ALU Verification Plan

VERIFICATION DOCUMENT- ALU

CONTENTS **Page Number**

CHAPTER 1 – DESIGN OVERVIEW

1.1 ALU introduction:

The Arithmetic Logic Unit (ALU) serves as the central processing component in any digital architecture. Imagine it as an advanced computational engine capable of executing both logical and arithmetic tasks. This implementation is notable for its adaptability it is designed as a parameterized module, allowing us to specify the bit-width to match diverse application requirements.

What sets this ALU apart is its all-encompassing treatment of digital computation. Beyond standard operations like addition and subtraction, it supports advanced features such as bitwise rotations evaluations and incorporates mechanisms for validating inputs and detecting errors. With synchronous operation and reliable clock and reset integration, the design aligns perfectly with the needs of contemporary digital systems.

1.2 Advantages of ALU:

- **Scalable Design:**
  You can easily scale to 8, 16, 32, 64-bit, or any-bit without changing the design. Saves time and avoids new bugs.

- **Wide Range of Operations:**
  Supports 14 arithmetic and 14 logic operations — from simple add to complex rotate. So, fewer external components are needed.

- **Smart Input Handling:**
  It uses INP_VALID to handle inputs arriving at different times. A timeout avoids system hangs if inputs are missing.

- **Status Flags:**
  Each operation gives useful flags like carry, overflow, greater, less, equal, and error — helping the system make smart choices.

- **Power Saving:**
  Clock enable (CE) lets you turn off the ALU when not needed — great for saving power in battery devices.

- **Error Detection:**
  Catches invalid operations, especially in rotate cases where bad input patterns can cause problems.

1.3 Disadvantages of ALU:

- **Timeout Isn't Flexible:**
  The ALU waits exactly 16 clock cycles for inputs — no more, no less. That might be too long for fast systems or too short for slower ones, but you can't adjust it.

- **Commands Can Be Confusing**:
  The same command number does different things depending on the mode. For example, CMD = 0 means ADD in one mode and AND in another. Easy to mix up if you're not careful.

- **One Error Signal for Everything**:
  If something goes wrong, you just get a generic error flag. It doesn't tell you what the problem is — was it a timeout, a wrong command, or a bad input? Makes troubleshooting harder.

- **Too Big for Simple Jobs:**
  The ALU supports a lot of features, which is great, but it also uses more hardware. If your system only needs basic stuff, this design might be overkill.

- **Rotate Operation Is Tricky:**
  Rotate left/right needs operand B to follow strict rules — like making sure bits [7:4] are zero. It's easy to mess this up in software, leading to errors.

1.4 Use cases of ALU:

- **Arithmetic Operations**
  ▪ Performs basic math like addition, subtraction, multiplication, and division
  ▪ Essential for tasks like updating counters, calculating addresses, or doing math in a program

- **Logical Operations**
  - Executes AND, OR, XOR, NOT operations
  - Used in comparing values, bit masking, and decision-making logic in CPUs and digital circuits

- **Bit Shifting and Rotation**
  - Shifts bits left or right, rotates bits for specific applications
  - Common in encryption, encoding/decoding, and data manipulation

- **Comparisons**
  - Compares two numbers to check greater than, less than, or equal
  - Useful for if-else, loops, and conditional branching in software and hardware

- **Address Calculations**
  - Used in memory operations to calculate next instruction or data address

- **Checksum and CRC Computation**
  - Helps compute checksums and cyclic redundancy checks for error detection in communication systems

- **Control Signal Generation**
  - Uses comparison results to generate control signals in FSMs (Finite State Machines) and controllers

- **Overflow and Carry Detection**
  - Detects arithmetic overflow or carry during operations
  - Important for signed/unsigned number handling and ensuring accuracy

