ALU Verification Plan

ALU

The Arithmetic Logic Unit (ALU) is a crucial component in digital processing systems. It is responsible for executing fundamental operations such as arithmetic calculations (like addition, subtraction, increment, and decrement) and logical evaluations (such as AND, OR, XOR, and NOT). Integrated into microprocessors and embedded systems, the ALU enables essential data manipulation and decision-making capabilities.

PROJECT OVERVIEW

The goal of this project is to verify ALU that can perform various arithmetic and logic functions. It includes custom commands, shift/rotate functionality, and intelligent error detection.

We started by understanding the functional requirements such as the need for ADD, SUB, CMP, INC, DEC, AND, OR, XOR, SHL, SHR, ROL, and ROR. These were mapped to the CMD signal with proper encoding for both arithmetic and logical modes.

Inputs like operand A and B, clock, reset, input validity, and carry-in are driven into the ALU. Based on the command selected and the operation mode, the ALU generates results along with flags like overflow, equality, carry, and error.

If the operand A is valid and if we get operand B with in or at 16th clock cycle then the particular operation is valid otherwise the error flag will set high.

In this project, our main focus is on verifying it using a System Verilog based testbench. This ensures our ALU behaves correctly for all supported operations and under all possible input conditions.

VERIFICATION OBJECTIVES

The primary objective of the verification process is to confirm the correctness, stability, and completeness of the ALU under a wide range of scenarios. This includes validating the output for all defined operations, ensuring correct propagation of flags such as carry-out, overflow, and comparison results, and verifying robustness against invalid inputs and timing violations. Additionally, the goal is to test the ALU under different conditions, such as changing inputs quickly or giving the same input in different ways, to ensure the design is stable and works in all possible situations. Apart from functional checks, the verification also includes making sure that all types of operations are tested (functional coverage) and that written checks (assertions) are triggered correctly when something goes wrong.

DUT INTERFACES

The alu interface defines a SystemVerilog interface that encapsulates all input and output signals between the testbench and the DUT (Design Under Test). This interface ensures clean, modular, and synchronized communication using clocking blocks and modports.

The ALU interface include:

INPUTS:

- OPA, OPB: Operand inputs (parameterized width N)
- CMD: Operation command (4 bits to select the desired ALU operation)
- INP_VALID: Indicates which operands are valid (00, 01, 10, 11)
- MODE: 1 for arithmetic, 0 for logical operations
- CIN: Carry input used in arithmetic operations
- CLK: Clock signal, positive edge triggered
- RST: Asynchronous reset
- CE: Clock enable

OUTPUTS:

- RES: Result of the ALU operation
- COUT: Carry-out from addition/subtraction
- OFLOW: Indicates overflow
- ERR: Indicates invalid conditions (e.g. wrong rotation inputs)
- G, L, E: Comparison flags (Greater, Less, Equal)

