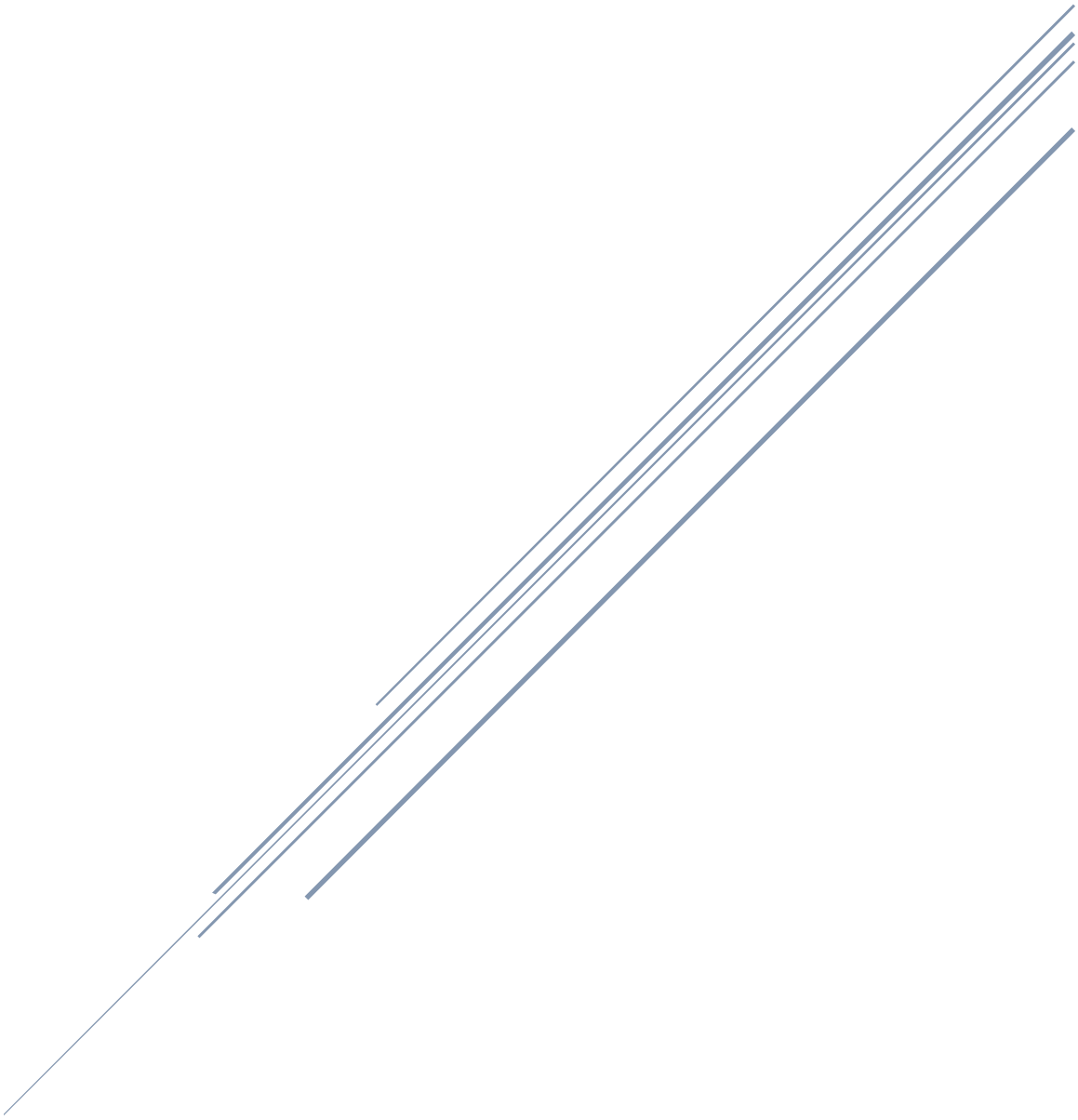
ALU Verification Plan

-SRIJAN S UPPOOR(6113)



VERIFICATION DOCUMENT- ALU

CHAPTERS	CONTENTS	PAGE.NO
CHAPTER 1	**Design Overview**	2-9
1.1	ALU introduction	2
1.2	Advantages of ALU	2-3
1.3	Disadvantages of ALU	3-4
1.4	Use Cases of ALU	4-5
1.5	Project Overview of ALU	5
1.6	Design Features	5-6
1.7	Design Limitation	6-7
1.8	Design diagram with interface signals	7-9
CHAPTER 2	**Verification Architecture**	10-16
2.1	Verification Architecture	10
2.2	Verification Architecture for ALU	11
2.3	Flow chart of SV components	12-16

CHAPTER 1 – DESIGN OVERVIEW

1. ALU

An Arithmetic Logic Unit (ALU) is a digital circuit that performs arithmetic and logic operations. It is the heart of most computational logic and forms the core of processors and microcontrollers. This ALU supports a rich set of operations based on CMD and MODE inputs.