- **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 focuses on building a powerful, yet flexible ALU (Arithmetic Logic Unit) that can be used in a wide range of digital systems, from processors to embedded controllers. The key idea is parameterization — allowing the ALU to work with different bit-widths such as 16, 32, 64, and 128 bits, making it suitable

for both low-end and high-performance applications. At its core, the ALU processes two operands (OPA and OPB) and operates in two distinct modes: arithmetic mode (MODE=1) and logical mode (MODE=0). Each mode supports 14 operations, ranging from basic addition, subtraction, and multiplication to advanced bitwise logic and rotate instructions. This dual-mode structure efficiently utilizes a 4-bit command system while keeping arithmetic and logic functions clearly separated.
To support real-world system requirements, the design includes advanced control features. The INP_VALID signal helps manage inputs that may not arrive at the same time, a common issue in pipelined or asynchronous environments. A built-in timeout mechanism ensures that the ALU doesn't wait forever for missing inputs, improving system reliability. The ALU also offers comprehensive status reporting, including overflow detection, carry flags, and comparison results (greater than, less than, equal), enabling smarter decisions by system software. For example, a control unit or CPU can use these flags for branching or error handling.
Error detection is another strong point of this design. It has dedicated error management for invalid operations—especially for complex ones like variable bit rotations, where certain patterns in operand B can cause undefined behaviour. Instead of failing silently, the ALU flags these issues, making debugging and system integration much easier. Overall, the project aims to create a highly reusable, scalable, and robust ALU that balances performance, flexibility, and reliability — essential for modern digital design.

1.6 Design Features:

- **Synchronous with Asynchronous Reset**
  Works on the rising edge of the clock and can reset instantly using an async reset — useful for reliable startup and emergency resets.

- **Flexible Bit-Width**
  Uses parameters to set data width (16, 32, 64, 128 bits, etc.), so it can be easily adapted without changing the core code.

- **Smart Operand Control**
  A 2-bit INP_VALID signal shows if none, one, or both inputs are ready — great for handling inputs that come at different times.

- **Advanced Rotate Operations**
  Supports variable rotate left/right based on operand B. Includes error checks to catch bad input values.

- **Rich Comparison Output**
  Gives all comparison results — greater (G), less (L), and equal (E) — at the same time for faster decisions in control logic.
- **Built-in Overflow Detection**
  Automatically detects overflow in arithmetic operations to help avoid errors in calculations.

- **Power Saving Option**
  Has a CE (clock enable) input so the ALU can be turned off when not needed — helpful for low-power 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-Based Command Confusion**
  Same command code does different things in arithmetic and logical modes — this can lead to software mistakes.

- **One Error Bit for All Issues**
  Only one ERR signal is used for all errors, so you can't tell if it was a timeout, invalid command, or input problem.

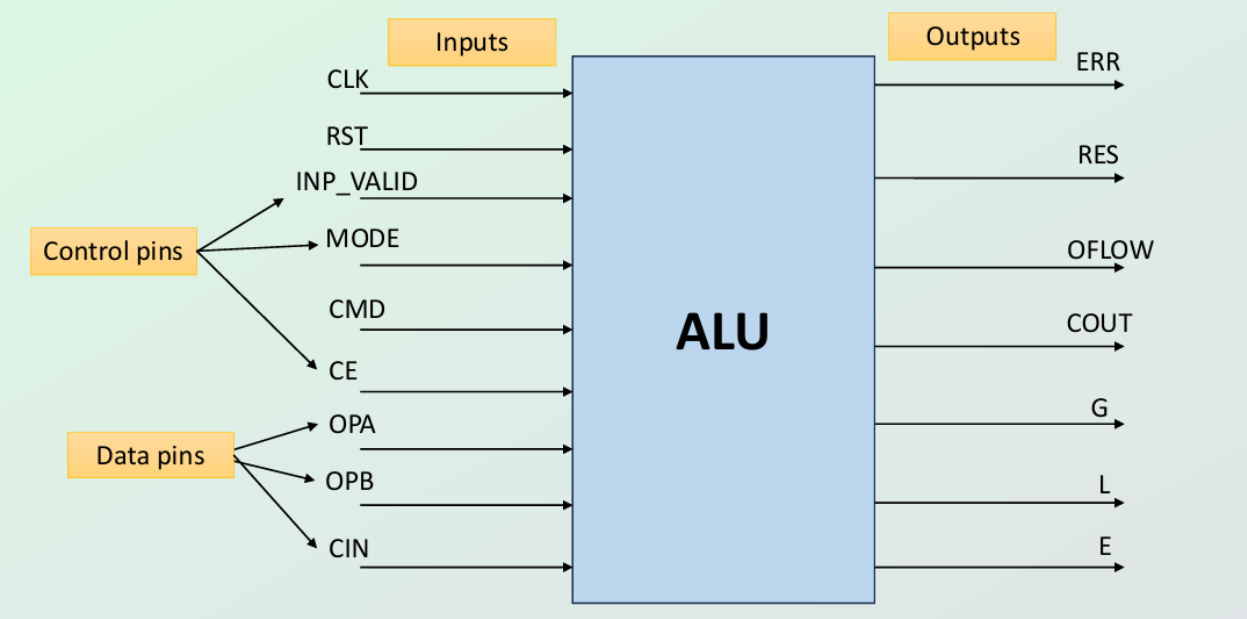
- **Limited Rotate Range**
  Rotate operations only support up to 8 positions — may not be enough for larger bit-widths.

