ALU VERIFICATION PLAN

-SHREYA

6108

ALU:

The Arithmetic Logic Unit, commonly known as the ALU, is a digital circuit that performs both arithmetic operations like addition and subtraction, and logical operations like AND, OR, and XOR. It forms a fundamental part of every computer system, microcontroller, or processor where data processing is involved.

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 16^{th} 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.

DUT INTERFACE:

The ALU has a clearly defined set of input and output signals. The primary inputs are OPA and OPB (the two operands), CMD (the command), MODE (arithmetic/logical), INP_VALID (validity of operand input), CLK (clock), RST (reset), CIN (carry in), and CE (clock enable).

On the output side, RES gives the result of the operation, and several 1-bit status outputs (OFLOW, COUT, ERR) indicate special conditions like overflow, carry out, or invalid operation. The comparison outputs G, L, and E show the result of comparing operand A and B.

VERIFICATION OBJECTIVE:

The primary goal of this verification is to ensure the correctness, reliability, and completeness of the ALU functionality as described in the design document. The verification process will validate each supported operation, check proper flag generation, handle error scenarios, and ensure the design meets timing and interface requirements.

A key objective is to test all arithmetic and logical operations under various input combinations. This includes operations like ADD, SUB, INC, DEC, AND, OR, XOR, shifts, and rotations. Both valid and invalid command sequences will be applied to confirm correct outputs and flag behavior. Each CMD value will be verified for both arithmetic (MODE=1) and logic (MODE=0) modes.

Another major objective is to verify error-handling logic. The timeout condition (if a second operand is not received within 16 clock cycles) must raise an error, and late input values should be handled correctly using last-priority logic.

The verification must also confirm correct behavior of output flags: COUT, OFLOW, G, L, and E. These outputs are essential for downstream decision-making and must reflect the operation result accurately. For example, the ALU must correctly set the OFLOW flag when an overflow occurs during addition or subtraction, and the comparator flags (G, L, E) must correctly indicate operand relationships.

TESTBENCH ARCHITECTURE:

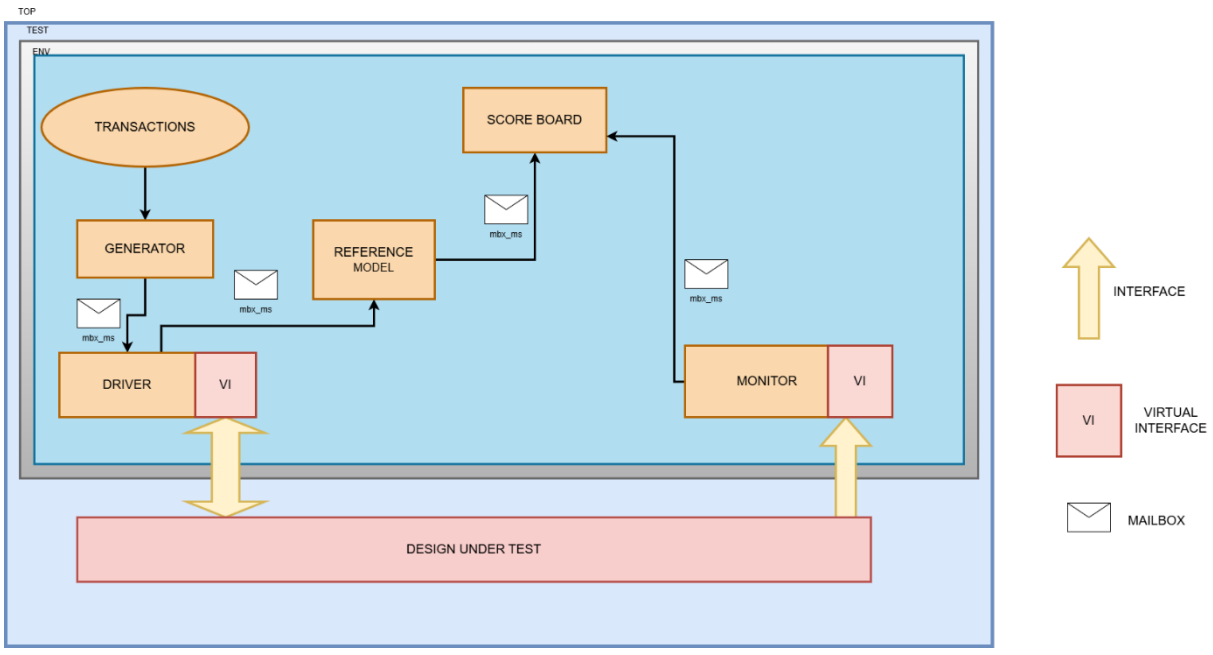

Fig 1: Testbench Architecture

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:

*This is the component where different test cases are written and run

* Here the environment is instantiated and built.

Environment:

*This is the component of the testbench which instantiates the generator, driver, monitor, reference model and scoreboard.

Transactions:

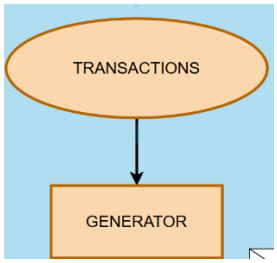

Fig 2 Transaction

* All the inputs and outputs of the Design Under Verification (DUV) are written inside the transaction class, excluding the clock and reset signals that are generated in the top module.

