ALU Verification Plan

PROJECT OVERVIEW

Arithmetic logic unit (ALU) is a combinational digital circuit that performs arithmetic and bitwise operations on integer binary numbers. It is responsible for carrying out arithmetic operations such as addition and subtraction, as well as logical operations like AND, OR, and NOT. This project presents the implementation of a parameterized ALU designed using syst em verilog. The ALU supports a wide range of operations including arithmetic, logical, shift, and rotate instructions. The goal is to create a modular, synthesizable, and simulation-verified ALU that can serve as a reliable component in larger digital design projects.

VERIFICATION OBJECTIVES

The main objective of verifying the ALU is to check that it performs all its supported operations correctly. This includes making sure that arithmetic operations like addition, subtraction, and increment, as well as logical operations like AND, OR, and NOT, produce the correct output for all types of inputs. It is also important to check that the output flags—such as overflow, carry, and comparison flags (greater, less, equal)—are set properly depending on the operation. 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)

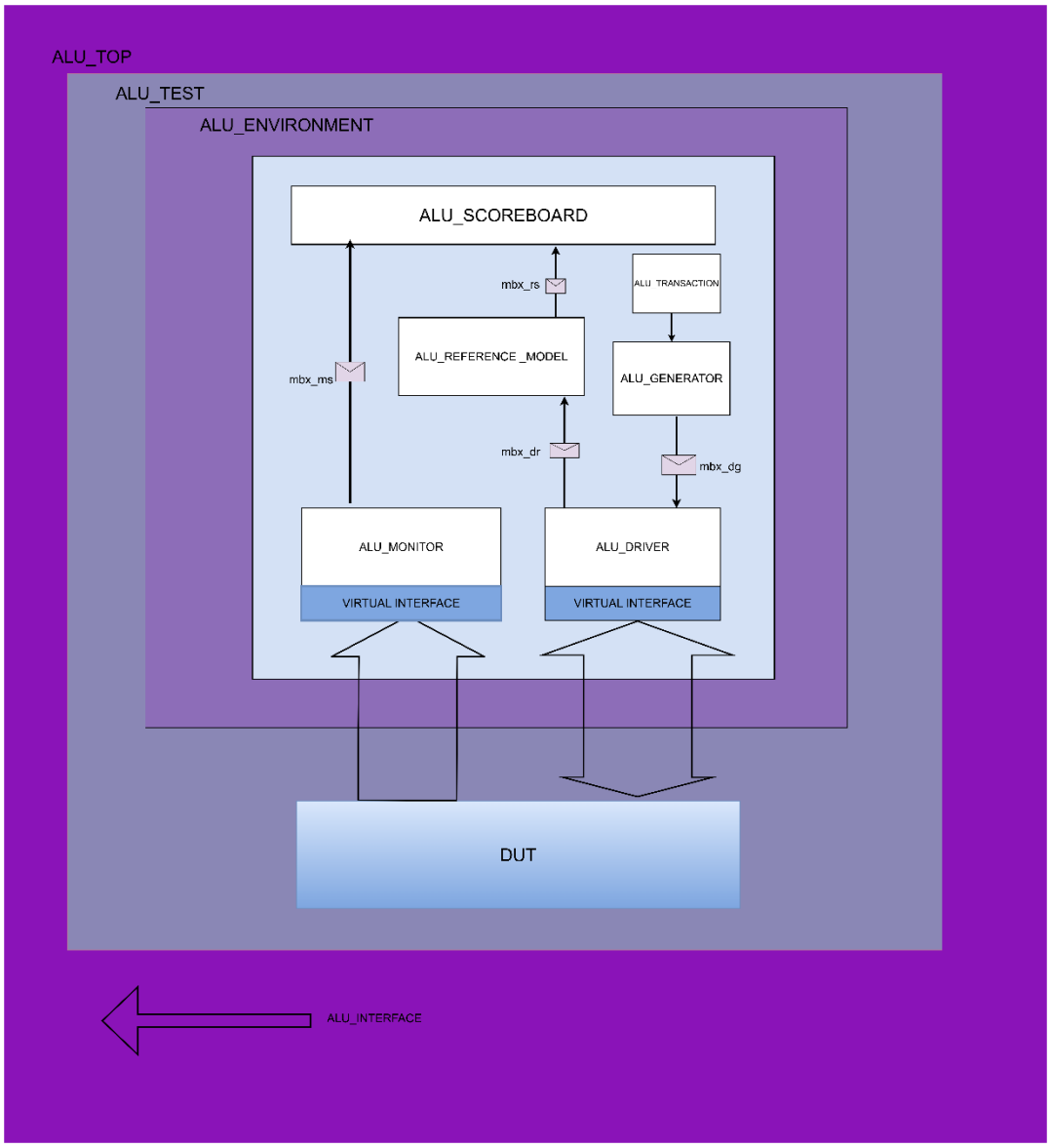
MODPORTS:

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. Top Module – TEST

The TEST module acts as the top-level wrapper that instantiates the test environment, connects it to the DUT, and manages simulation control.