- **Fixed 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 Design diagram with interface signals:



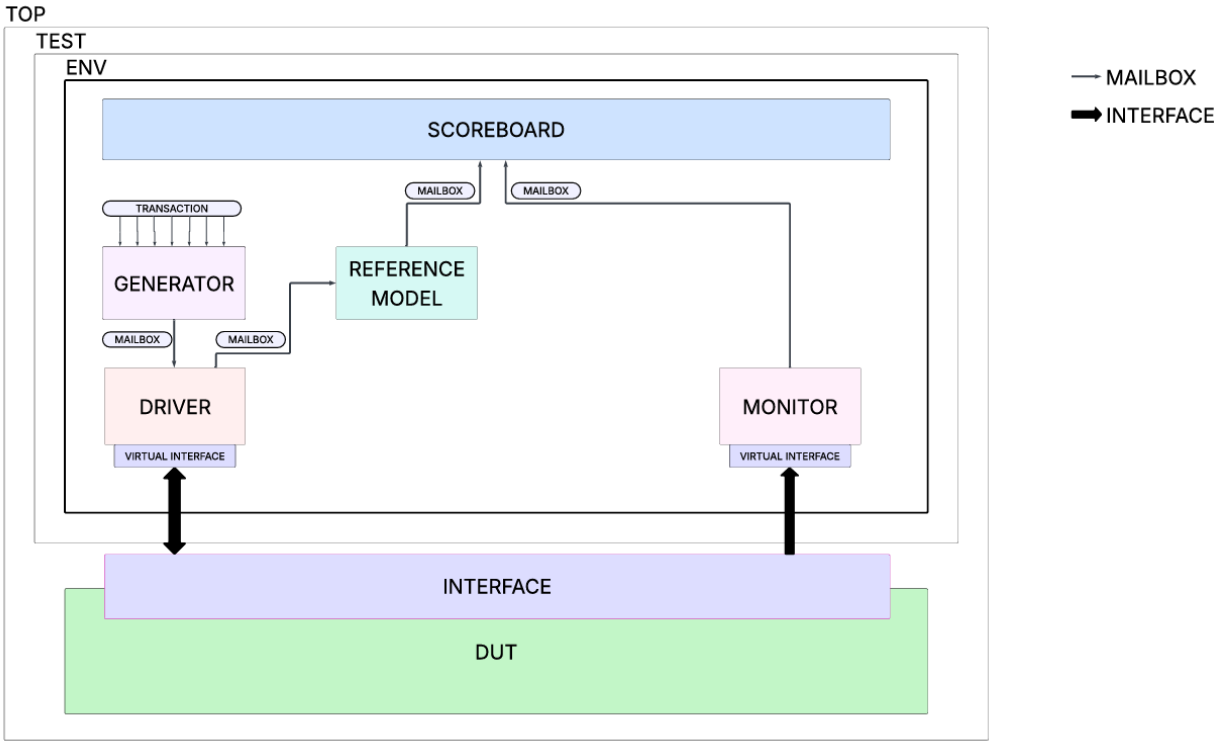
Signal Name	Type	Description
INP_VALID	Input	Input valid signal - indicates when input data is valid
MODE	Input	Mode selection signal - determines ALU operation mode
CMD	Input	Command signal - specifies the specific ALU operation
OPA	Input	Operand A - first arithmetic/logic operand
OPB	Input	Operand B - second arithmetic/logic operand
CIN	Input	Carry In - input carry for arithmetic operations

Output ports:

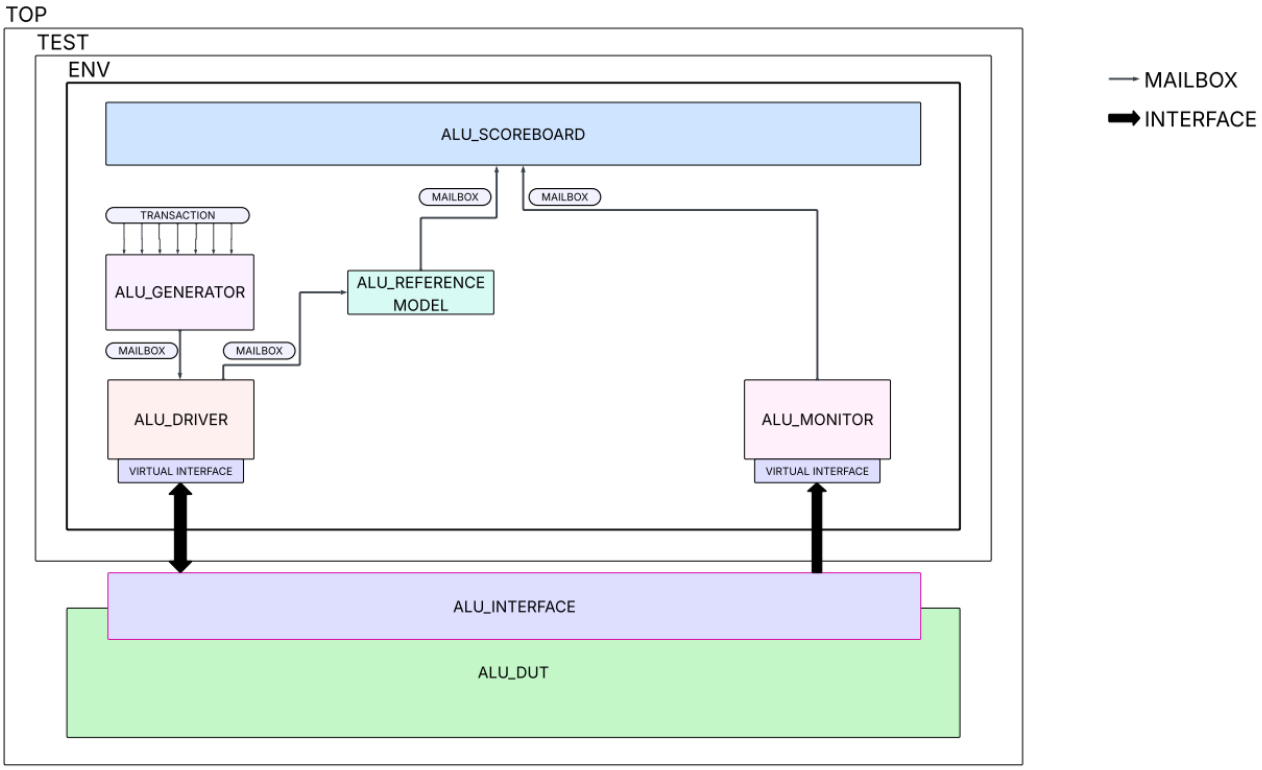
Signal Name	Type	Description
ERR	Output	Error signal - indicates if an error occurred during operation
RES	Output	Result - the output result of the ALU operation
OFLOW	Output	Overflow - indicates arithmetic overflow condition
COUT	Output	Carry Out - output carry from arithmetic operations
G	Output	Greater than - comparison result flag
L	Output	Less than - comparison result flag
E	Output	Equal - comparison result flag

CHAPTER 2 - Verification Architecture

2.1 Verification Architecture :



2.2 Verification Architecture for ALU:

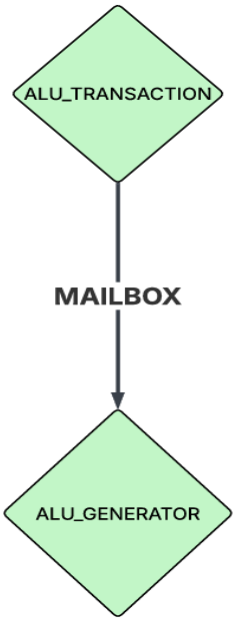


The figure shows general testbench architecture used for verifying digital designs. At the top module is the Design Under Verification (DUV). The test has the testbench environment that includes a transaction generator that creates and manages the input testcases for testing. These transactions are randomized within the generator and then transmitted to the driver, which applies them as signals to the DUV through a virtual interface. Outputs from the DUV are monitored by a monitor, which forwards this data to a scoreboard for checking correctness against expected results calculated by a reference model. The environment block encapsulates all these components, ensuring coordination among them, while the scoreboard oversees the entire verification process to validate the DUV.

