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.

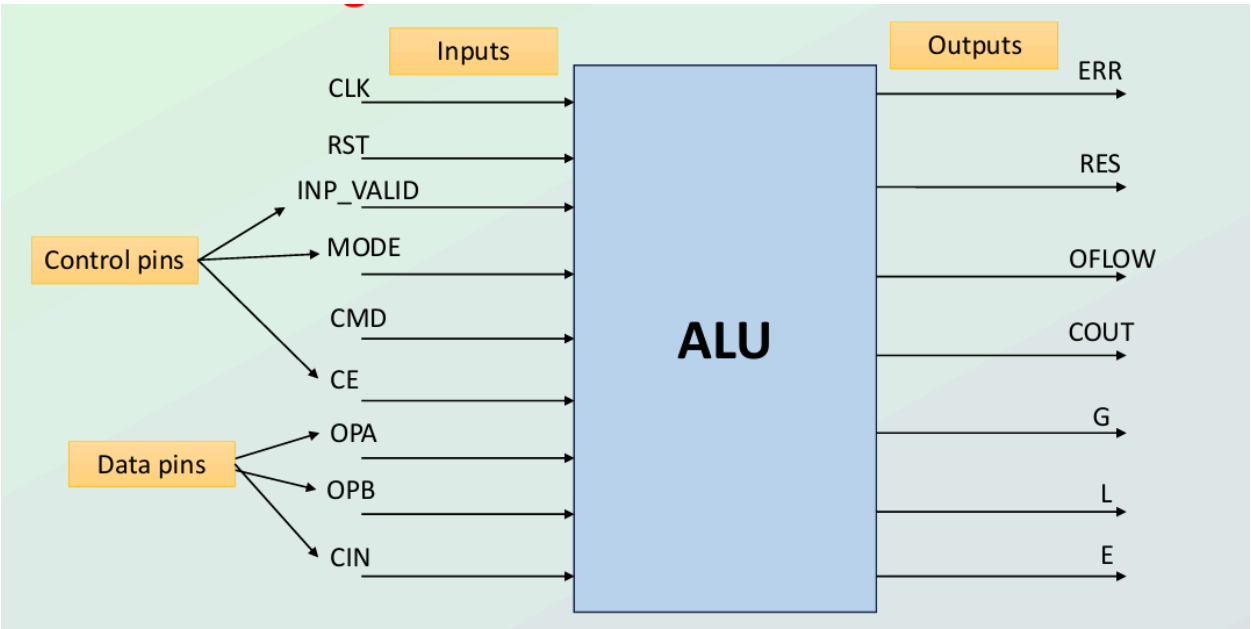
- **Limited Rotate Range**
  Rotate operations only support up to 8 positions — may not be enough for larger bit-widths.

- **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:



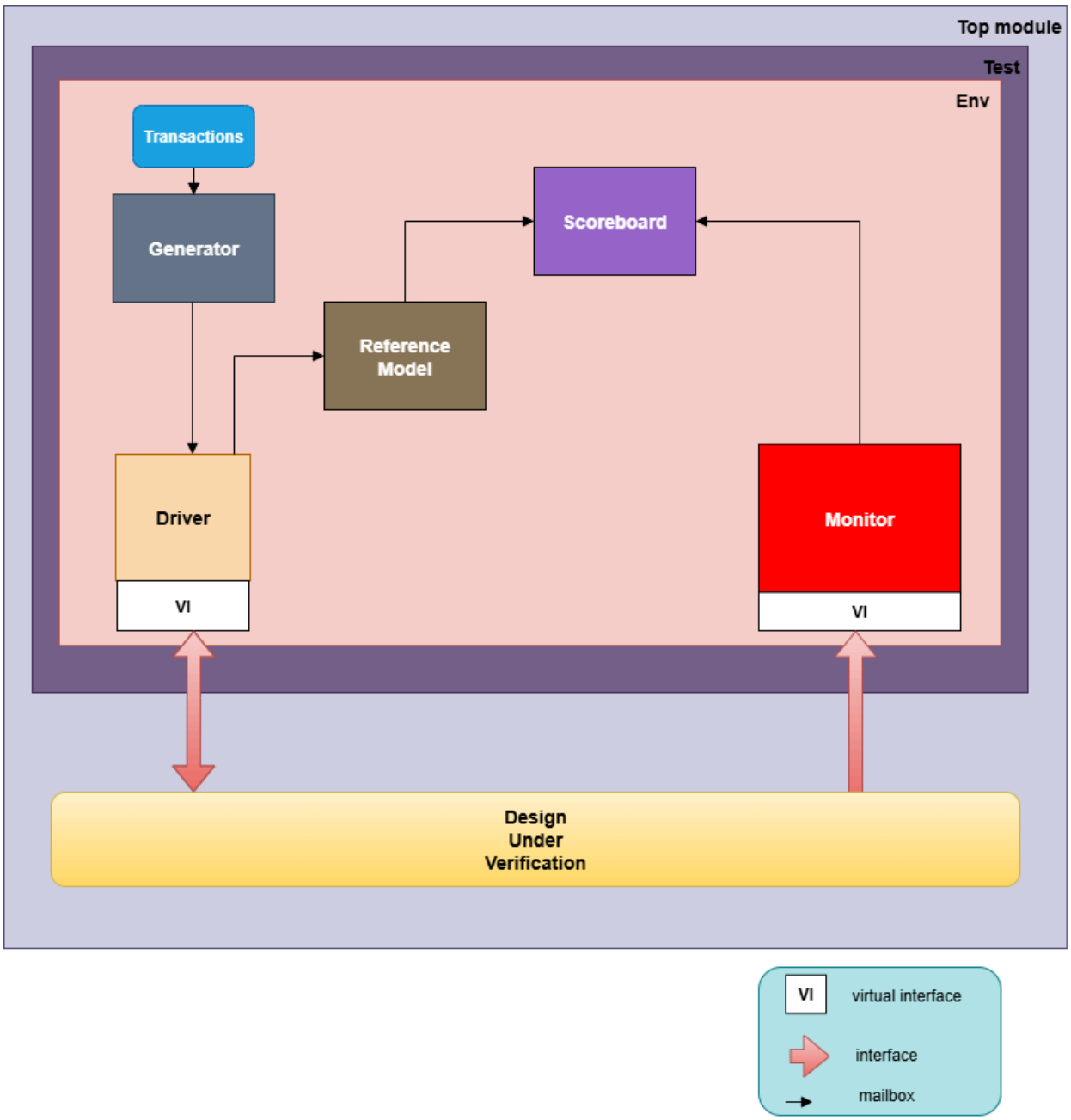
Signal Name	Type	Description
INP_VALID	Input	Input valid signal - indicates when input data is valid
MODE	Input	Mode selection signal - determines ALU operation mode
CMD	Input	Command signal - specifies the specific ALU operation
OPA	Input	Operand A - first arithmetic/logic operand
OPB	Input	Operand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

Output ports:

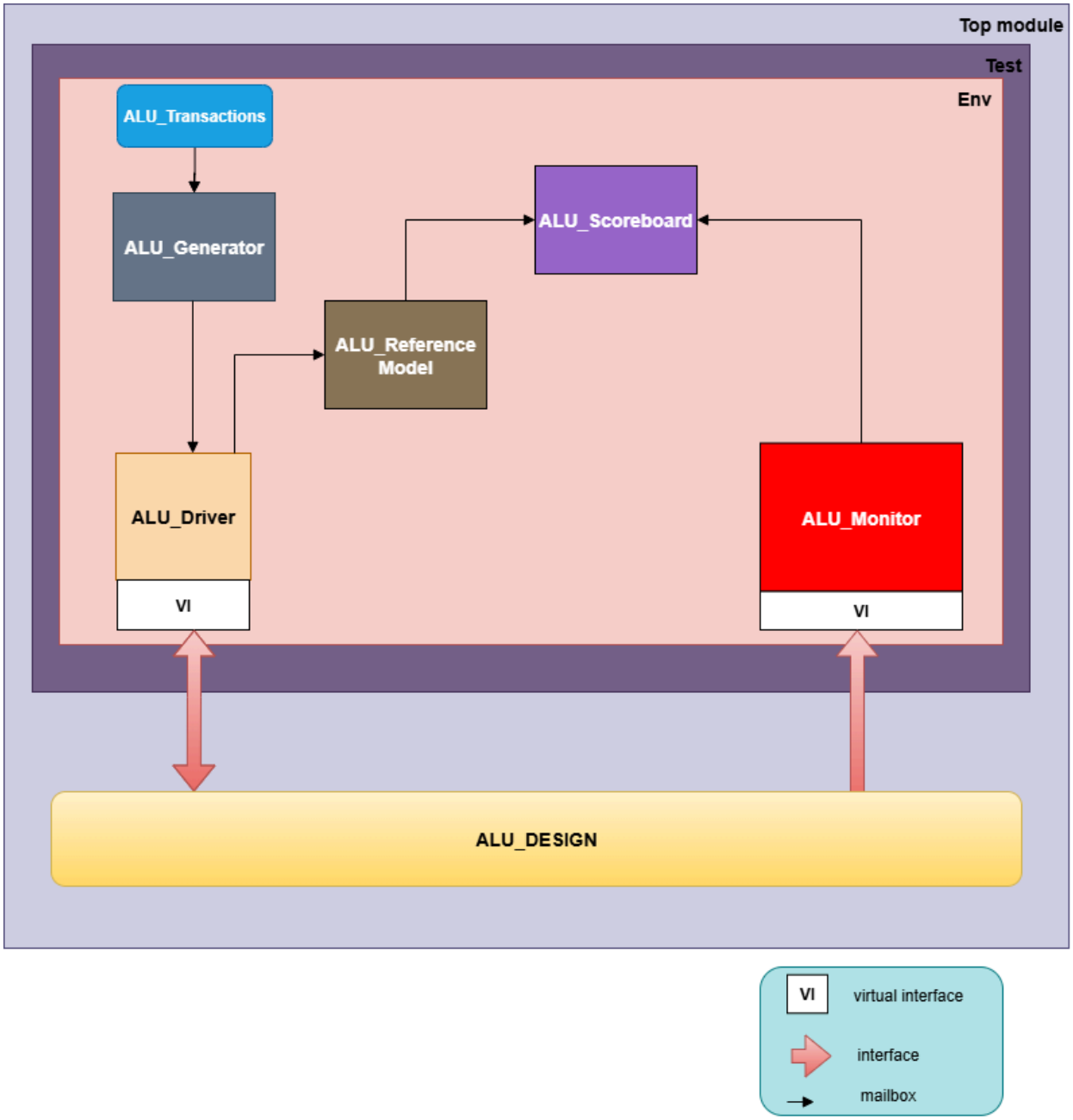
Signal Name	Type	Description
ERR	Output	Error signal - indicates if an error occurred during operation
RES	Output	Result - the output result of the ALU operation
OFLOW	Output	Overflow - indicates arithmetic overflow condition
COUT	Output	Carry Out - output carry from arithmetic operations
G	Output	Greater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

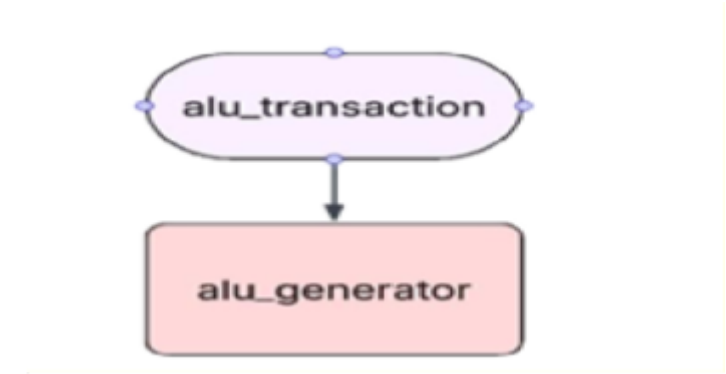


The figure shows general testbench architecture used for verifying digital designs. At the top module is the Design Under Verification (DUV). The test has the testbench environment that includes a transaction generator that creates and manages the input testcases for testing. These transactions are randomized 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.

