**VERIFICATION DOCUMENT**

**Project Overview of ALU**

This project involves the verification of a parameterized Arithmetic Logic Unit (ALU) that supports both arithmetic and logical operations, with configurable data width (e.g., 16, 32, 64, or 128 bits). Unlike basic ALUs with fixed one-cycle latency, this design introduces operation-dependent timing behavior. As a verification engineer, the objective is to develop a robust testbench that ensures the design under test (DUT) behaves according to specification under all operating conditions and corner cases.

The ALU operates in two modes-arithmetic (mode=1) and logical (mode=0). In arithmetic mode, it supports operations such as ADD, SUB, ADD\_CIN, SUB\_CIN, CMP, INC, DEC, INC\_MUL, and SHIFT\_MUL. In logical mode, it performs operations such as AND, OR, NAND, NOR, XOR, XNOR, NOT, SHL, SHR, ROL, and ROR. The control signals include a 2-bit inp\_valid that indicates whether operands are valid, and the operation only executes when both operands are valid (inp\_valid == 2'b11). A key functional constraint is that each command must be paired with appropriate operand validity; invalid combinations lead to error handling.

One of the distinguishing features of this ALU is its variable latency behavior:

* Most operations generate output after one clock cycle from valid input.
* The ALU waits for 16 clock cycles for the missing operand (not both), and if it doesn’t arrive, the error signal (ERR) is raised in the next clock cycle.

The verification environment will be developed in SystemVerilog using a layered architecture. It includes a driver to apply input stimuli, a monitor to observe outputs, a reference model (scoreboard) to compute expected results, and assertions to ensure and functional correctness. A dedicated functional coverage model will track verification progress across all supported commands, modes, and flag combinations.

By the end of the verification process, the ALU will be tested for correctness, protocol compliance, flag generation, timing behavior, and edge cases. A full suite of reusable testbenches, test scenarios, assertions, coverage reports, and documentation will be delivered, ensuring the ALU design meets all required specifications and is ready for integration or silicon implementation.

**Verification Objectives**

The objective of this verification project is to ensure the functional correctness, timing accuracy and robustness of the ALU design under all supported configurations, operations, and corner cases. Given the design's advanced features-including multi-cycle operations, input-validity protocols, and error signaling-it is essential that the verification environment thoroughly checks both expected and exceptional behaviors.

1. Correctness of Functional Operations

* Verify the accurate execution of all supported arithmetic and logical operations (as defined by the CMD signal).
* Ensure correctness under both mode = 0 (logical) and mode = 1 (arithmetic).
* Ensure correct operation sequencing and flag generation across all valid command types.

1. Timing behaviour validation

* Ensure that standard operations produce results after exactly one clock cycle once both operands are valid.
* For two-operand operations, if only one operand is valid (2'b01 or 2'b10), ALU must wait up to 16 clock cycles for the second operand.If the second operand arrives within the 16-cycle window, the operation is performed; otherwise, the err flag must be raised.
* For multiplication operations (INC\_MUL, SHIFT\_MUL), result must be generated after 3 clock cycles once both operands are valid.Multiplication operations must also follow the same 16-cycle wait logic when only one operand is valid initially.
* For single-operand operations (e.g., INC\_A, NOT\_A, SHL1\_A), result must be produced after 1 clock cycle once the correct operand is valid.Single-operand operations do not involve any waiting mechanism or 16-cycle timeout logic.

1. Flag Output Verification:

* COUT : carry out for addtion
* OFLOW : overflow for subtraction
* E, G, L : comparison flags.
* ERR : timeout, invalid combinations, or command-specific constraints

1. Output assertions

* No result (RES) or flag should update unless Clock Enable (CE) is high.
* After reset (RST), all outputs must be cleared to zero or set to their default values.
* After reset (RST), all outputs must be cleared to zero or set to their default values.
* For rotate and shift operations, if OPB[7:4] contains any '1', the ERR flag must be asserted.
* Even when OPB[7:4] is invalid, the operation must proceed using the lower bits OPB[2:0] to determine the shift/rotate amount.