Generator:

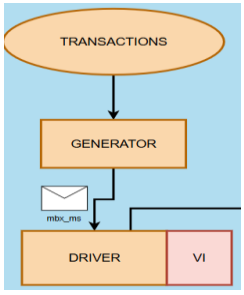

Fig 3 Generator

*This is a component of the testbench which generates constrained random stimuli (transactions) for the DUV.

* The generator sends the generated stimuli to the driver through a mailbox

Driver:

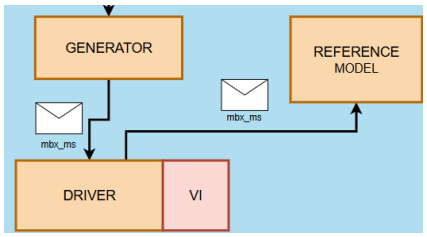

Fig 4 Driver

* It receives the generated transactions from the generator through a mailbox and drives it to the DUV through a virtual interface according to the DUV protocol

*It also sends the received transactions from the generator to the reference model through another mailbox.

Monitor:

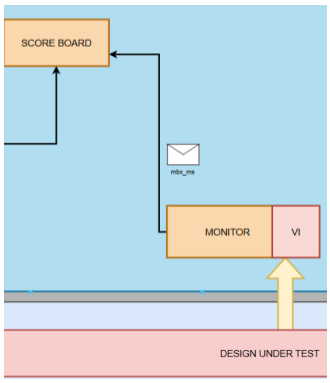

Fig 5 Monitor

* It collects the outputs of the DUV through a virtual interface and sends it to the scoreboard through a mailbox

Reference Model:

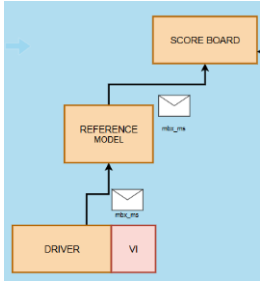

Fig 6 Reference Model

* This is a ‘golden’ model that mimics the functionality of the DUV and generates reference results used for comparison against simulation results.

Scoreboard:

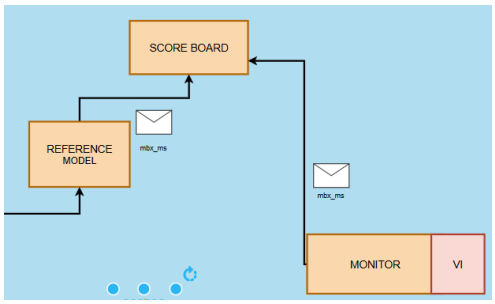

Fig 7 Scoreboard

*This is the component that collects the transactions (expected results) from the reference model through a mailbox.

* It collects the transactions (actual results) from the monitor through another mailbox.

* It compares the expected transactions with the actual transactions and generates a report.

Interface and Virtual interface:

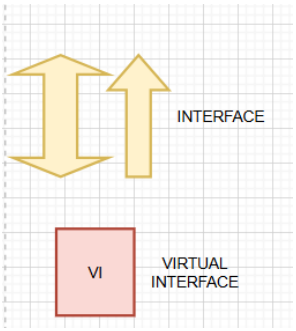

Fig 8 Interface

*The interface consists of all the input and output pins 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.

Mailbox:

Mailboxes provide a traction-level communication channel between various testbench and components.

TEST PLAN:

Test Scenarios:

We begin by verifying all arithmetic operations. This includes basic functions like ADD and SUB, as well as the variants with carry-in or borrow-in . We also test increment (INC) and decrement (DEC) on both operands (OPA and OPB), including edge cases like maximum and minimum values to ensure no overflows or unexpected behavior.

Next, we validate the logical operations: AND, OR, XOR, NAND, NOR, XNOR, along with unary operations NOT_A and NOT_B. These tests confirm bitwise logic works correctly across all bit positions.

We also test shift and rotate functions: right shift (SHR), left shift (SHL), rotate right (ROR), and rotate left (ROL). Any invalid operation codes—for example when the high nibble of OPB (OPB[7:4]) contains an unsupported value—should trigger an error (ERR), checking that invalid inputs are handled gracefully.

Comparisons are covered by the CMP command, which sets the greater-than (G), less-than (L), or equal (E) flags. Each comparison scenario is tested to ensure flags reflect the correct relationship between operands.

We then examine the validity and timing behavior. Whenever the INP_VALID signal is asserted, both operands may not arrive at the same time. We test various delays and combinations to make sure the ALU only processes when inputs are ready.

A timeout scenario is also critical: if the second operand hasn't arrived within 16 clock cycles from the first, the ALU should raise the ERR flag. This ensures proper handling of incomplete inputs.

Finally, reset and clock-enable behaviors are checked. When RST (reset) is activated, it should always cancel ongoing operations and clear outputs. If the clock enables (CE) signal is low, the ALU should temporarily pause operation, holding its current state until CE goes high again.

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.