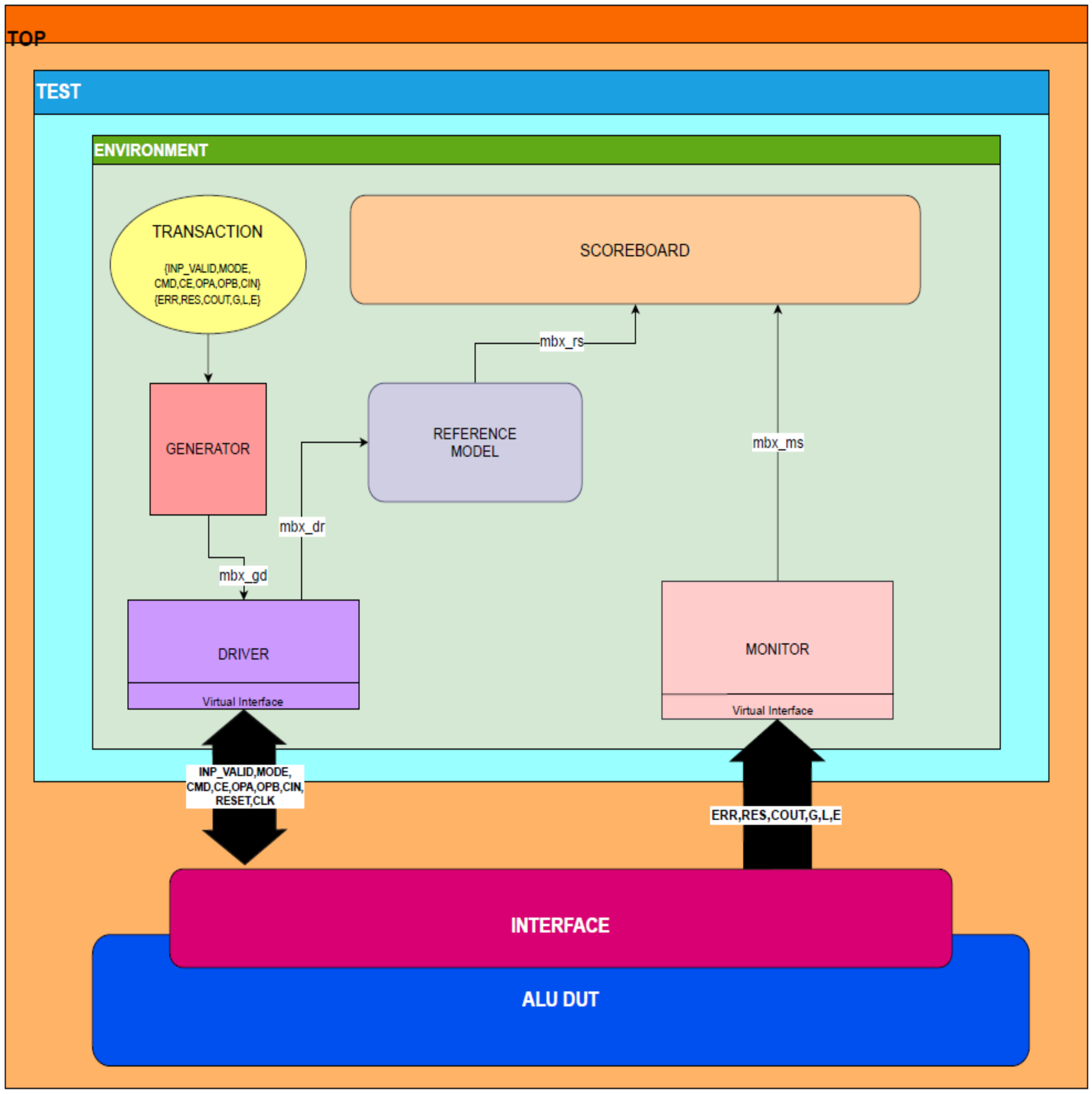
MODPORTS:

The modport groups and specifies the direction to the signals specified within the interface.

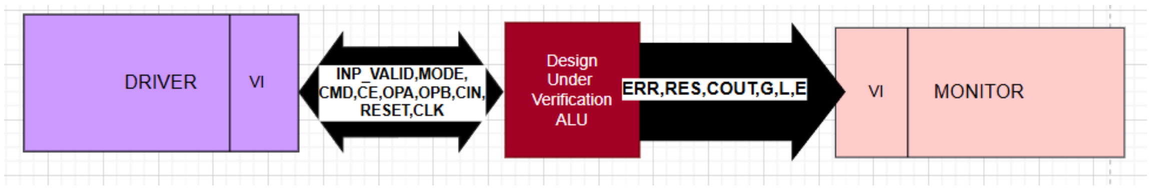
CLOCKING BLOCK:

A clocking block in SystemVerilog defines the timing and synchronization for signal interaction between the testbench and DUT. It specifies a clock event as a reference, the signals to be sampled or driven, and the timing of these actions relative to the clock. This helps ensure stable and predictable communication between the testbench and DUT.

