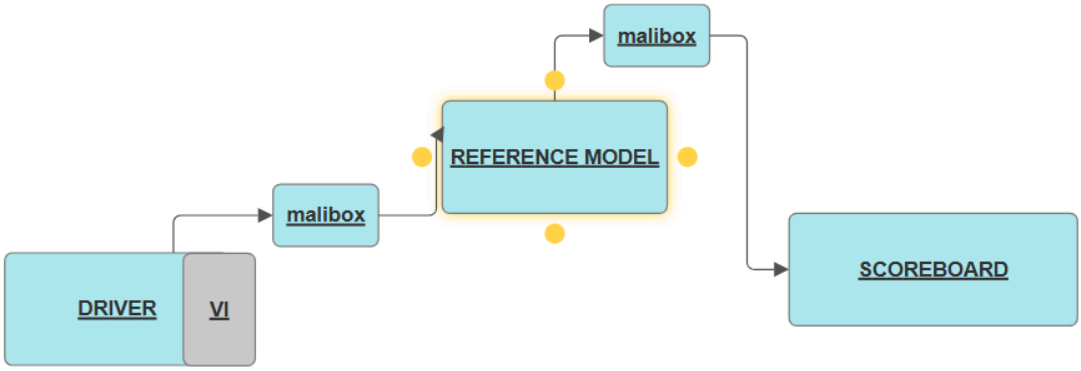
Observes the ALU: It constantly watches the output pins of the **ALU** (like `RES`, `COUT`, `ERR`, etc.). This connection is made through the **Virtual Interface (VI)**, which gives the Monitor access to the physical signals.

Converts Signals to Transactions: When it detects activity, the Monitor collects the values from the output pins and assembles them into a single transaction object, which represents the "actual result" from the DUT.

Sends to Scoreboard: Finally, it sends this completed transaction to the **Scoreboard** via a `mailbox`. The Scoreboard will then compare this actual result with the expected result from the Reference Model to check for correctness.



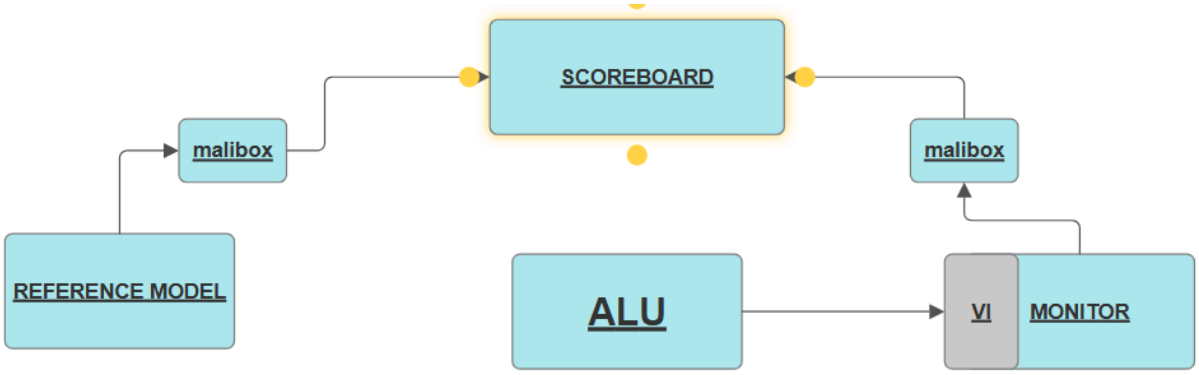
5) REFERENCE MODEL



Get Stimulus: It receives the exact same transaction from the Driver (via a `mailbox`) that the actual ALU is receiving.
Calculate Result: It performs the specified operation (e.g., addition, rotation) in a functionally-correct way, acting as an ideal version of the ALU.

Send to Scoreboard: It sends its calculated "expected result" to the Scoreboard (via another `mailbox`), which will use it as the correct answer for comparison against the actual ALU's output.

6) SCOREBOARD



The **Scoreboard** acts as the automated judge of the testbench, determining if the ALU is behaving correctly.

As the diagram clearly shows, its function is to compare the two results it receives:

1. **Expected Result:** It gets the "correct answer" in a transaction from the **Reference Model** via a `mailbox`.
2. **Actual Result:** It gets the "actual result" in a transaction from the **Monitor** (which observed the ALU) via another `mailbox`.

The Scoreboard's only job is to compare these two transactions. If the actual result from the ALU matches the expected result from the Reference Model, the test passes. If they differ in any way, the Scoreboard flags a mismatch, indicating a bug in the design.