This ALU design stands out due to its flexibility and adaptability. It is parameterized, allowing its bit-width to be tailored to the needs of various applications—such as 8-bit, 16-bit, or higher. Beyond basic operations like addition and subtraction, it supports a broad set of functions, including bitwise rotations, comparisons, as well as error detection and input validation to enhance reliability.

The entire architecture is synchronous, synchronized with a clock signal and reset controls. This design approach guarantees predictable and stable behaviour, making it ideal for integration into modern, high-performance digital systems.

1.2 Advantages of ALU:

*Flexible bit width support

The design is scalable it can start as a 16-bit ALU and expand to 32, 64, or even 128 bits without redesigning the core logic. This saves development time and reduces the chance of introducing new errors.

*Extensive Functionality:

It supports a total of 25 operations (11 arithmetic and 14 logical), enabling everything from basic math to advanced bit manipulations like rotation reducing the need for additional external processing units.

*Intelligent Input Management:

With an INP_VALID signal and built-in timeout mechanism, the ALU gracefully handles input synchronization. If inputs don’t arrive in time, the timeout ensures the system doesn’t freeze or hang

*Status Flags:

Each operation generates status outputs such as carry, overflow, comparison results (greater, less, equal), and error indicators providing the system with rich feedback for better decision-making.

*Built-in Error Detection:

The ALU is capable of catching invalid operation scenarios, particularly during rotate operations where improper input formats can lead to malfunction. This built-in check helps prevent such issues

1.3 Disadvantages of ALU:

•Fixed Timeout Duration:

The input waiting mechanism is hard-coded to wait for exactly 16 clock cycles. This may not be ideal for all systems, particularly those that are too slow for high-speed applications or too fast for slower ones, and it can't be reconfigured.

•Ambiguous Command Structure:

Some command codes serve multiple purposes depending on the selected mode (e.g., arithmetic vs logic). For instance, CMD = 0 could mean ADD or AND, which can easily lead to programming mistakes if not handled carefully

•Generic Error Flag:

The ALU provides a single error signal for all error types, without specifying what went wrong. Whether it was a timeout, an invalid command, or a bad input, the system gets the same flag, making debugging more difficult

•Hardware Overhead:

While feature-rich, the ALU’s complexity means it consumes more logic resources. For applications with simple requirements, this design might be unnecessarily large and inefficient

•Strict Rotate Input Rules:

Rotate operations have tight constraints for example, certain bits in operand B (like bits [7:4]) must be zero. These requirements can be tricky to meet in software and may lead to unintended errors if not handled properly

1.4 Use cases of ALU:

***Arithmetic Operations

Handles essential arithmetic functions such as addition, subtraction, multiplication, and division. These are fundamental for tasks like incrementing counters, calculating memory addresses, and performing general computations in software or hardware.

*Logical Operations
Executes standard logic operations like AND, OR, XOR, and NOT. These are widely used for value comparisons, implementing decision-making structures, and applying bitwise masks in digital systems.

*Bit Manipulation (Shifting & Rotation):

Supports left and right shifts, as well as bit rotations, which are vital in tasks such as data encoding, cryptographic algorithms, and efficient bit-level operations.

*Data Comparison:

Compares two operands to determine equality, greater than, or less than relationships. This functionality supports conditional logic, loop control, and decision branching in both software and hardware.

*Address Calculations

Used in memory operations to calculate next instruction or data address

*Checksum and CRC Computation

Can be used to compute checksums or cyclic redundancy checks (CRC), which are widely applied in error detection for communication protocols and data transmission systems.

*Control Signal Generation

Uses comparison results to generate control signals in FSMs (Finite State Machines) and controllers

*Carry and Overflow Detection:

Monitors for carry-out or arithmetic overflow, ensuring proper handling of signed and unsigned data and maintaining operation accuracy during calculations.