TESTBENCH ARCHITECTURE



1.INTERFACE

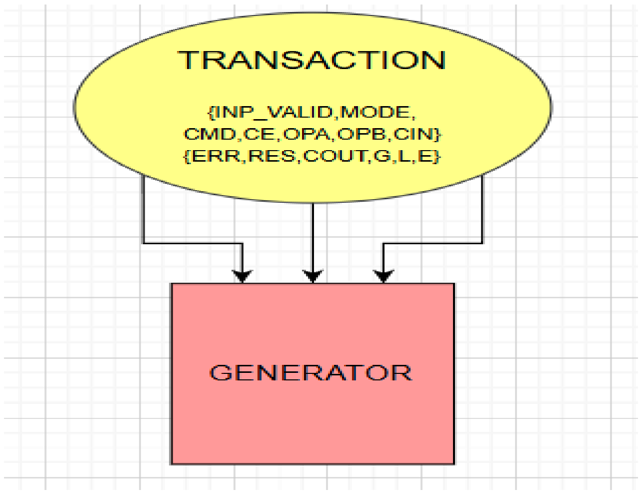

DRIVER
VI
INP_VALID,MODE, CMD,CE,OPA,OPB,CIN, RESET,CLK
Design Under Verification ALU
ERR,RES,COUT,G,L,E
VI
MONITOR

An interface is a bundle of signals or nets through which a testbench communicates with a design. The interface construct is used to connect the design and testbench.

2.VIRTUAL INTERFACE

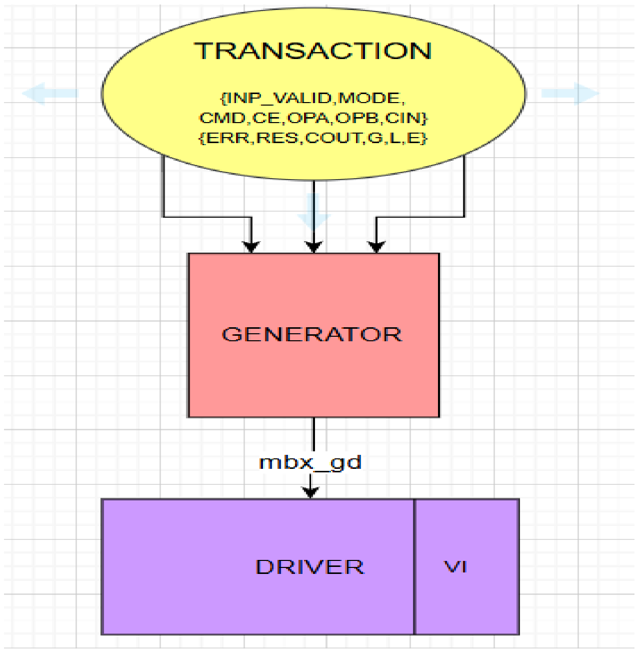
A virtual interface is a variable of an interface type that is used in classes to provide access to the interface signals. A virtual interface is a variable that represents an interface instance.

3.TRANSACTION



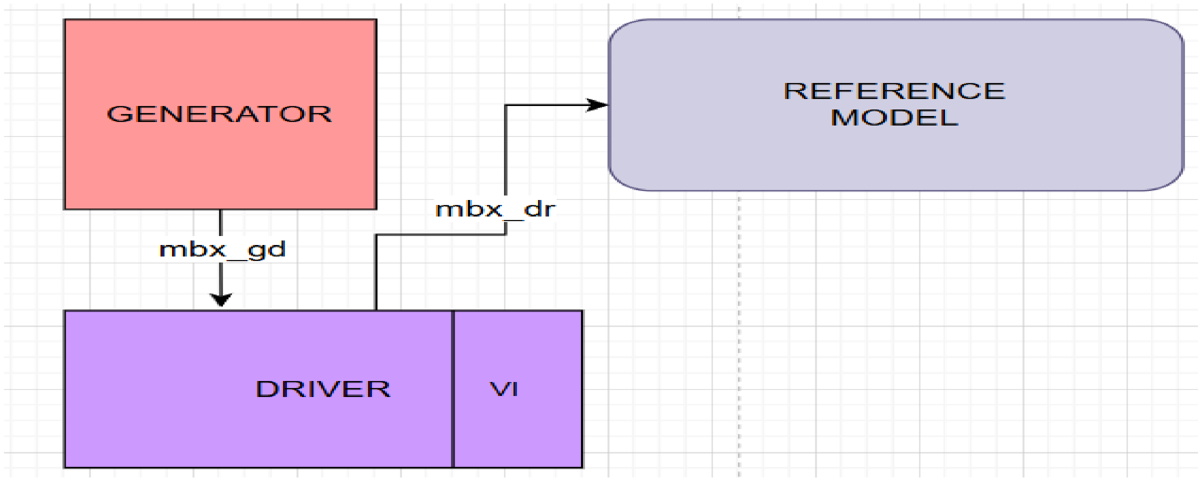
A transaction is a class-based object that encapsulates all the fields required to stimulate the DUT, such as OPA, OPB, CMD, MODE, CE, CIN, and INP_VALID. It represents a single operation or test instance and serves as the communication unit between various testbench components.