2. Environment

The environment contains all the essential components such as the driver, monitor, generator, scoreboard, reference model, and communication mailboxes. These elements work together to provide stimulus, capture outputs, generate expected

results, and perform verification comparisons.

3. Generator

The Generator creates transactions including fields like opa, opb, mode, cmd, inp_valid, ce, cin, etc. It sends the transactions to the Driver via the mailbox mbx_gd.

4. Driver

The Driver receives the randomized values from the generator and drives it to the DUT through the interface. It also sends the values to the reference model through mailbox.

5. Monitor

The Monitor observes and captures the DUT outputs such as res, cout, oflow, err, g, l, and e. It sends this data to the Scoreboard (via mbx_ms) for comparison.

6. Reference Model

It receives transactions and performs the expected operation, and sends the results to the Scoreboard.

7. Scoreboard

The Scoreboard performs comparison between actual DUT outputs and the expected outputs from the Reference Model. It logs pass/fail information for each transaction and reports mismatches.

8. Mailboxes

Mailboxes provide a transaction-level communication channel between various testbench components. They include:

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

9. Design Under Test (DUT)

The DUT receives driven inputs from the Driver and sends responses back to the Monitor. It is treated as a black-box connected through virtual interfaces.

10.INTERFACE

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.

11.VIRTUAL INTERFACE

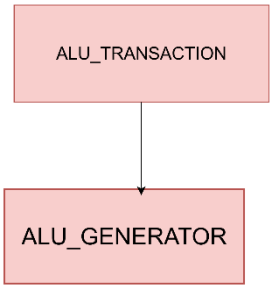
A virtual interface is a variable of an interface type that is used in classes to provide access to the interface signals. A virtual interface is a variable that represents an interface instance.

12.TRANSACTION

Transaction consists of ALU inputs that need to be randomized and outputs that are not randomized.

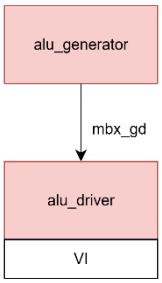
FLOWCHART OF SV COMPONENTS

1.Transaction class



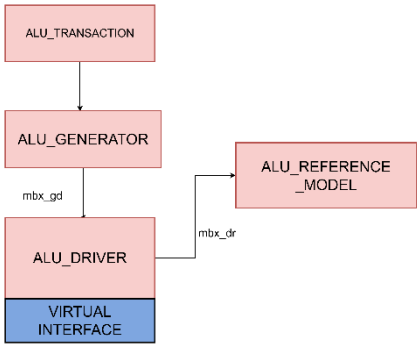
The transaction class consists of randomized inputs and non-randomized outputs.
It consists of constraints and a deep copy function for creating transaction copies.

2.Generator class



This component generates randomized input values and is sent to the driver through a mailbox(mbx_gd).

3.Driver class



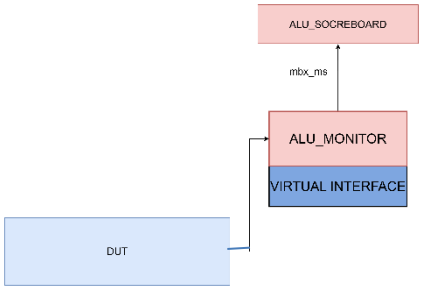
It consists of two mailbox:

Generator to driver mailbox(mbx_gd): It transfers randomized transactions from the generator to the driver

Driver to reference model mailbox(mbx_dr): It sends the same transactions to the reference model

The task applies the stimulus to the DUV based on the received transaction

4.Monitor class



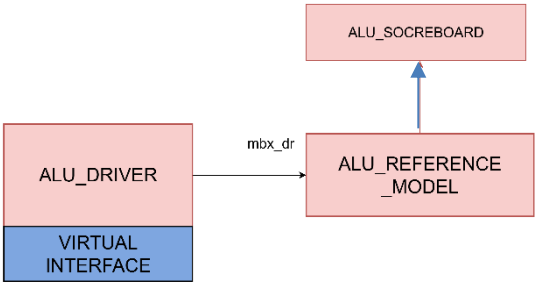
The ALU Monitor observes the DUT's outputs and converts them into transaction objects for validation.