1. Corner case and exception handling

* Max and Min operand values (e.g., all 0’s and all 1’s)
* Handle simultaneous application of reset and input transitions without corrupt output.
* Detect and reject undefined or unsupported CMD values with proper ERR signaling.
* For rotate operations, assert ERR if OPB[7:4] contains 1s but still process the rotation using OPB[2:0].
* During the 16-cycle wait period for two-operand commands, if operand values change, the ALU must use the latest updated inputs once both become valid.But in this case the previous values of command and mode should not be changing.
* Ensure proper operation even when inputs are applied just before the 16-cycle timeout.
* Handle back-to-back different operations without result overlap or timing violations.

1. Coverage Closure

* Achieve full functional coverage of all CMD types in both mode = 0 (logical) and mode = 1 (arithmetic).
* Cover all combinations of inp\_valid values: 00, 01, 10, and 11.
* Cover all possible flag outputs: E, G, L, OFLOW, COUT, and ERR.
* Include coverage for scenarios where the ALU enters a 16-cycle wait and either receives the second operand or times out.
* Ensure that operations are exercised with various operand values including corner cases.
* Strive to achieve 100% code coverage including line, branch, toggle, and FSM coverage where applicable.

**DUT interface**

The ALU's Design Under Test (DUT) interface consists of a set of input and output signals that control its operation and communicate the results. The inputs include data operands (OPA, OPB), control signals (CMD, MODE, CE, INP\_VALID), and timing signals (CLK, RST, CIN). These inputs determine the type of operation to be performed and provide the necessary operands. The outputs (RES, COUT, OFLOW, E, G, L, ERR) convey the result of the operation and relevant status flags. The interface is designed to support both single and dual-operand instructions with dynamic timing behavior, and it plays a crucial role in synchronizing the ALU’s functionality with the testbench environment during verification.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SL.No | Pin name | Direction | No.of bits | Description |
| 1. | OPA | INPUT | N | Parameterized operand 1 |
|  | OPB | INPUT | N | Parameterized operand 2 |
|  | CIN | INPUT | 1 | This is the active high carry in input signal of 1-bit |
|  | CLK | INPUT | 1 | This is the clock signal to the design and it is edge sensitive |
|  | RST | INPUT | 1 | This is the active high asynchronous reset to the design |
|  | CE | INPUT | 1 | This is the active high clock enable signal 1 bit |
|  | MODE | INPUT | 1 | MODE signal 1 bit is high, then this is an Arithmetic Operation otherwise it is logical operation |
|  | INP\_VALID | INPUT | 2 | Operands are valid as per below table 00 : No operand is valid  01: Operand A is valid  10: Operand B is valid  11:Both operands are valid |
|  | CMD | INPUT | M | Parametrized Arithematic Commands |
|  | RES | OUTPUT | N+1 | This is the total parameterized plus 1 bits result of the instruction performed by the ALU. |
|  | OFLOW | OUTPUT | 1 | This 1-bit signal indicates an output overflow, during Addition/Subtraction |
|  | COUT | OUTPUT | 1 | This is the carry out signal of 1-bit, during Addition/Subtraction |
|  | G | OUTPUT | 1 | This is the comparator output of 1-bit,which indicates that the value of OPA is greater than the value of OPB |
|  | L | OUTPUT | 1 | This is the comparator output of 1-bit,which indicates that the value of OPA is lesser than the value of OPB |
|  | E | OUTPUT | 1 | This is the comparator output of 1-bit,which indicates that the value of OPA is equal to the value of OPB |
|  | ERR | OUTPUT | 1 | When Cmd is selected as 12 or 13 and mode is logical operation , if 4th ,5th ,6th and 7th bit of OPB are 1, then ERR bit will be 1 else it is zero  Or if both the input does not appear within the 16 clock cycle window ERR flag will be raised. |

**TESTBENCH ARCHITECTURE**

The ALU testbench is built as a modular, layered environment that cleanly separates stimulus generation, DUT observation, result checking, and coverage collection. At its core is an environment that ties together a driver, which translates high‑level transactions into pin‑level stimulus (including operand‑wait and multi‑cycle behavior), and a monitor, which samples the DUT’s outputs each clock. A scoreboard (with a behavioral reference model) compares those outputs against expected results and flags mismatches, while a suite of assertions enforces timing rules in real time. Parallel to checking, a functional coverage collector tracks which commands, modes, timing classes, and error conditions have been exercised. Testcases-both directed and constrained‑random-drive the driver through corner cases, reset sequences, and long random streams. Finally, built‑in logging, waveform dumps, and coverage reports give clear pass/fail summaries and highlight any verification gaps. This structure ensures the ALU is exhaustively tested, easy to debug, and simple to extend or reuse for future enhancements.