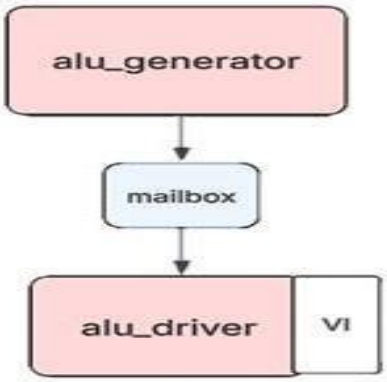
2.3 FLOW CHART OF SV COMPONENTS :

1.Transaction



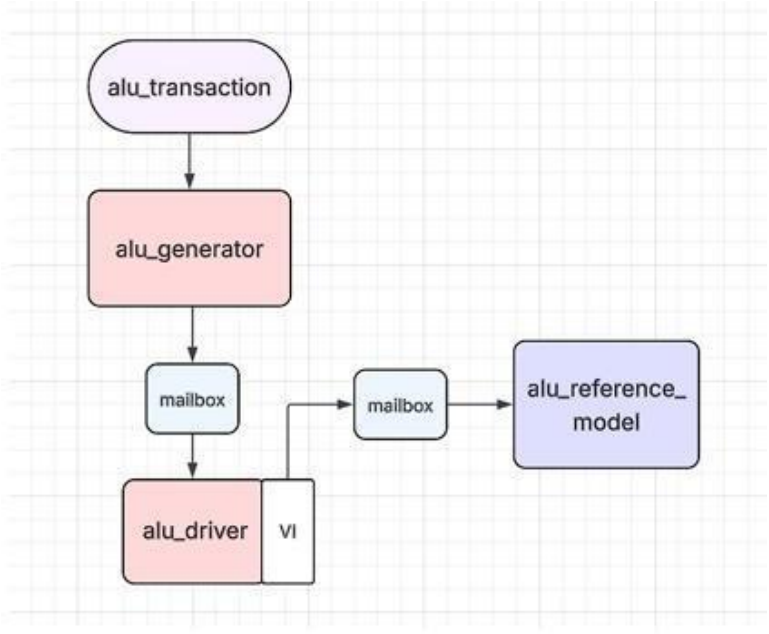
The Transaction class is the main data container used to keep communication between all testbench parts consistent. It defines common fields like address, data, and control signals in one standard format. It supports constrained randomization, which helps verification engineers create valid and realistic data based on actual use cases. The class includes useful methods like copy constructors for safe data transfer, comparison functions to check results, and display functions to help with debugging. This setup removes the need for each component to deal with low-level signal details, making the design more modular and easier to manage. It also allows for inheritance, so the class can be extended for specific protocols while still being clean and reusable in other projects. By putting all stimulus and response information in one format, the Transaction class helps data move smoothly between the Generator, Driver, Monitor, Reference Model, and Scoreboard.

2. Generator



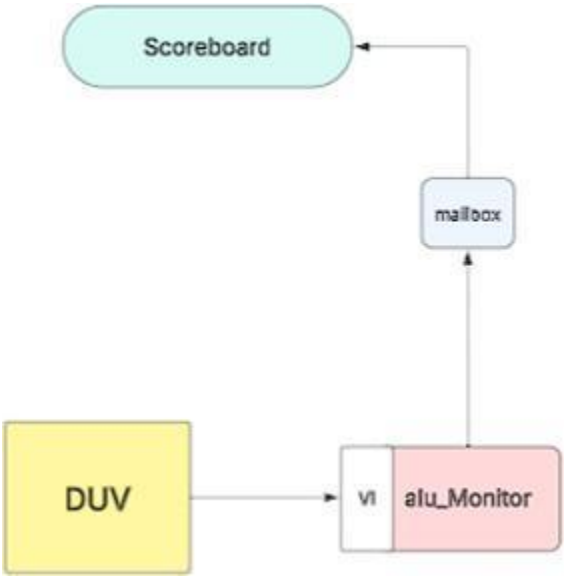
The Generator is a smart part that creates test inputs using advanced randomization methods. It helps build full verification tests by making many different transaction patterns to check all parts of the DUT's behavior. It uses several random strategies, including special tests for rare cases and weighted choices to improve coverage. The same transaction is sent to both the Driver and the Reference Model so they stay in sync. The Generator also supports extra features like timing rules for making sequences, checking combinations of events, and focusing on parts of the design that haven't been tested yet. It includes controls for switching between test cases, setting test stages, and using seeds to make tests repeatable.