*Executing CPU Instructions

The ALU is the core part of the execution stage of a CPU it carries out actual operations for instructions

1.5 Project Overview of ALU:

This project aims to design a flexible and high-performance Arithmetic Logic Unit (ALU) suitable for a variety of digital systems, ranging from simple embedded devices to complex processors. One of the core strengths of this ALU is its parameterized structure, allowing it to support different bit-widths such as 16, 32, 64, or even 128 bits. This scalability makes it adaptable to both low-resource and high-performance environments. The ALU takes in two operands, OPA and OPB, and operates in two clearly defined modes: arithmetic mode when MODE is set to 1, and logical mode when MODE is 0. Arithmetic mode supports 11 operations and Logical mode supports 14 operations covering everything from basic arithmetic like addition, subtraction, and multiplication to logical operations like AND, OR, XOR, and advanced functions such as bit rotations. The dual-mode system efficiently leverages a parameterized command input, which ensures logical separation and efficient decoding of operations.

To meet the demands of real-world system integration, the ALU includes intelligent control features. In addition, a built-in timeout mechanism ensures the ALU doesn't stall indefinitely, enhancing reliability. The design also includes comprehensive status outputs such as carry detection, overflow flags, and comparison results like greater than, less than, and equal. These flags allow external systems like control units or processors to make decisions based on the outcomes of ALU operations, such as branching or exception handling.

Another highlight of this design is its robust error-handling capability. For operations like bit rotations, where operand B must follow strict rules, the ALU is equipped to detect and flag invalid inputs instead of failing silently. This proactive error detection simplifies debugging and integration, especially in complex systems. Overall, the project focuses on creating a reusable, scalable, and reliable ALU design that balances computational power with smart system-level

1.6 Design Features:

• Synchronous Operation with Asynchronous Reset

Triggers on the rising edge of CLK and can reset immediately via an async reset — ideal for reliable start-up and emergency recovery.

• Flexible Bit-Width via Parameterization

Data width is set by a parameter (16, 32, 64, 128 bits, etc.) so you can scale the ALU to your application without touching the core logic.

• Smart Operand Control

A 2-bit INP_VALID input indicates whether none, one, or both operands are ready perfect for handling staggered inputs in pipelined or asynchronous designs.

• Advanced Rotate Operations

Supports variable rotate left/right based on operand B. Includes error checks to catch bad input values.

• Comprehensive Comparison Outputs

Simultaneously provides Greater (G), Less (L), and Equal (E) flags to speed up control-logic decisions.

• Built-in Overflow Detection

Automatically detects overflow in arithmetic operations to help avoid errors in calculations.

• Power-Saving Clock Enable (CE)

An optional CE input lets you gate the clock and shut down the ALU when idle, saving power in low-energy systems.

1.7 Design Limitation:

*Fixed Timeout
The 16-cycle timeout is hardcoded. If a system needs a shorter or longer wait, the design must be changed.

*Mode-Dependent Command Codes

Same command code does different things in arithmetic and logical modes — this can lead to software mistakes.

*Single Error Signal (ERR)

Only one ERR signal is used for all errors, so you can't tell if it was a timeout, invalid command, or input problem.
*Limited Rotate Range

Rotate operations only support up to 8 positions — may not be enough for larger bit-widths.

* Operand Priority

In case of timeout, the ALU always uses the latest operand — which may not match what some systems expect.

*No Pipelining

It executes one operation at a time with no pipeline support, which can slow down performance in fast systems.

*Wasted Resources

Even unused operations still take up hardware space due to the parameterized design — not ideal for small or resource-limited chips.