2.3 Flowchart of SV Component:

1. Transaction Class:

The alu_transaction class encapsulates all ALU input stimuli and output responses for verification purposes



.

The diagram shows two key components: ALU_TRANSACTION and ALU_GENERATOR, connected through a MAILBOX. The ALU_TRANSACTION component likely handles or processes arithmetic and logic unit transactions, acting as an interface or controller in managing data or instructions. The MAILBOX serves as a communication channel or buffer enabling data flow between these components, ensuring synchronization and orderly data transmission. The ALU_GENERATOR probably generates the arithmetic and logic signals or operations required by the system. Together, these components coordinate to facilitate effective ALU operations within a digital design or processor environment, ensuring proper data handling and execution of arithmetic/logic tasks.

Components Randomized Input Stimuli: The transaction contains ALU input signals declared with the rand keyword for automatic randomization:

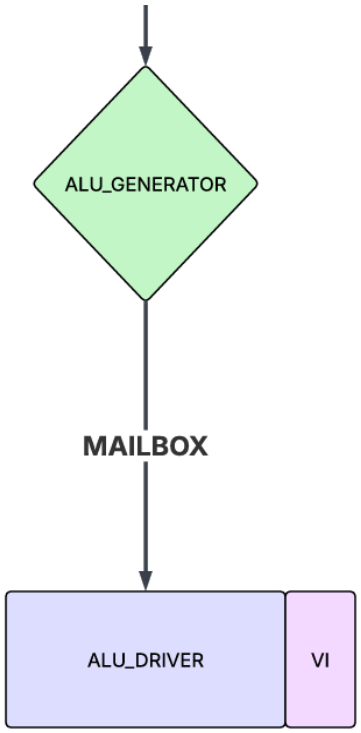
• INP_VALID, MODE, CMD, OPA, OPB, CIN:

Input signals that will be randomized by the generator

Non-Randomized Output Signals Output signals are declared without rand as they represent the ALU's response:

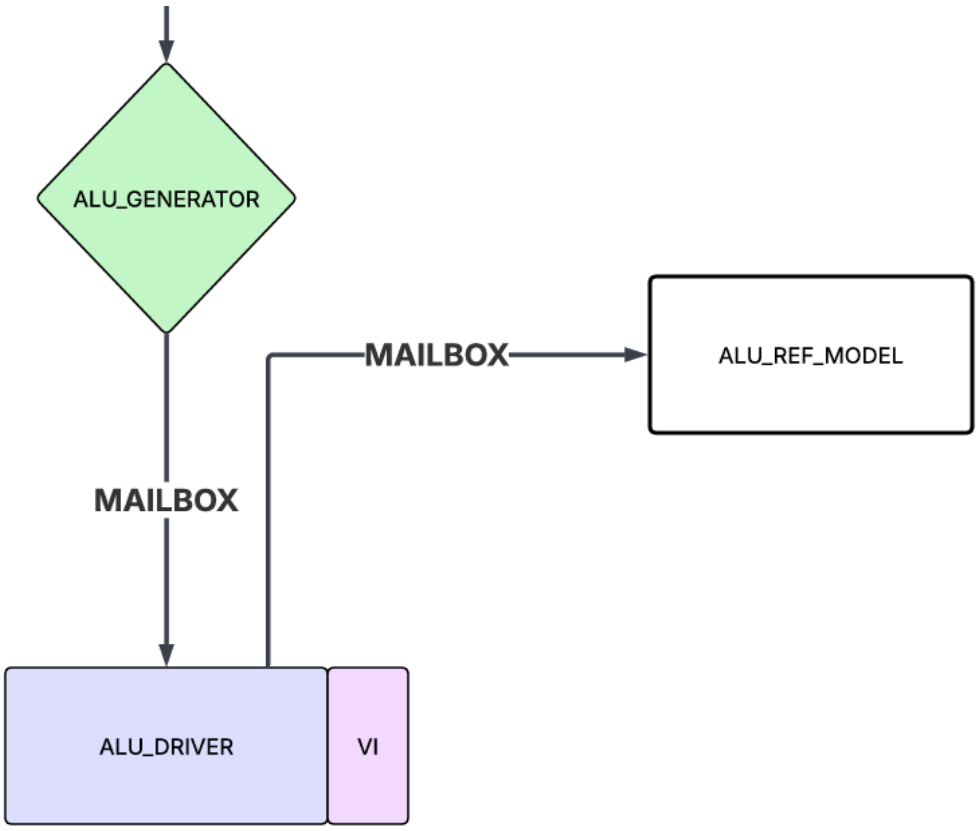
• ERR, RES, OFLOW, COUT, G, L, E: Output signals monitored for verification