4.GENERATOR



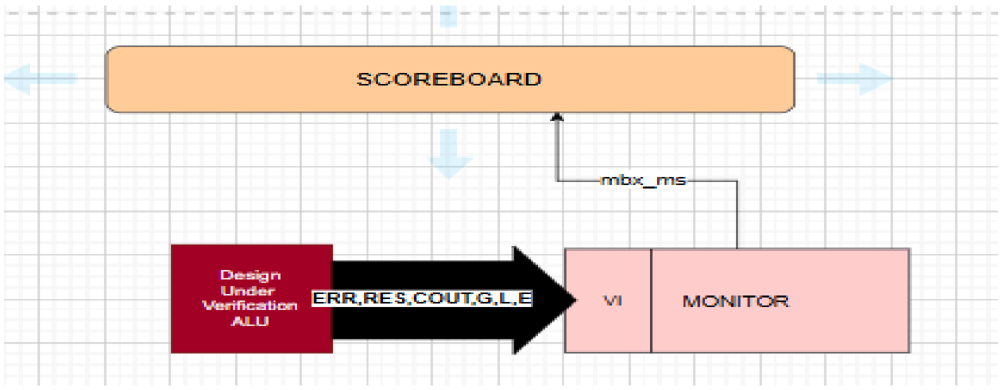
The generator is responsible for creating transactions, either randomly or through directed constraints. It introduces diversity and randomness to the test environment, ensuring that all possible corner cases are explored. It sends these transactions to the driver via a mailbox.

5.DRIVER



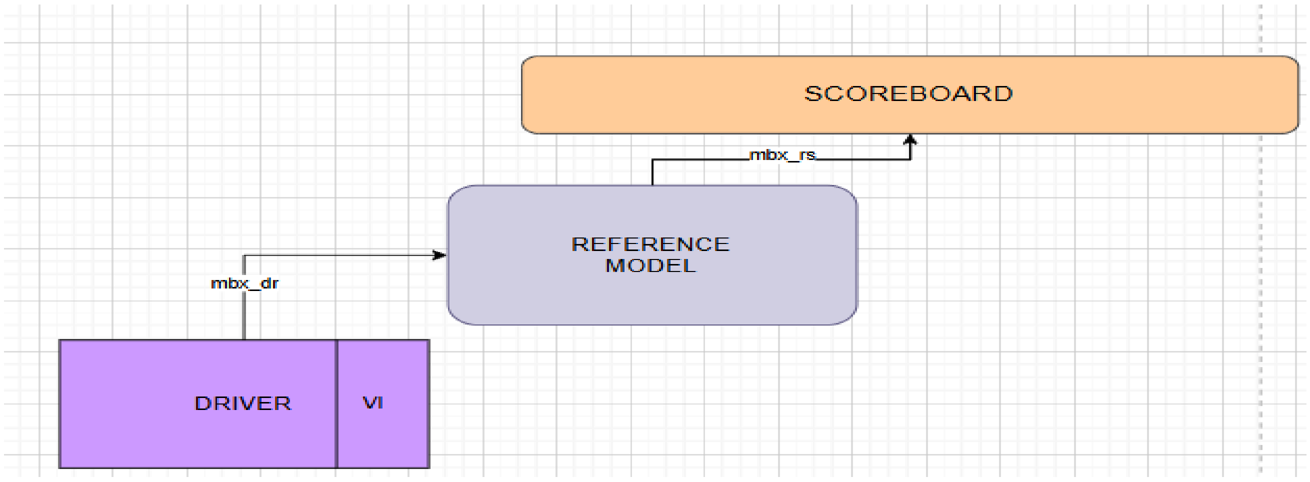
The driver receives the transactions from the generator and translates the high-level transaction data into pin-level signal activity. It drives the input signals of the DUT using the values from the transaction. Additionally, the driver forwards a copy of the transaction to the reference model for generating expected results.