1.8 Interface signal design diagram

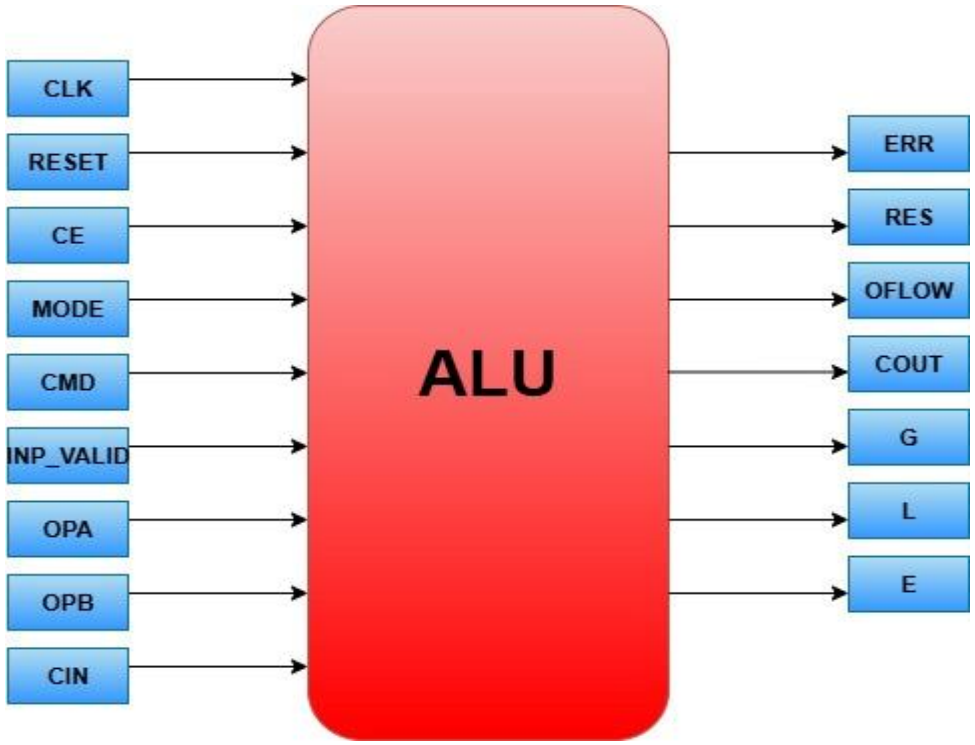


Fig No.1 ALU BLOCK DIAGRAM

Sl. No	Signals	Description
1	OPA [DATA_WIDTH-1:0]	Operand A – First operand used in all operations. Required for both arithmetic and logical operations. For single-operand operations, this is used depending on the CMD.
2	OPB [DATA_WIDTH-1:0]	Operand B – Second operand used in dual-operand instructions. May be ignored in certain single-operand operations.
3	CIN	Carry-in – Only relevant for operations that involve carry propagation (e.g., ADD_CIN or SUB_CIN operations). It is considered in CMD 2 and CMD 3.
4	CMD[CMD_WIDTH:0]	Command Select – A 4-bit field that selects the operation to be performed. Ranges from basic arithmetic (ADD, SUB) to logical (AND, OR, NOT) and shift operations (SHL, SHR, ROL, ROR).
5	MODE	Mode Select – 1-bit field to switch between Arithmetic (1) and Logical (0) operation sets. It determines the interpretation of CMD and which operations are valid.
6	INP_VALID [1:0]	Indicates the validity of inputs provided:
		1: Only OPA is valid (for operations on A)
		2: Only OPB is valid (for operations on B)
		3: Both operands are valid (for dual operand operations)
7	RES [2*DATA_WIDTH-1:0]	Result – Output of the operation based on selected CMD and MODE. Width is extended to accommodate operations like multiplication.
8	COUT	Carry-Out – Used for ADD, ADC and related operations to indicate carry-out from the MSB.
9	OFLOW	Overflow – Indicates overflow in subtraction operations or arithmetic underflow. Set high when a wrap-around is detected.
10	E	Equal – Result of comparison operations. Set when OPA == OPB.
11	G	Greater – Result of comparison operations. Set when OPA > OPB.
12	L	Less – Result of comparison operations. Set when OPA < OPB.

13	ERR	Error – Set when invalid combinations of CMD, MODE, or INP_VALID are detected, or if INP_VALID is not 3 even after 16 cycles (for dual operand commands).