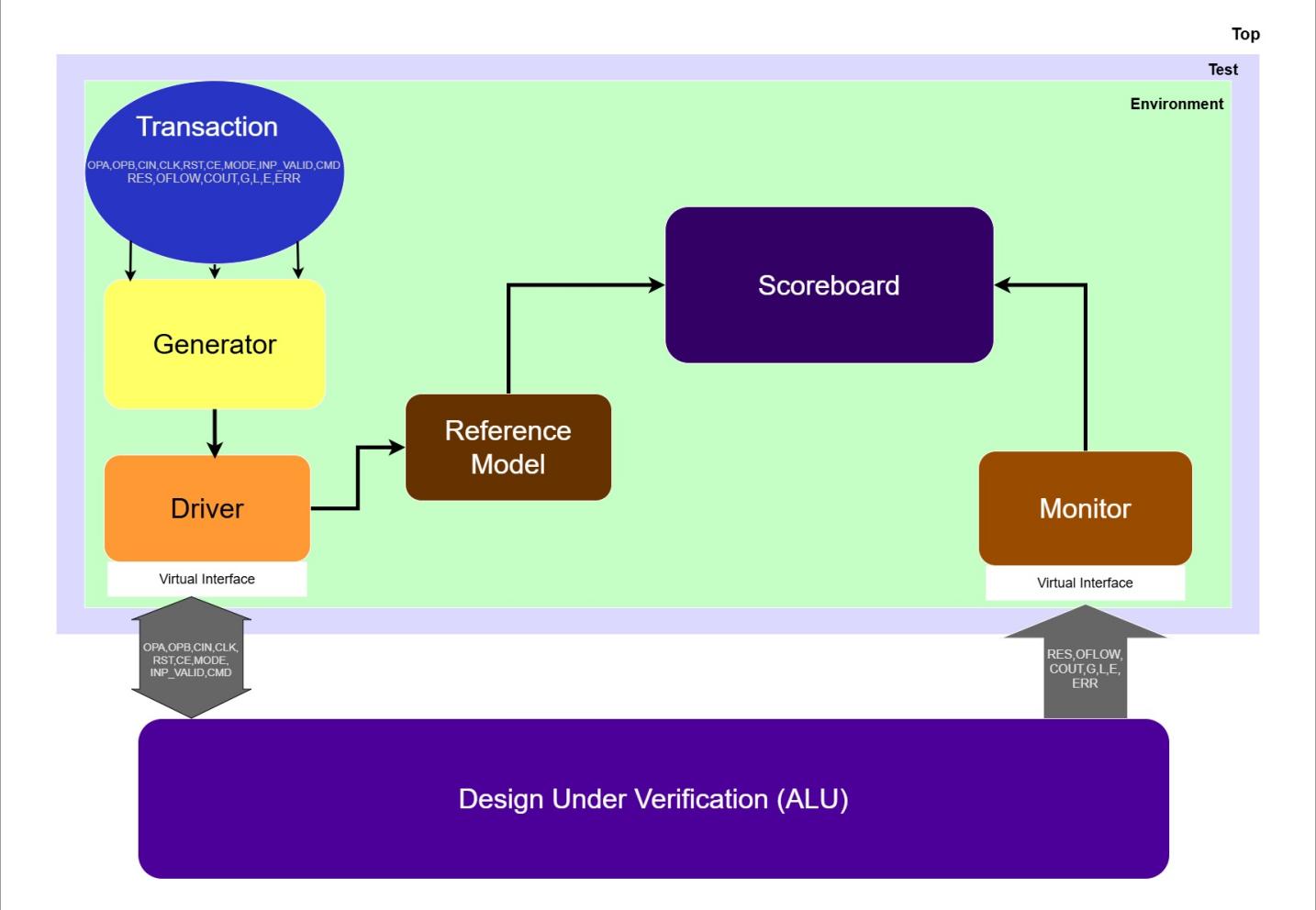


Figure1: ALU testbench architecture

**ALU interface:**

* The interface consists of all the input and output signals of the ALU
* The interface communicates with the driver and monitor in the testbench architecture
* The interface is static, whereas the testbench components driver and monitor which are written inside classes are dynamic. Therefore to connect a static interface with a dynamic testbench component, a Virtual Interface (VI) is used
* The alu\_intf encapsulates all DUT signals and defines dedicated clocking blocks (drv\_cb, mon\_cb, ref\_cb) plus modports (DRV, MON, REF\_SB) so that each testbench component only gets the required signal in the specified direction.

**ALU Transaction :**

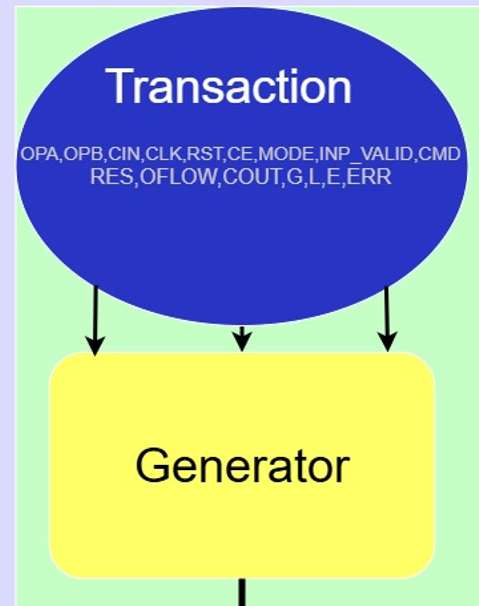


Figure 2: Transaction component

* Transactions consists of the ALU input stimuli which will have to be randomized in the generator, as well as the ALU outputs that are non-randomized.