3. Driver



The Driver is a key part that changes high-level transactions into exact signal patterns that follow the protocol and are sent to the DUT. It uses smart timing control to manage setup and hold times, multi-step protocols, and flow control methods like backpressure and handshaking. The Driver follows the interface rules closely and supports extra features like adding delays or purposely breaking rules to test how the DUT reacts. It uses virtual interfaces to keep the test logic separate from the DUT design, making it easy to reuse with other DUT versions. The Driver also has strong error checking and recovery features. It watches for protocol mistakes and timeouts and gives helpful debug information. This active part of the testbench makes sure all inputs sent to the DUT are correct in timing and follow the rules, giving reliable test results.

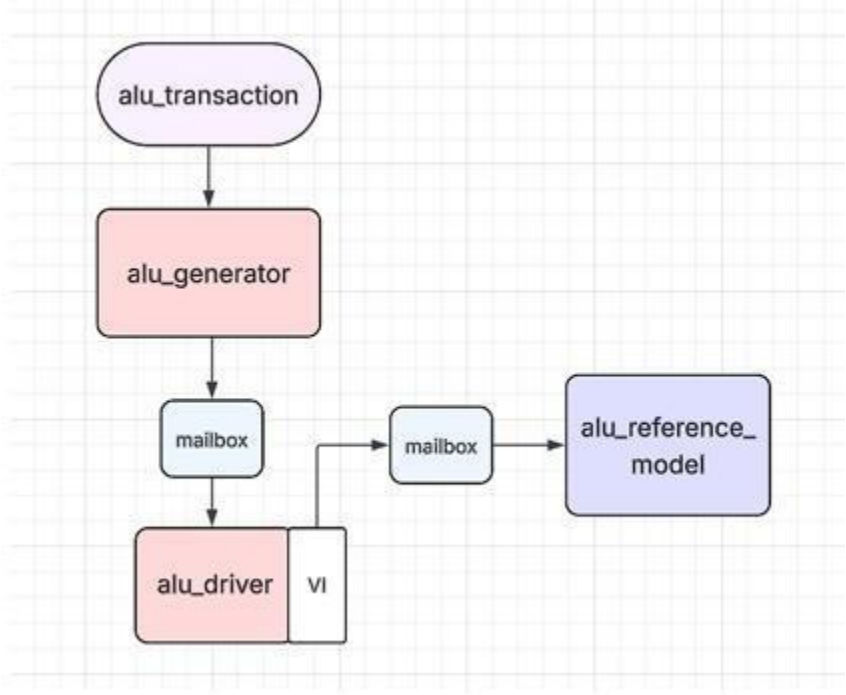
4.monitor



The Monitor is a passive part that watches the DUT's output signals without changing how the DUT works. It keeps checking the signals and turns them into high-level transactions. It uses smart methods to understand complex protocols, handle timing differences, and convert low-level signal activity into useful data. The Monitor can work with different interface types, burst data, and responses that come out of order, making sure it builds the right transaction from the signals. It uses advanced ways to catch signals, like edge detection and adjustable timing windows, to capture full transactions and avoid noise. It also checks for errors like protocol mistakes, timing issues, or strange

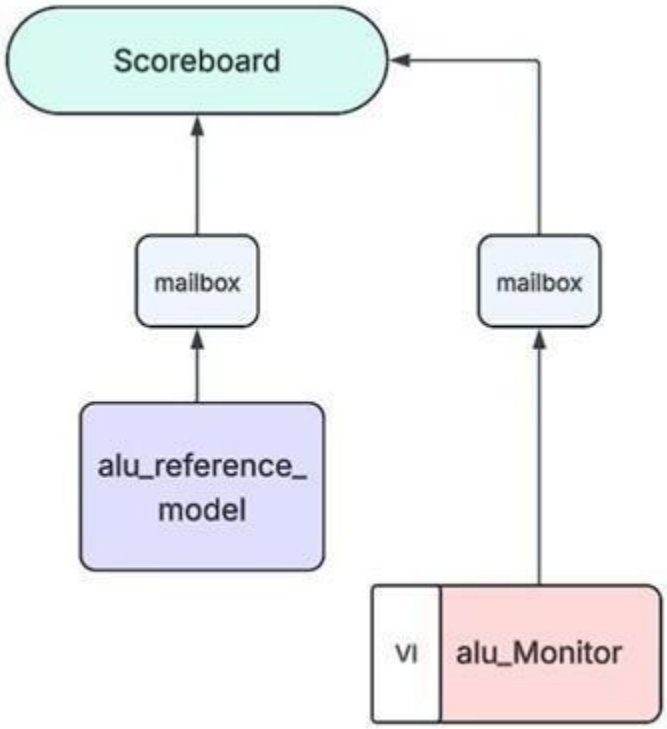
signal patterns and gives clear reports when something goes wrong. The Monitor also tracks coverage in real time, checking if all features and signal changes are tested, helping teams see how far the testing has gone.