CHAPTER 2 - Verification Architecture

2.1 Verification Architecture :

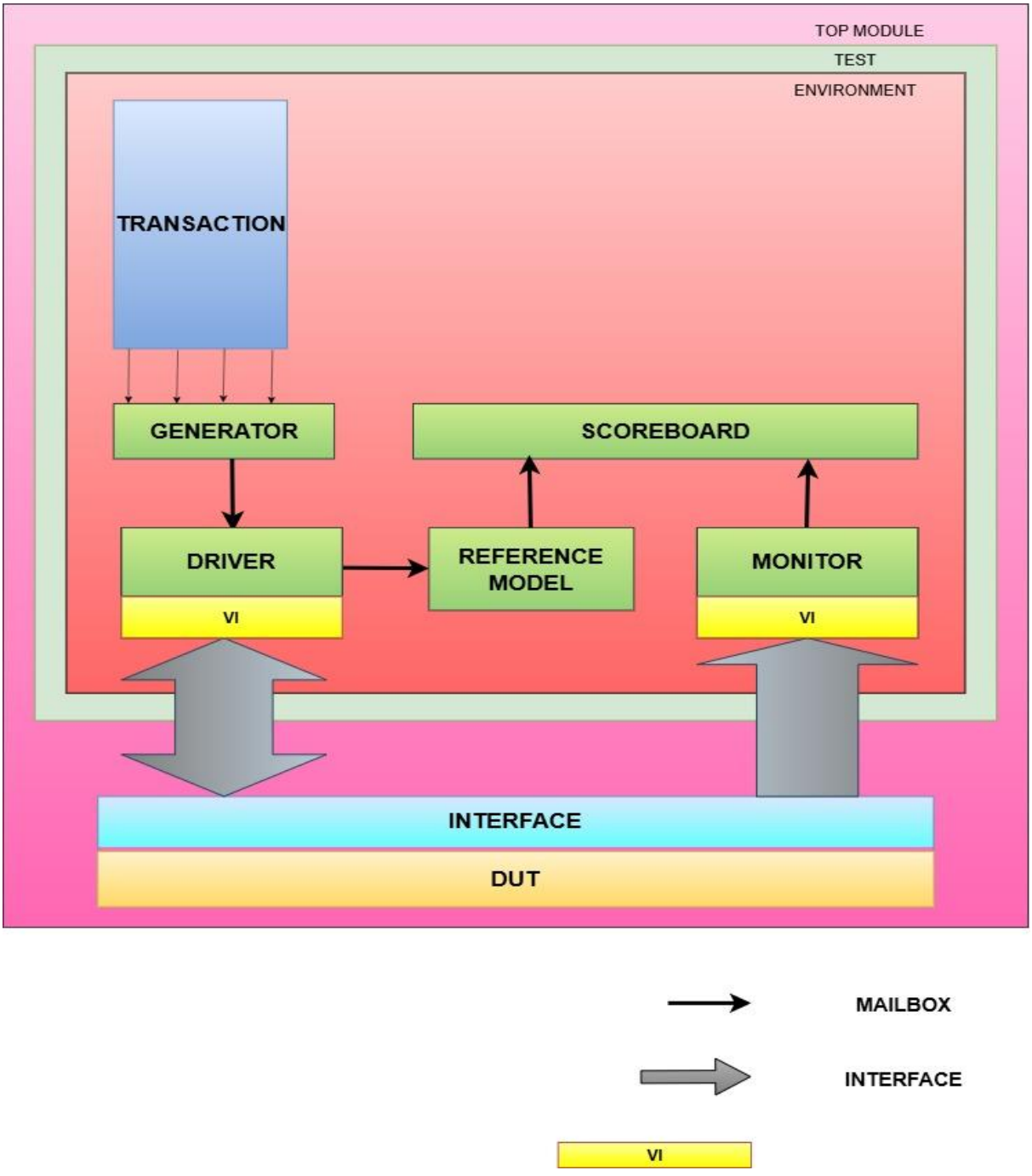


Fig No.2 VERIFICATION ARCHITECTURE

2.2 Verification Architecture for ALU:

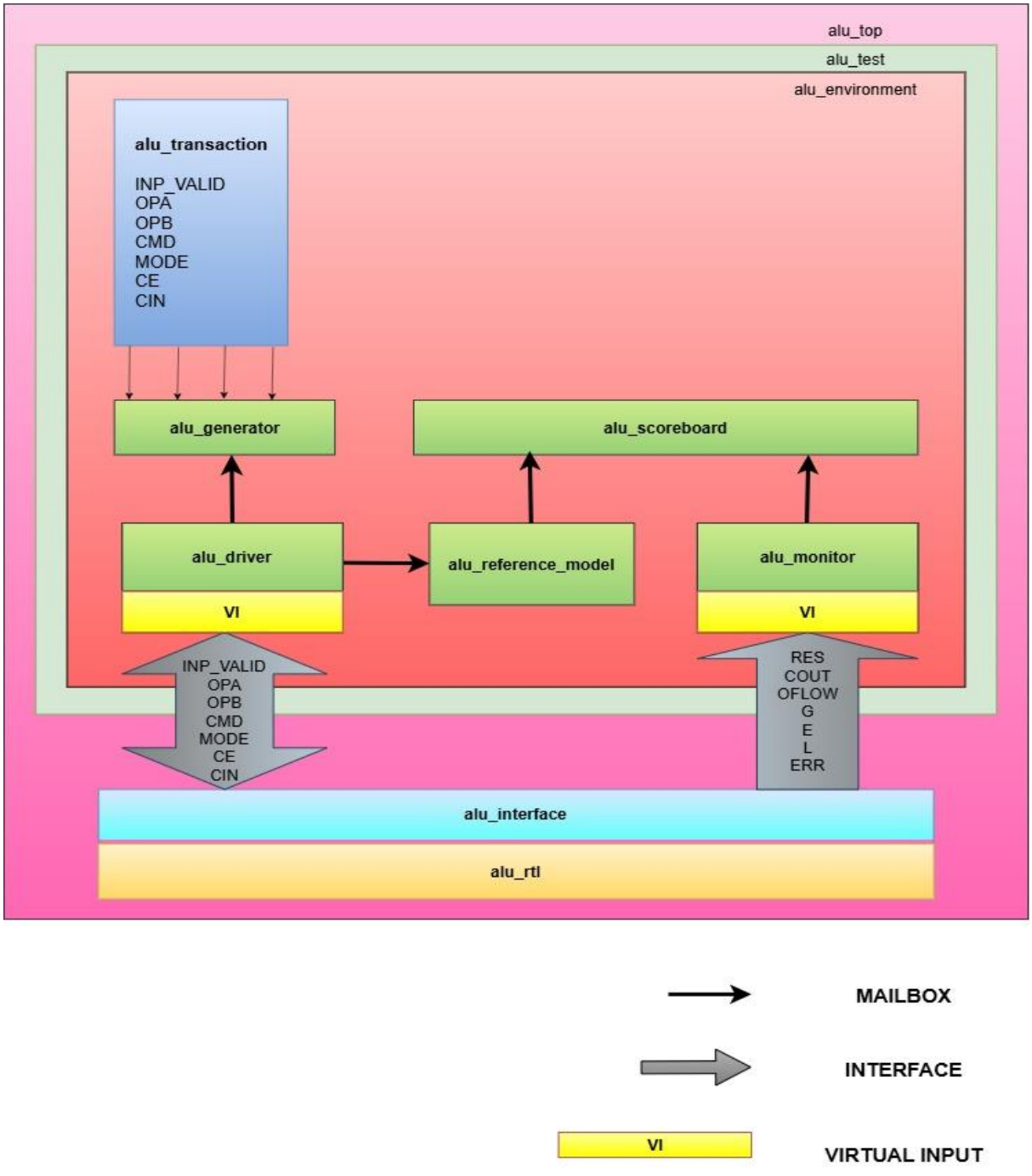


Fig No.3 VERIFICATIN ARCHITECHURE FOR ALU

2.3 FLOW CHART OF SV COMPONENTS :

Generator:

Generator module of the testbench that produces constrained random stimuli (transactions) for the DUT. The generator subsequently transmits the generated stimuli to the driver via a mailbox (mbx_gen2drv).