* Applies distribution constraints to inp\_valid, ce, mode, and cmd to balance test scenarios.
* Provides a copy() method to copy the inputs similar to deep copy operation.
* Serves as the primary data packet between sequencer, driver, monitor, and scoreboard components

**ALU generator:**

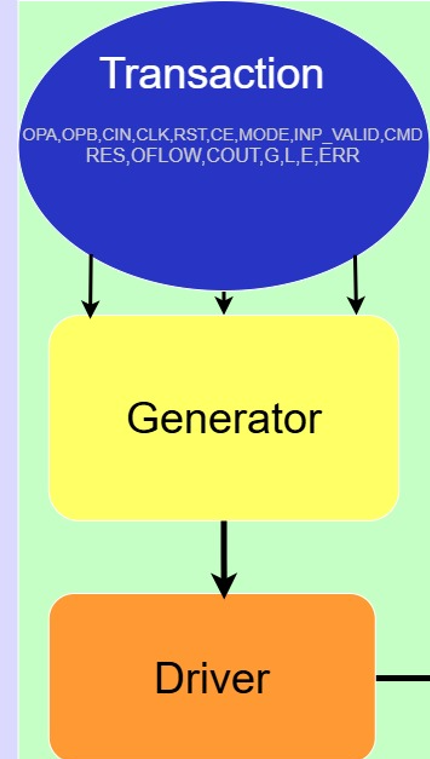


Figure 3: Generator

* The generator consists of the transaction class handle, the mailbox handle which has to be connected to the driver and the task start() which randomizes the transactions and sends the randomized transactions to the driver through the mailbox.
* Connects to the driver via a mailbox #(alu\_transaction) passed into its constructor.
* In its start() task, loops for num\_transaction iterations to generate stimuli.
* Uses blueprint.randomize() to produce constrained‑random input values each cycle.
* Sends a deep copy of the randomized blueprint to the driver mailbox (mbx\_gd.put()).
* Serves as the primary source of stimulus for driving the ALU under test