5.reference model



The Reference Model is an independent part that shows how the DUT should work. It acts as the golden reference and creates the expected outputs for checking during verification. It takes the same input from the Generator but uses simple, high-level logic instead of hardware code, making sure it stays separate from the RTL design. The model can run in different modes and with different settings, so it can test the DUT under various conditions while keeping accurate timing when needed. It also keeps track of the DUT's internal states, memory, and settings during complex tests. This model is usually built from the design specs using methods that focus on being correct and easy to maintain. The expected results it produces are then used by the Scoreboard to check if the DUT outputs are right or wrong.

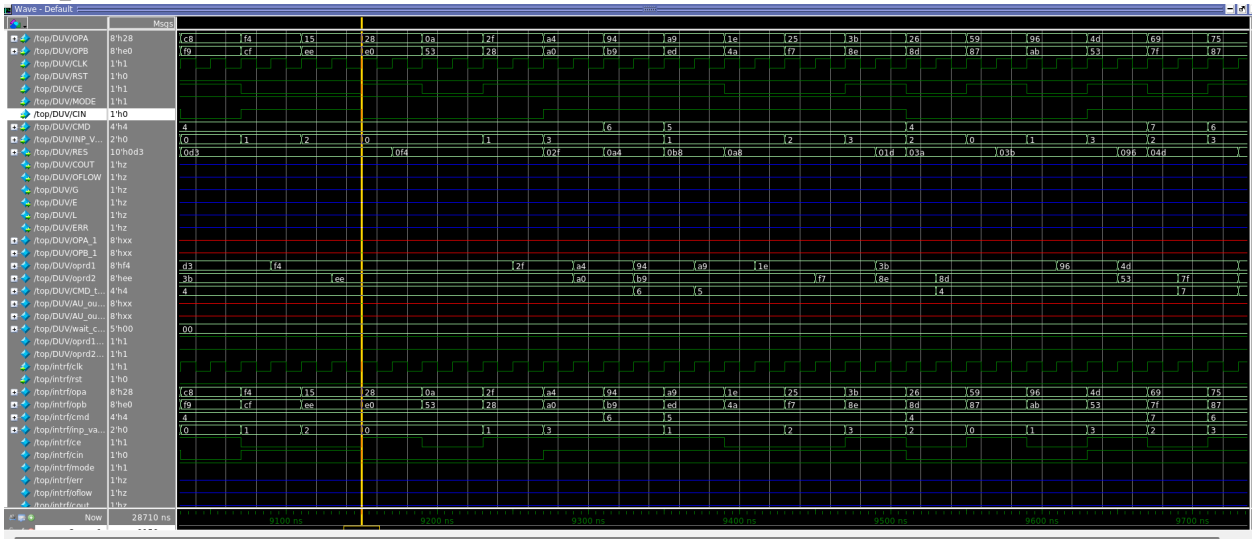
6.Scoreboard



The Scoreboard is the main checking part of the testbench that compares the expected results from the Reference Model with the actual outputs from the DUT, collected by the Monitor. It uses smart comparison methods to handle tough cases like delays, out-of-order responses, and multiple data streams. It matches data carefully, even when there are timing differences or small acceptable errors, and gives clear details when there is a mismatch to help find the problem. The Scoreboard creates full test reports with stats, coverage details, performance checks, and reasons for any failures. It allows different types of checks depending on the test phase, from basic checks to deeper performance testing. A live dashboard shows the current status of verification, how much coverage has been reached, and any failure patterns, helping teams make quick changes and debug faster.