2. Generator Class:



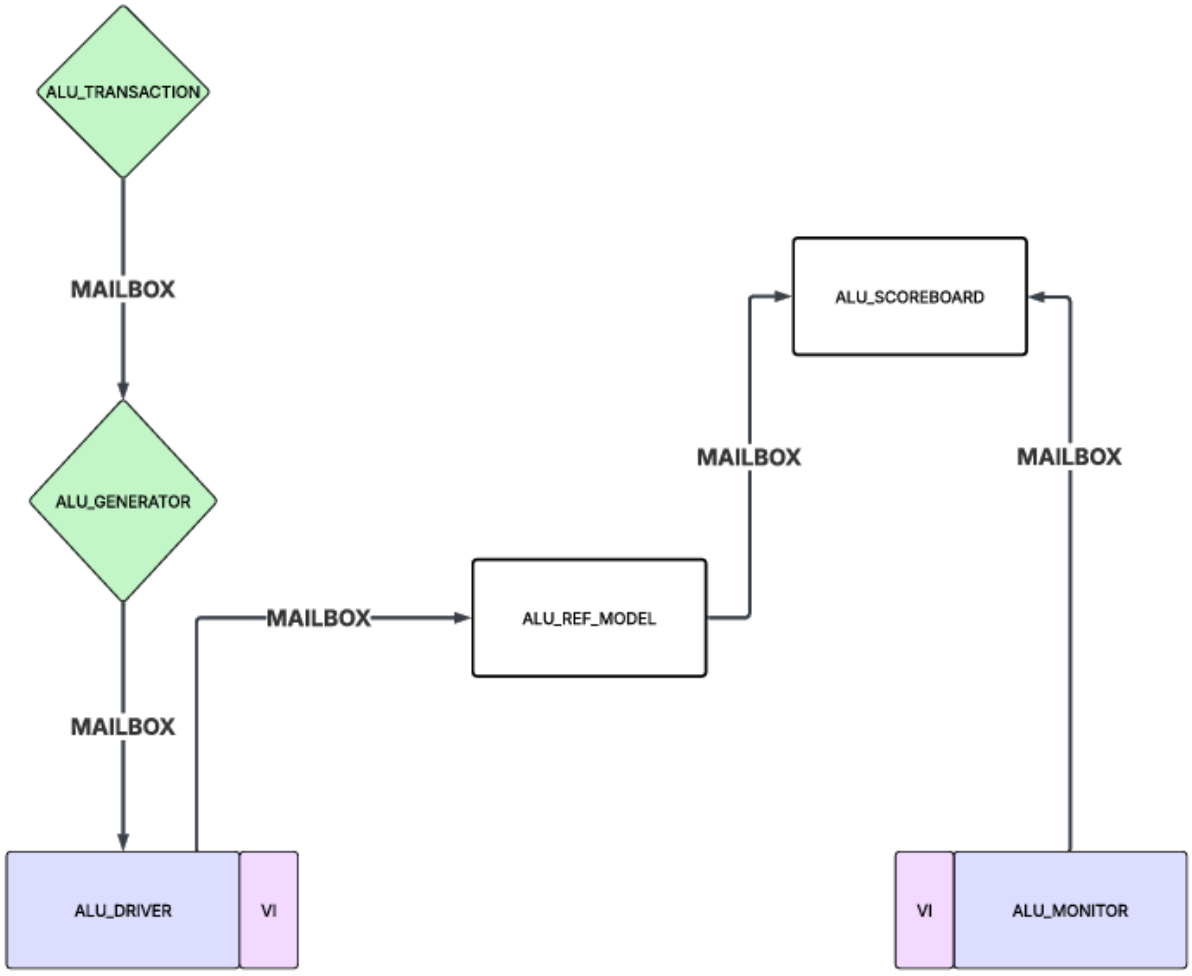
The generator consists of the ALU transaction class handle, the mailbox handle which connects to the driver, and the start() task which randomizes transactions and sends the randomized transactions to the driver through the mailbox.