**ALU driver:**

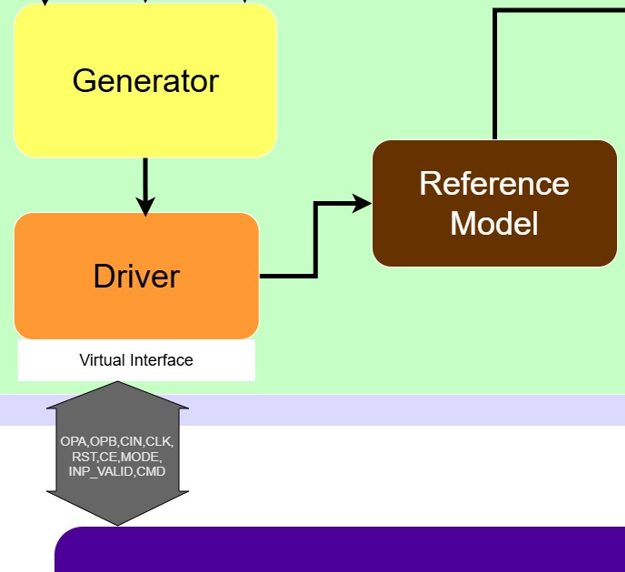


Figure 4: Driver

* Uses two mailboxes: mbx\_gd (from generator) and mbx\_dr (to scoreboard).
* Holds a virtual interface vif\_drv for clock‑aligned driving via drv\_cb.
* In start(), waits reset, then for each transaction:
* Drives inputs immediately for single‑operand or fully valid two‑operand ops.
* For two‑operand ops when only one operand is valid, disable randomization of cmd, mode, and ce, then re‑drive updated opa/opb each clock for up to 16 cycles until both operands become valid.
* After driving, calls mbx\_dr.put() to forward the transaction downstream.

**ALU monitor:**

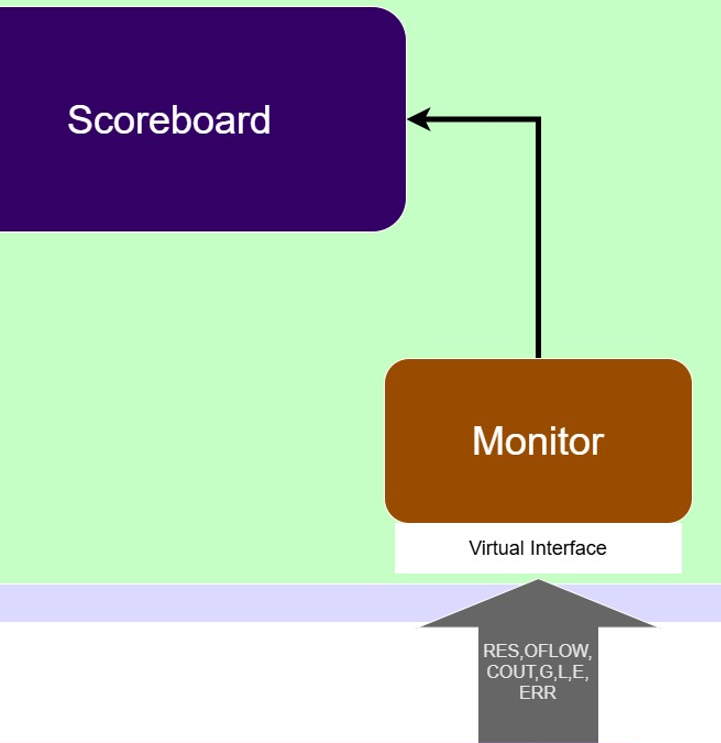


Figure 5: Monitor

* Instantiates a local alu\_transaction (mon\_trans) to hold DUT outputs.
* Connects to the scoreboard via mbx\_ms mailbox.
* Uses the mon\_cb clocking block on vif\_mon to sample outputs synchronously at each rising CLK edge.
* In start(), waits for reset de‑assertion, then loops over num\_transaction cycles.
* On each cycle, captures RES, OFLOW, COUT, G, L, E, and ERR from the interface into mon\_trans.
* Sends the populated mon\_trans to the scoreboard with mbx\_ms.put().