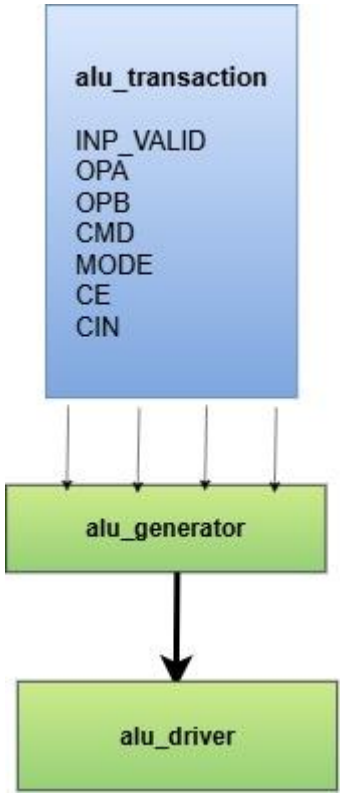


Fig No.3 GENERATOR FLOW DIAGRAM

Driver:

Driver component transforms high-level transactions into pin-level activities at the DUT inputs. It obtains transactions from the generator through a mailbox (mbx_gen2drv) and forwards them to the DUT using a virtual interface. Additionally, it sends the obtained transactions to the reference model via another mailbox (mbx_drv2ref) for predicting results and making comparisons.