3. Driver Class:



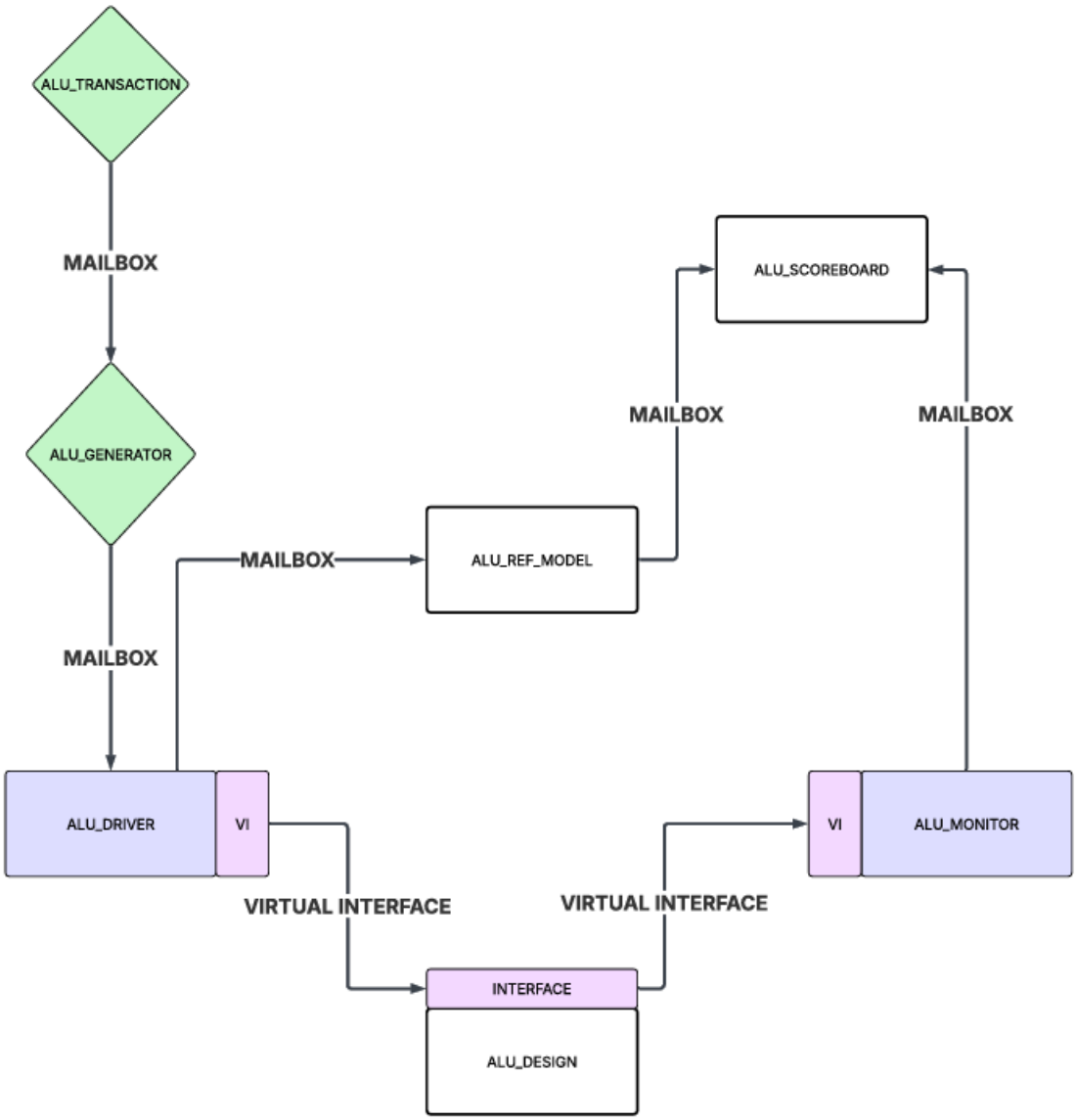
The alu_transaction defines input/output data. The alu_generator creates transactions, sending them via a mailbox to the alu_driver, which drives inputs to the DUT via a virtual interface (VI). Simultaneously, transactions are sent to the alu_reference_model through another mailbox for comparison. This structure enables functional verification by checking DUT outputs against reference results.