**ALU reference model:**

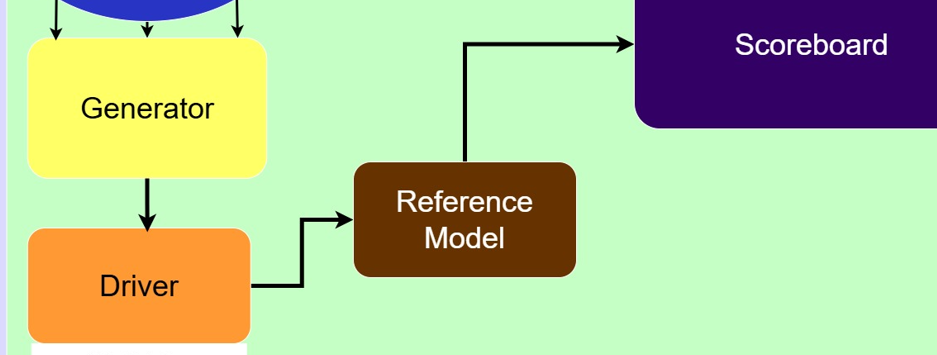


Figure 6: Reference Model

* 2 mailboxes, one which is used to receive transactions from the driver and the other which is used to send transactions to the scoreboard.
* Reads stimulus transactions from mbx\_dr and drives them through the reference model.
* Uses the ref\_cb clocking block on vif\_ref to align model evaluation with DUT’s clock.
* Implements the ALU’s multi‑cycle behavior: 1‑cycle latency for normal ops, 3‑cycle for multiplication, and up to 16‑cycle wait for two‑operand operand arrival.
* Contains a cycle‑accurate behavioral model that mirrors all arithmetic and logical commands, including flag (COUT, OFLOW, E, G, L, ERR) computations.
* Handles reset by zeroing all outputs when reset is asserted.
* After computing results and respecting the required delay, writes the completed ref\_trans back to mbx\_rs for scoreboard comparison.
* Ensures error conditions (timeout, invalid commands, operand‑validity violations) are flagged exactly as the DUT specification dictates.

**ALU scoreboard:**

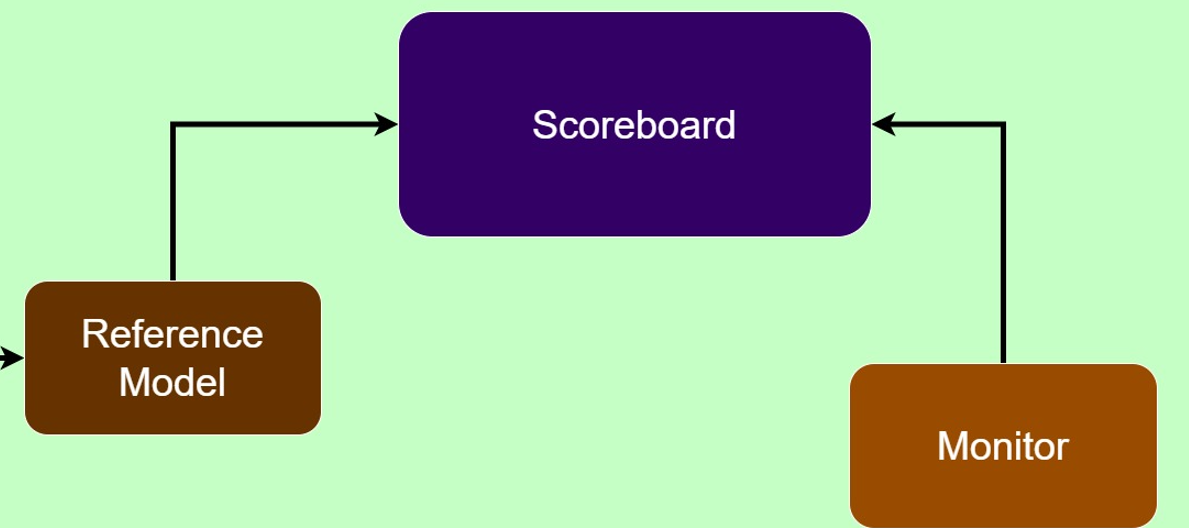


Figure 7: Scoreboard

* 2 mailboxes, one which is used to receive expected transactions from the reference model, and the other which is used to receive the actual transactions from the monitor.
* 2 tasks, one to collect the transactions from the reference model and the monitor, and the other to compare these transactions and report whether the data matches or mismatches.