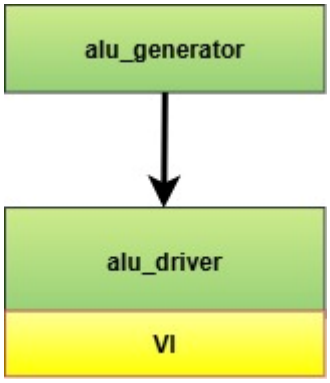


Fig No.4 DRIVER FLOW DIAGRAM

Interface:

The Interface module contains a collection of signals that link the testbench to the DUT at the pin level. It directly corresponds to the input and output connections of the DUT. This interface is utilized by testbench elements like the driver and monitor through virtual interface handles, facilitating organized and reusable connections.

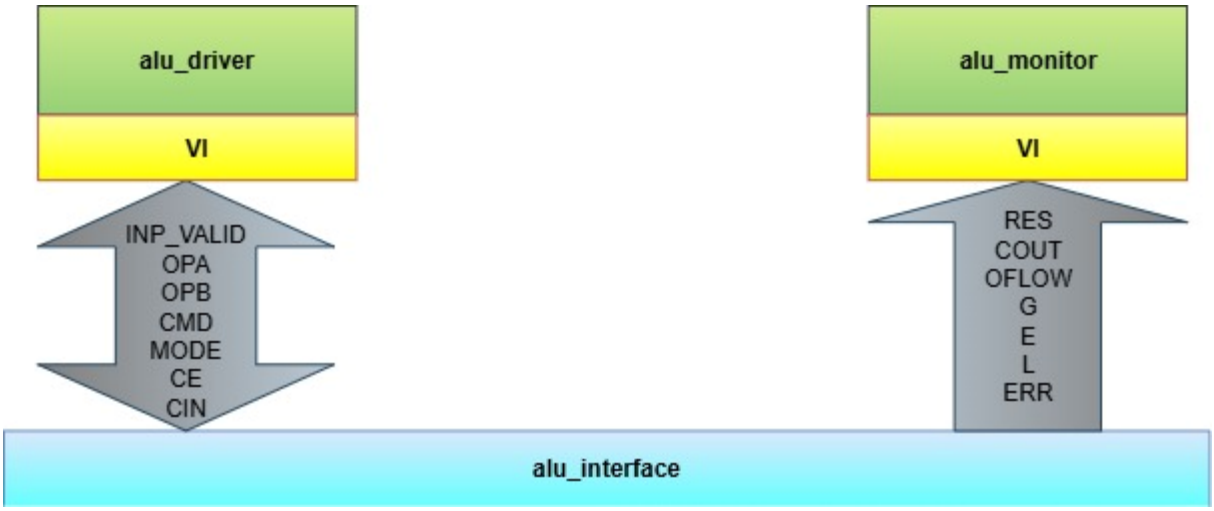


Fig No.5 INTERFACE FLOW DIAGRAM

Reference Model:

For output prediction, validation, and evaluation of the actual output, the Reference Model is a perfect example of implementation and expected output. Usually, it cannot be synthesized. It is employed to assess system performance and verify functionality. Through a mailbox called mbx_drv2ref, the model gets input transactions from the driver. It then processes them in accordance with the desired functionality and delivers the projected outputs to the scoreboard for comparison through a different mailbox called mbx_ref2scr.