6.MONITOR



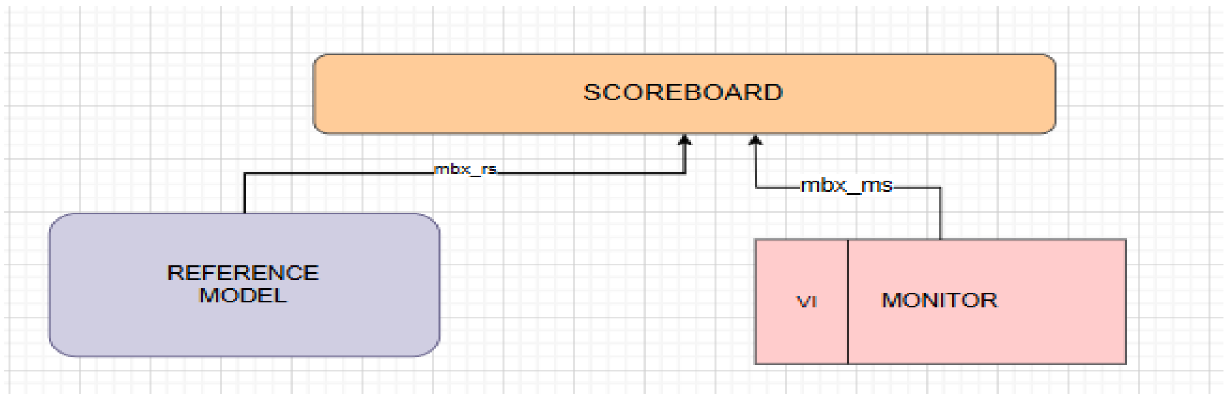
The monitor passively observes the signals on the DUT outputs and captures the results after the ALU completes an operation. It then converts these outputs into transaction-level objects and sends them to the scoreboard for checking.

7.REFERENCE MODEL



The reference model is a behavioral model of the ALU, written to mimic the functional intent of the DUT. It receives the same inputs as the DUT and calculates the expected result, which is sent to the scoreboard.

8.SCOREBOARD



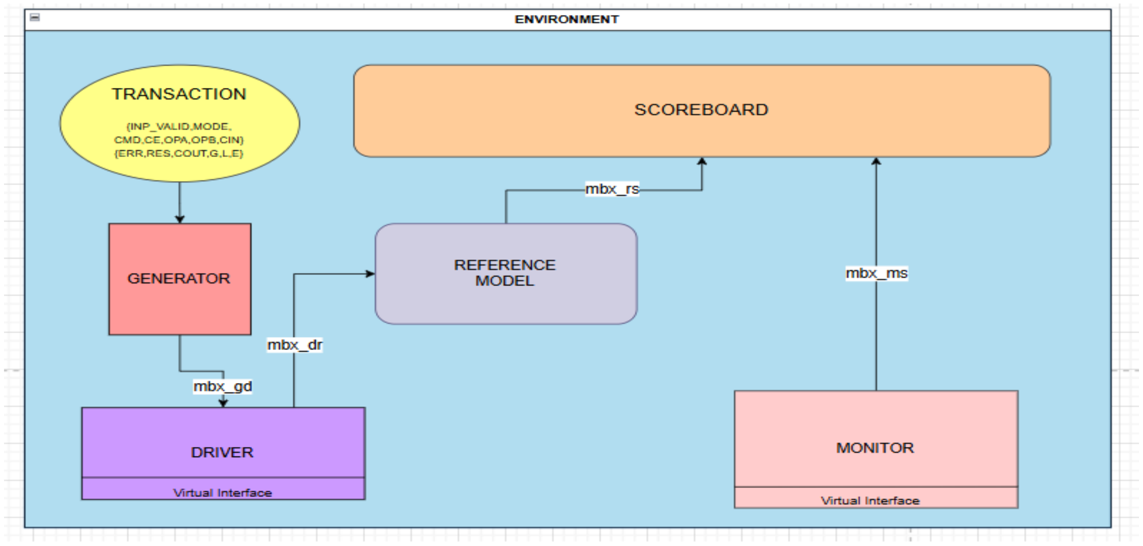
The scoreboard performs the actual verification check. It compares the expected outputs received from the reference model with the actual DUT outputs captured by the monitor. It logs discrepancies, passes, or failures, helping in debugging and validating the DUT.