**ALU environment:**

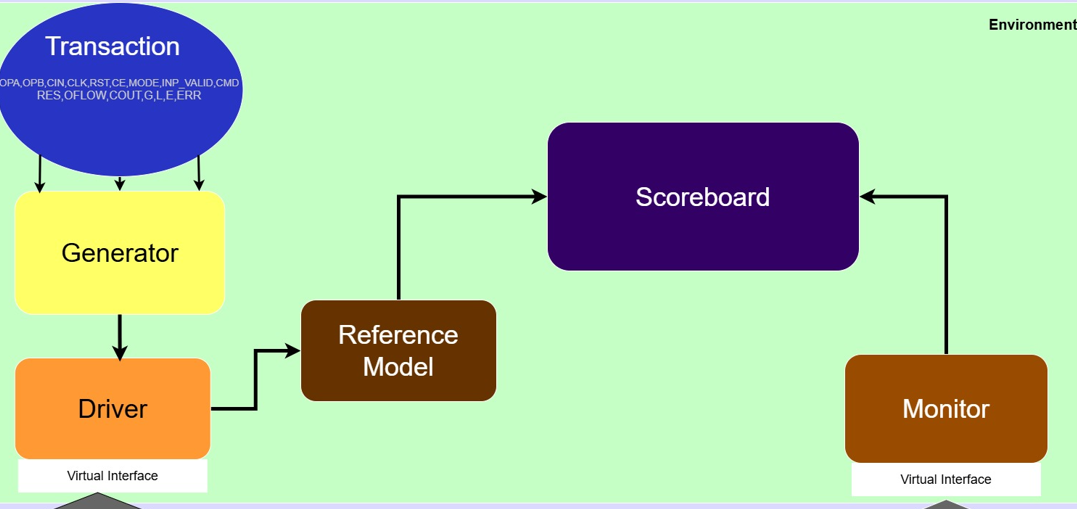


Figure 8: Environment

* 3 virtual interfaces – one for the driver, one for the monitor and one for the reference model.
* 4 mailboxes - a mailbox for the generator to driver connection, a mailbox for the driver to reference model connection, a mailbox for the reference model to scoreboard connection, and, a mailbox for the monitor to scoreboard connection.
* Handles for generator, driver, monitor, reference model and scoreboard.
* A task called build() which creates objects for all the mailboxes and components and a task start() which calls all the start methods.