4. Reference Class:



The ALU reference model acts like a trusted Miror of the DUT always gives the right answer. It takes the same inputs as the actual ALU and processes them separately to produce the expected result. These results are sent to the scoreboard, where they're compared with what the real ALU produced during testing. By doing this, the reference model helps us spot mistakes or mismatches, making sure the ALU works correctly and behaves as expected under all test conditions.

5. Monitor Class:



Monitor connects to the DUT through a **virtual interface** and continuously samples the input and output signals during simulation. Without influencing the DUT behavior, it captures the real transaction data and sends it to the **scoreboard** using a mailbox. Its job is to translate pin-level activity into meaningful transactions for checking.

6. Scoreboard Class:

The scoreboard serves as the comparator or validator in the testbench. It receives:

Expected results from the reference model

Actual results from the monitor

It then compares both sets of data to check if the DUT output matches the expected behavior. If mismatches are found, it logs errors, helping identify bugs in the design. The scoreboard ensures the DUT is functionally correct under all test scenarios.

CHAPTER 3 : RESULTS AND ANALYSIS

S.No	Test Case / Bug Category	Reason for Failure / Description
1	16-Clock Cycle Timeout Logic	Error flag is not set even after waiting for 16 clock cycles when expected timeout condition occurs.
2	Single-Operand Operations	DUT does not perform increment/decrement unless both INP_VALID signals are high, violating spec.
3	Carry-Out Logic for Increment Operations	COUT logic incorrect/missing for increment operations; flag not asserted when operand overflows (e.g., 255→256).
4	Increment A Operation	INC_A operation does not increment: value remains same as OPA, causing no update.
5	Increment B and Decrement B Operations	INC_B operation decrements instead of incrementing; DEC_B operation increments instead of decrementing.
6	Shift Operations	Right shift on OPA/OPB fails; left shift OPB operation fails.
7	Decrement B Operations	DEC_B operation increments instead of decrementing.
8	Overflow Logic for Decrement Operations	OVER_FLOW logic missing for decrement. OVER_FLOW is not asserted on underflow (e.g., 0 → -1).
9	Carry-In Signal Timing	CIN signal delayed by one clock cycle; causes misalignment in operations using CIN.
10	Rotate Right A by B Error Condition	ROR_A_B error detection remains zero even when condition occurs; fails to set error flags.
11	Multiplication Operation	Issues present in multiplication operation functionality.
12	Logical OR	Logical OR operation fails, even with valid inputs and timing.

3.2 Coverage Report:

• Overall coverage

Coverage Summary by Type:

Total Coverage:					81.75%	**73.16%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Covergroups	314	305	9	1	97.13%	**80.50%**
Statements	474	388	86	1	81.85%	**81.85%**
Branches	161	136	25	1	84.47%	**84.47%**
FEC Expressions	10	5	5	1	50.00%	**50.00%**
FEC Conditions	36	16	20	1	44.44%	**44.44%**
Toggles	340	241	99	1	70.88%	**70.88%**
Assertions	2	2	0	1	100.00%	**100.00%**

• Code coverage:

Covergroups Coverage Summary:

Search:

Covergroups/Instances	Total Bins	Hits	Misses	Hits %	Goal %	Coverage %
/alu_pkg/alu_driver/cg_drv	46	41	5	89.13%	**89.58%**	**89.58%**
/alu_pkg/alu_monitor/cg_monitor	268	264	4	98.50%	**71.42%**	**71.42%**
work.alu_pkg::alu_driver/cg_drv	46	41	5	89.13%	**89.58%**	**89.58%**
work.alu_pkg::alu_monitor/cg_monitor	268	264	4	98.50%	**71.42%**	**71.42%**

• Functional Coverage:

Local Instance Coverage Details:

Total Coverage:					79.22%	**74.01%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	117	93	24	1	79.48%	**79.48%**
Branches	73	64	9	1	87.67%	**87.67%**
FEC Conditions	20	10	10	1	50.00%	**50.00%**
Toggles	204	161	43	1	78.92%	**78.92%**

• Assertion Coverage:

Assertions Coverage Summary:

Search:

Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
/top/intf/assert__clk_must_toggle	0	16047	16048	0	0	1	3	Covered
/top/intf/assert__clock_toggle	0	8023	16048	8025	0	0	1	Covered
/work.alu_if/assert__clk_must_toggle	0	16047	16048	0	0	1	3	Covered
/work.alu_if/assert__clock_toggle	0	8023	16048	8025	0	0	1	Covered

3.3 Output waveform: