ALU Verification Plan

Shenoy Apoorva Ganapati

6107

Project Overview:

The Arithmetic Logic Unit (ALU) implemented in this project is a parameterized hardware block capable of executing a wide range of arithmetic and logical operations.

It accepts two operands (OPA and OPB), a command input (CMD), and a mode signal (MODE) that determines whether the operation is arithmetic or logical. In arithmetic mode, the ALU supports basic operations like addition, subtraction, increment, decrement, and comparison, as well as compound operations such as multiplying incremented operands or combining shift and multiplication. In logical mode, it performs bitwise operations including AND, OR, XOR, NOT, NAND, NOR, XNOR, and also supports left/right shift and rotate functions.

Operand validity is managed using a 2-bit INP_VALID signal. If a required operand is missing, the design waits up to 16 clock cycles, after which an ERR signal is raised if the input doesn't arrive. It also includes logic to prioritize the most recently received operands in case of updates during the waiting period. The outputs of the ALU include the result (RES), arithmetic flags such as carry out (COUT) and overflow (OFLOW), comparison flags (G, L, E), and an error flag (ERR) to indicate invalid conditions such as illegal rotate inputs. The design supports parameterization for varying operand widths, making it scalable and adaptable for different use cases.

DUT Interfaces:

The Design Under Test (DUT) consists of the following interface signals:

Inputs:

OPA: Operand A (parameterized width)

OPB: Operand B (parameterized width)

CMD: 4-bit command input specifying the operation

CIN: Carry-in signal used in arithmetic operations

MODE: Selects arithmetic (1) or logic (0) mode

INP_VALID [1:0]: Indicates which operands are currently valid

CE: Clock enable signal

CLK: System clock (positive-edge triggered)

RST: Active-high asynchronous reset

Outputs:

RES: Operation result (parameterized width + 1)

COUT: Carry-out flag for addition/subtraction

OFLOW: Overflow flag for signed operations

G: High if OPA > OPB

L: High if OPA < OPB

E: High if OPA == OPB

ERR: High if operation or operand is invalid (e.g., rotate error or timeout)

MODPORTS:

The modport 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.

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.

Another key part of verification is checking how the ALU behaves when given wrong or incomplete inputs. For example, if one of the required operands is missing or arrives late, the ALU should raise an error. The rotate operations are also tested carefully to make sure they work correctly and raise errors if used with invalid input patterns.

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.

Testbench Architecture:

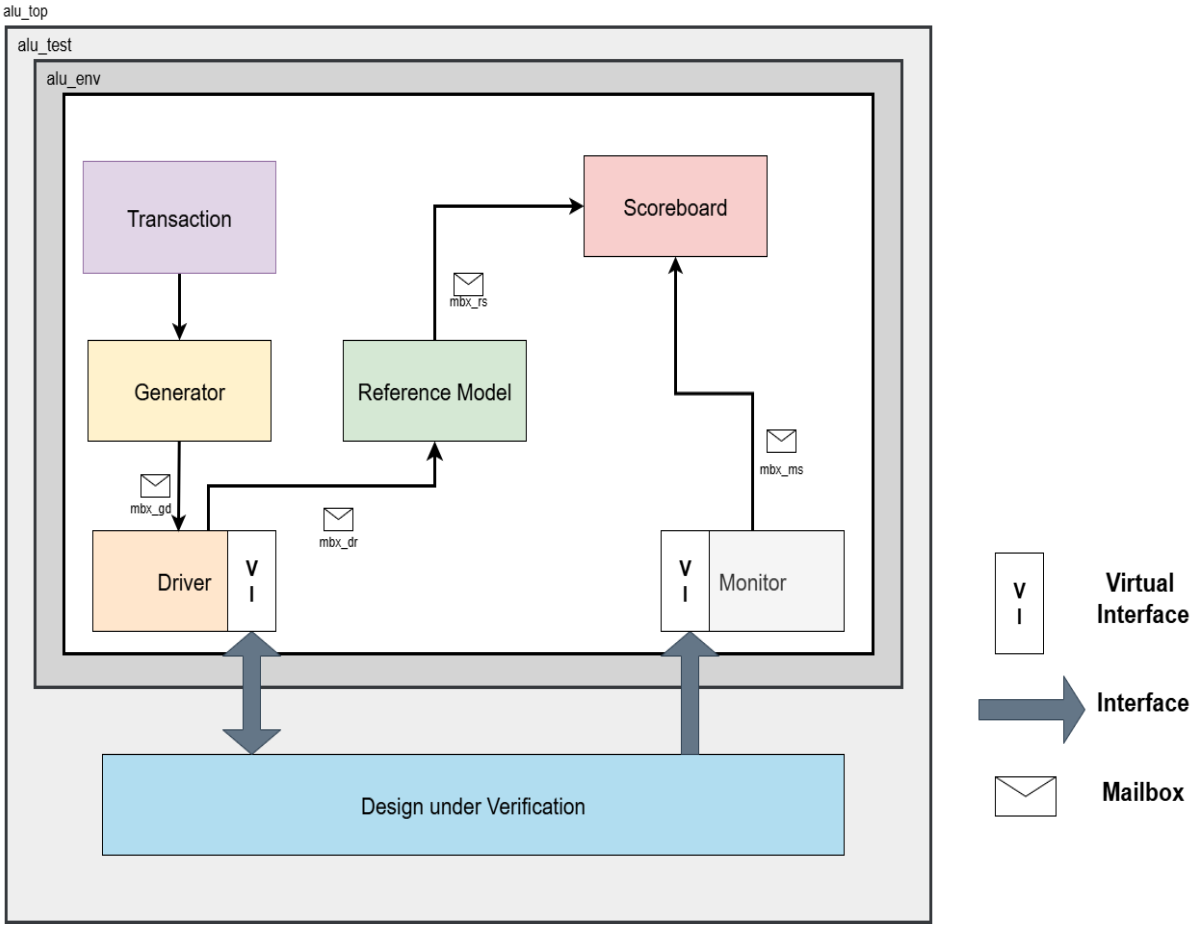


Figure 1: Testbench Architecture

1.Top

The alu_top is the topmost file which connects the DUT and the testbench.

2.TEST

The alu_test is responsible for,

- Configuring the testbench
- Initiate the testbench components construction process
- Initiate the stimulus driving.

3. 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.

4.Transaction

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

5. 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.

6. 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.

7. 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.

8. Reference Model

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

9. 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.

10. 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

11. 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.

12.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.

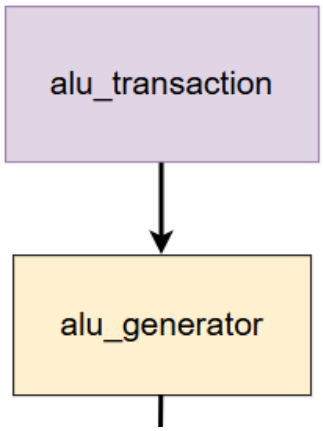
13.Virtual Interface

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.

FLOW CHART OF SV COMPONENTS:

1.Transaction Class

The alu_transaction class consists of input and output stimuli.



Randomized input stimuli: The transaction contains ALU input signals declared with rand keyword.

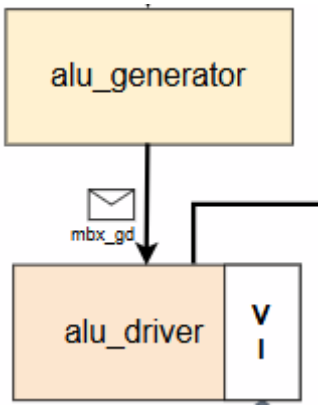
OPA, OPB, MODE, CMD, CIN, INP_VALID are the inputs that will be randomized.

Non randomized Output stimuli:

Output signals are declared without rand keyword.

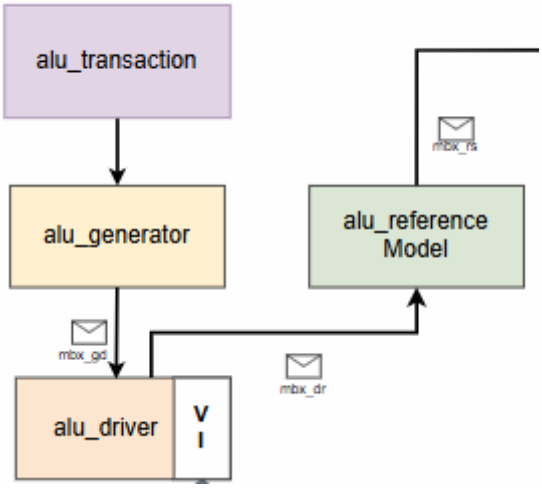
Output signals are: ERR, RES, COUT, OFLOW, G, L, E.

2.Generator class



The generator consists of the ALU transaction class handle, the mailbox handle which connects to the driver, which randomizes transactions and sends the randomized transactions to the driver through the mailbox.

3. Driver Class



Driver class consiste of two mailboxes:

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

Virtual Interfaces:

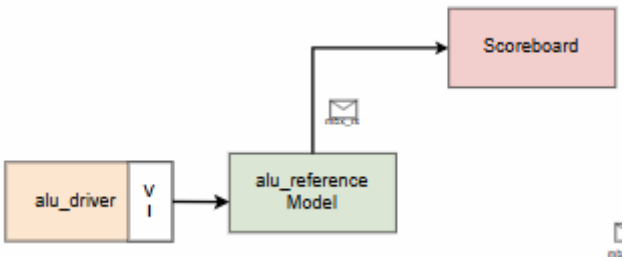
The interfaces facilitate synchronized communication between the testbench and the Design Under Verification (DUV)

Functional Coverage: Captures all combinations of input operands and operations

Drive Task:

Applies the stimulus to the DUV based on the received transaction

4. Reference Model



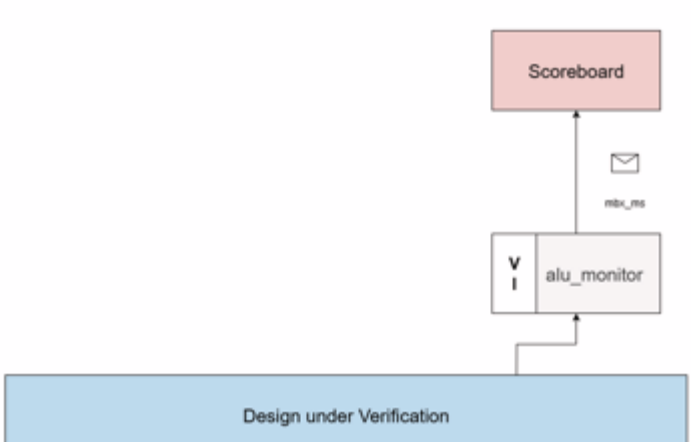
Acts as the golden reference for result validation. It processes the same inputs as the DUV and produces the expected outputs.

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

5. Monitor



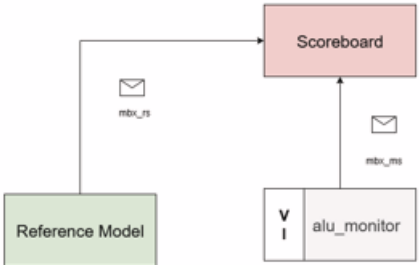
The ALU Monitor observes the DUV's outputs and converts them into transaction objects for validation.

Mailbox Communication:

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

6. Scoreboard



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

Mailbox Communication:

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 coverage plan ensures thorough verification of all key signals and operations. Input validation coverage includes all combinations of the INP_VALID signal (00, 01, 10, 11). The CMD signal is exercised across all 14 supported ALU operations (0 to 13).

Control signals such as CE, CIN, and RESET are monitored to ensure transitions between 0 and 1 are properly handled. The MODE signal is verified to cover both arithmetic and logical modes.

Operand inputs OPA and OPB are required to span their full value ranges from 0 to $(2^n – 1)$, while the result output RES must cover values from 0 to $(2^{n+1} – 1)$. Output flags like ERR, COUT, and OFLOW are validated under conditions that trigger their assertions.

Comparison flags E, G, and L are tested for correctness in scenarios where OPA == OPB, OPA > OPB, and OPA < OPB. Additionally, cross-coverage between CMD and MODE (CMD_X_MODE) is implemented to ensure every operation is tested in both arithmetic and logic modes.

Assertion Plan:

Assertions are implemented to verify signal correctness and timing behavior. These include:

Ensuring the CMD signal remains within its valid range.

Verifying INP_VALID accurately reflects the readiness of inputs.

Checking that CMD, OPA, and OPB arrive together when expected.

Confirming that the RST signal immediately clears all outputs and flags, independent of the clock.

Functional assertions ensure that:

In logical mode, COUT and OFLOW remain low

The comparison flags (G, L, E) are mutually exclusive

An error is triggered if CMD is 12 or 13 and OPB[7:4] is not 0000

The result output (RES) remains stable when CE is low or INP_VALID is 00