9.MAILBOXES

Mailboxes are used for communication between testbench components. Each component pair (e.g., generator to driver, monitor to scoreboard) communicates through its dedicated mailbox. This ensures safe, synchronized, and transaction-level data transfer without race conditions.

- mbx_gd: Generator to Driver
- mbx_dr: Driver to Reference Model
- mbx_rs: Reference Model to Scoreboard
- mbx_ms: Monitor to Scoreboard

10.ENVIRONMENT



The environment is a wrapper that instantiates and connects all reusable verification components including the generator, driver, monitor, scoreboard, reference model, and mailboxes. It acts as the backbone of the testbench, managing the flow of transactions and results between components. The environment ensures modularity, reusability, and maintainability of the verification structure. It is also responsible for coordinating communication between class-based components using the interface and virtual interface.

11.TEST

This is the component where different test cases are written and run. Here the environment is instantiated and built.

12.TOP

This is the top most module of the SV testbench architecture where control signals like clock and reset are generated. Its code is written inside a module and the interface, the DUV and the test are instantiated in it

TEST PLAN

The testing process begins by checking all the arithmetic operations supported by the ALU. We validate common operations like addition and subtraction, as well as those involving a carry or borrow input. Increment and decrement operations are tested for both operands (OPA and OPB), including extreme values like maximum and minimum, to ensure the ALU handles edge cases properly without incorrect overflow or wrap-around results.

Next, we move to logical operations. This includes standard bitwise operations like AND, OR, XOR, and their inverses like NAND and NOR, along with unary NOT operations on both operands. These tests ensure that each bit position behaves as expected for different operand values.

We also evaluate shift and rotate instructions. Right and left shifts (SHR and SHL), and right and left rotates (ROR and ROL) are tested with different operand values. If the high nibble (bits 7 to 4) of operand B contains invalid values during a rotate operation, the ALU should raise the ERR flag—this checks the design's robustness against incorrect command usage.

The comparison command (CMP) is verified by ensuring the greater-than (G), less-than (L), and equal (E) flags are set accurately depending on the relationship between OPA and OPB. We test each condition individually.

We then explore timing and input validity. Since the operands might not always arrive at the same time, especially in pipelined or asynchronous designs, we simulate different INP_VALID combinations and delays to verify the ALU processes inputs only when both are ready. A timeout mechanism is tested where, if the second operand doesn't appear within 16 clock cycles after the first, the ALU should assert the ERR flag to indicate the failure.

Finally, reset and clock-enable behaviors are tested. When the reset signal (RST) goes high, all ALU operations and outputs should be cleared instantly, regardless of the clock. If the clock enable (CE) signal is low, the ALU should pause any operation and hold the current state until CE becomes high again.

Functional Coverage Plan:

All possible signal combinations are covered: INP_VALID values (00–11), all CMD values (0–13), and both arithmetic and logic modes. Control signals like CE, CIN, and RST are toggled. Operand and result ranges are fully exercised. Flags like ERR, COUT, OFLOW, G, L, and E are verified across different inputs. CMD × MODE cross-check ensures every operation is tested in both modes.

Assertions:

Assertions check CMD range and INP_VALID consistency. They verify that the ALU only acts when all inputs are valid. Reset clears outputs immediately. In logic mode, COUT and OFLOW stay low. G, L, and E flags are checked to never overlap. ERR must be high for invalid rotate operations. RES should not change if CE is low or INP_VALID is 00.