**ALU test:**

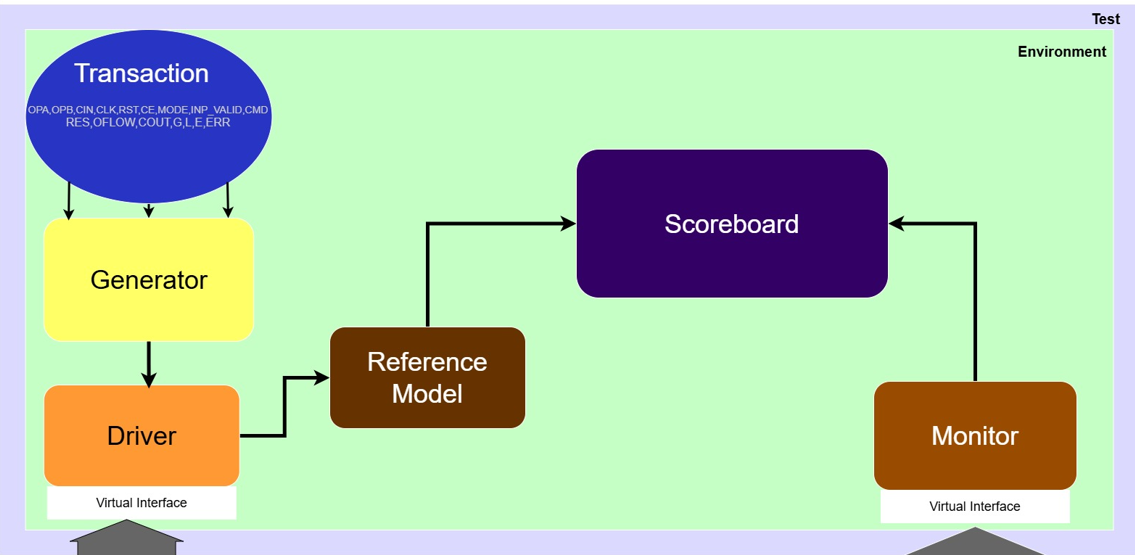


Figure 9: test

* 3 virtual interfaces, one for the driver, one for the monitor and one for the reference model.
* A handle for the environment.
* Code that explicitly overrides the new() constructor to connect the virtual interfaces from driver, monitor and reference model to test.
* A task called run, which builds the object for the environment handle and calls the build and start methods of the environment.

**ALU top module:**

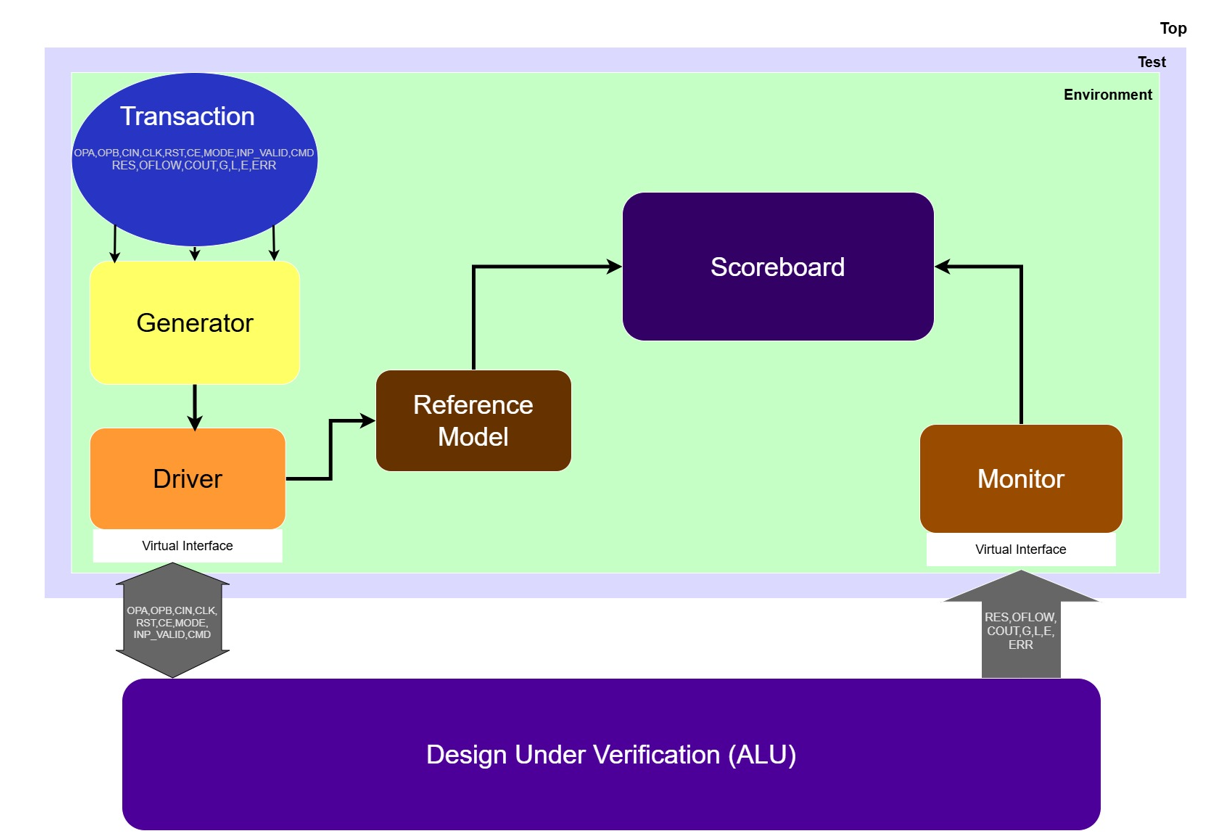


Figure 10: Top

* Importing the ALU package.
* Clock generation code.
* Reset generation code.
* Interface, DUV (ALU) and test instantiation.
* An initial block to run the test(s).