It consists of one mailbox:

Monitor to Scoreboard Mialbox: It transfers captured output transactions to the scoreboard

Detects valid output transactions, samples the result, and sends it as a transaction object to the scoreboard

5.Reference model



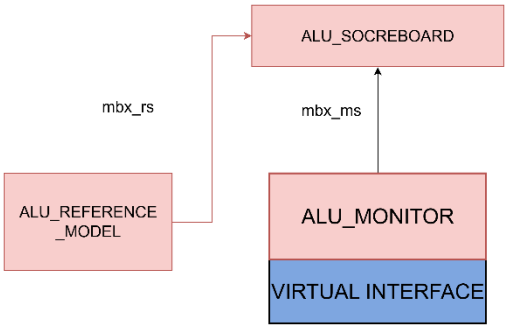
Acts as the reference for result validation. It processes the same inputs as the DUV and produces the expected outputs.

It consists of two Mailboxes:

Driver to reference model mailbox: It receives transactions from the driver

Reference to scoreboard mailbox: It sends computed expected results to the scoreboard

6.Scoreboard



The scoreboard verifies the correctness of the DUV by comparing its outputs with the reference model's expected results.

It consists of two mailbox:

Reference model to scoreboard mailbox : It receives the expected results from the reference model

Monitor to Scoreboard mailbox: It receives actual results from the monitor

Compares the actual and expected results, and flags mismatches and generates final statistics, including pass/fail counts and functional coverage details.

TEST PLAN

Test Scenarios:

The test process begins with validating all arithmetic operations. This includes fundamental functions such as addition (ADD) and subtraction (SUB), along with their variants that include carry-in and borrow-in functionality. Increment (INC) and decrement (DEC) operations are tested for both input operands (OPA and OPB), covering edge cases such as maximum and minimum values to ensure proper handling without overflow or unexpected results.

Following arithmetic testing, all logical operations are verified: AND, OR, XOR, NAND, NOR, XNOR, as well as unary operations like NOT_A and NOT_B. These checks confirm correct bitwise logic behavior across all bit positions.

Shift and rotate functionalities are then tested, including right shift (SHR), left shift (SHL), rotate right (ROR), and rotate left (ROL). Additionally, invalid command codes such as unsupported values in the high nibble of OPB (OPB[7:4]) are tested to confirm that the ALU correctly flags these as errors (ERR).

Comparison operations are verified through the CMP command, which sets the greater-than (G), less-than (L), or equal (E) flags. Each possible comparison outcome is tested to confirm that the flags correctly represent the relationship between operands.

Input validity and timing behavior are also evaluated. Since the INP_VALID signal may be asserted even if the operands do not arrive simultaneously, scenarios with varied

delays are tested to ensure the ALU begins processing only when both inputs are fully received.

A timeout condition is introduced where, if the second operand is not received within 16 clock cycles after the first, the ALU must assert the ERR flag, ensuring proper handling of incomplete input sequences.

Finally, reset and clock enable behaviors are validated. Activation of the reset (RST) signal must cancel any ongoing operations and reset all outputs. If the clock enable (CE) signal is low, the ALU should pause its operation, preserving its current state until CE is reasserted.

Functional Coverage Plan

The ALU functional coverage plan includes monitoring of key input, output, and control signals to ensure comprehensive verification. Coverage for INP_VALID ensures all possible values (00, 01, 10, 11) are exercised. The CMD signal must cover all 14 ALU operations (0 to 13). Control signals like CE, CIN, and RESET are checked for toggle activity between 0 and 1. The MODE signal ensures both arithmetic and logic modes are tested. Operand inputs OPA and OPB should each span the full value range from 0 to (2^N)-1, while the result RES must cover from 0 to (2^(N+1))-1. Output flags such as ERR, COUT, and OFLOW are verified for correct assertion under relevant conditions. Comparison flags—E, G, and L—are checked for proper behavior when OPA == OPB, OPA > OPB, and OPA < OPB, respectively. Lastly, cross coverage between CMD and MODE (CMD_X_MODE) ensures that each operation is tested under both arithmetic and logic modes, totaling 28 cross bins.

Assertions:

We verify signal validity by checking that CMD stays within its range and INP_VALID correctly reflects input readiness. Timing alignment assertions ensure CMD, OPA, and OPB arrive together as expected. Reset (RST) behaviour assertions guarantee an immediate clear of all outputs and flags, regardless of the clock. For functional checks, logical modes keep COUT and OFLOW low, and comparison flags (G, L, E) are mutually exclusive. An error condition is asserted if CMD is 12 or 13 while OPB[7:4] isn’t 0000, forcing ERR high. Finally, result stability is checked so RES remains unchanged when CE is low or INP_VALID is 00.