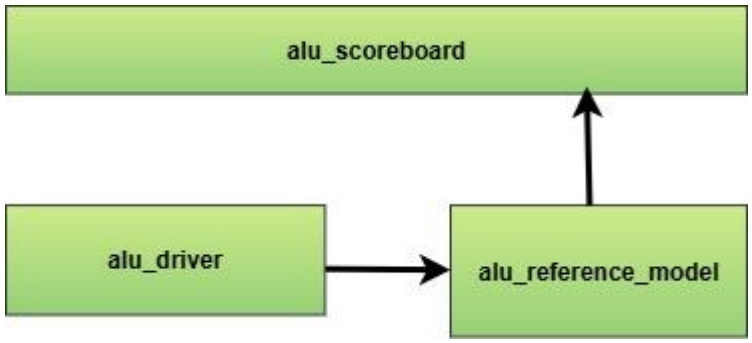


Fig No.6 REFERENCE MODEL FLOW DIAGRAM

Scoreboard:

The scoreboard component obtains the anticipated transactions from the reference model through one mailbox (mbx_ref2scb) and the real transactions from the monitor through another one (mbx_mon2scb). It analyzes these transactions to ensure functional accuracy and produces a report that points out any discrepancies or verifies successful execution.

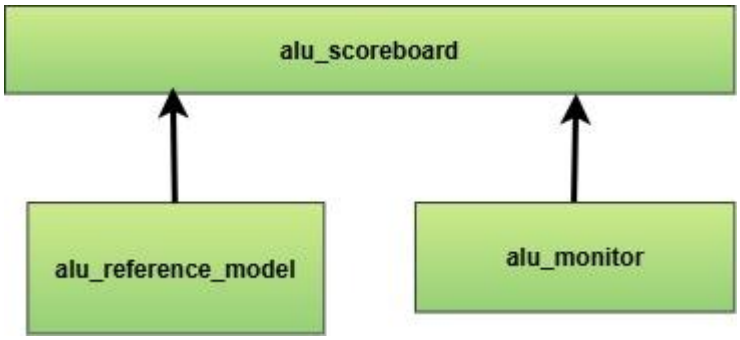


Fig No.7 SCOREBOARD FLOW DIAGRAM

Transactions:

The transaction component is an entity that encapsulates the data exchanged between testbench components, comprising all randomized inputs and non-randomized outputs of the DUT, excluding the clock signal, which is produced independently in the top module. The transaction can impose restrictions to focus on particular test situations.

Fig No.8 TRANSACTION FLOW DIAGRAM

Monitor:

The monitor component transforms pin-level activity from the DUT outputs into elevated-level information. operations. It records the DUT's output signals through a virtual interface and compiles them into transactions that are subsequently delivered to the scoreboard via a mailbox (mbx_mon2scb) for evaluation and examination.

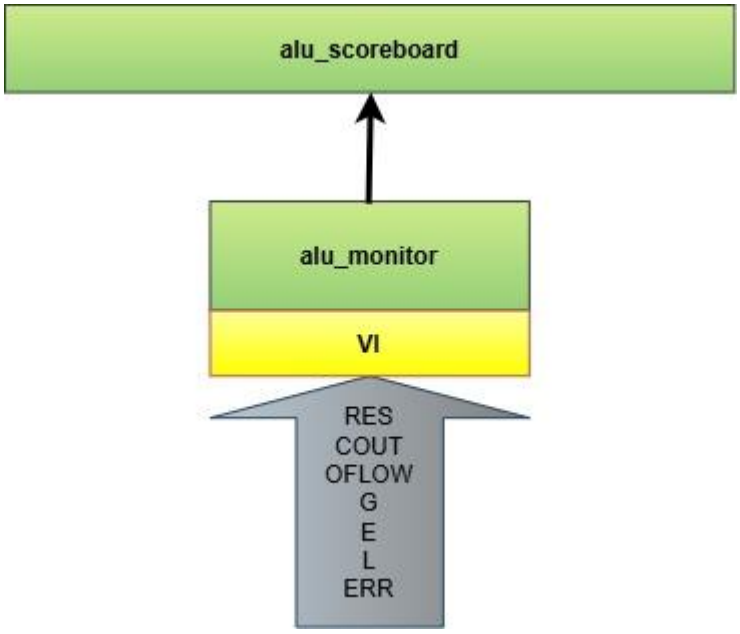


Fig No.9 MONITOR FLOW DIAGRAM

Environment:

The environment component acts as the core of the testbench, tasked with instantiating and linking all key sub-components, such as the generator, driver, monitor, and reference, framework, and ranking system. It constructs and arranges the testbench, guaranteeing effective communication and data transfer between elements for smooth validation.

Test

Test component is responsible for defining and executing various test cases. It instantiates and builds the verification environment, configuring it as needed to apply specific test scenarios and stimuli to the DUT.

Top

Top module is the component that instantiates all components (DUT, interface and test) and is responsible for the clock generation.