CHAPTER 3- Result

output waveform



Assertion report

Assertions Coverage Summary:

Search:

Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
/top/intrf/assert__0	12338	356	999292	986597	1	0	5	Faile
/top/intrf/assert__1	14796	210	999292	984285	1	0	4	Faile
/top/intrf/assert__2	9861	172	999292	989257	1	1	4	Faile
/top/intrf/assert__3	106504	1632	999292	891155	1	0	2	Faile
/top/intrf/assert__ppt_clock_enable	0	422976	999292	576315	1	0	2	Covere
/top/intrf/assert__ppt_reset	0	1	999292	999291	0	0	1	Covere
/top/intrf/assert__ppt_valid_inp_valid	10	999281	999292	0	1	0	1	Faile
/top/intrf/assert__wait_16	79477	789	999292	919021	1	4	13	Faile
/work.alu_if/assert__0	12338	356	999292	986597	1	0	5	Faile
/work.alu_if/assert__1	14796	210	999292	984285	1	0	4	Faile
/work.alu_if/assert__2	9861	172	999292	989257	1	1	4	Faile
/work.alu_if/assert__3	106504	1632	999292	891155	1	0	2	Faile
/work.alu_if/assert__ppt_clock_enable	0	422976	999292	576315	1	0	2	Covere
/work.alu_if/assert__ppt_reset	0	1	999292	999291	0	0	1	Covere
/work.alu_if/assert__ppt_valid_inp_valid	10	999281	999292	0	1	0	1	Faile
/work.alu_if/assert__wait_16	79477	789	999292	919021	1	4	13	Faile

output functional coverage

Covergroup type:

mon_cg

Summary	Total Bins	Hits	Hit %
Coverpoints	13	10	76.92%
Crosses	0	0	0.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
cout	2	2	0	100.00%	100.00%	100.00%
equal	2	1	1	50.00%	50.00%	50.00%
error	2	2	0	100.00%	100.00%	100.00%
great	2	1	1	50.00%	50.00%	50.00%
less	2	1	1	50.00%	50.00%	50.00%
result	3	3	0	100.00%	100.00%	100.00%

input functional coverage

Covergroup type:

drv_cg

Summary	Total Bins	Hits	Hit %
Coverpoints	28	28	100.00%
Crosses	101	101	100.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CIN_CP	2	2	0	100.00%	100.00%	100.00%
CMD_CP	14	14	0	100.00%	100.00%	100.00%
INP_VALID_CP	4	4	0	100.00%	100.00%	100.00%
MODE_CP	2	2	0	100.00%	100.00%	100.00%
OPA_CP	3	3	0	100.00%	100.00%	100.00%
OPB_CP	3	3	0	100.00%	100.00%	100.00%

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CMD_X_IP_V	56	56	0	100.00%	100.00%	100.00%
MODE_X_CMD	28	28	0	100.00%	100.00%	100.00%
MODE_X_INP_V	8	8	0	100.00%	100.00%	100.00%
OPA_X_OPB	9	9	0	100.00%	100.00%	100.00%

overall coverage

top	91.13%	82.70%
intrf	96.62%	98.25%
DUV	89.61%	82.07%
alu_pkg	96.88%	95.15%
alu_transaction/copy	100.00%	100.00%
alu_transaction1/copy	100.00%	100.00%
alu_transaction2/copy	100.00%	100.00%
alu_transaction3/copy	100.00%	100.00%
alu_transaction4/copy	100.00%	100.00%
alu_transaction5/copy	100.00%	100.00%
alu_transaction6/copy	100.00%	100.00%
alu_transaction7/copy	100.00%	100.00%
alu_generator/new	100.00%	100.00%
alu_generator/start	100.00%	100.00%
alu_driver	99.05%	99.18%
alu_monitor	91.17%	87.50%
alu_reference_model/new	100.00%	100.00%
alu_reference_model/both_op	100.00%	100.00%
alu_reference_model/op_store	100.00%	100.00%
alu_reference_model/alu_result	100.00%	100.00%
alu_reference_model/start	88.00%	77.06%
alu_scoreboard/new	100.00%	100.00%
alu_scoreboard/start	100.00%	100.00%
alu_scoreboard/compare_report	100.00%	100.00%
alu_environment/new	100.00%	100.00%
alu_environment/build	100.00%	100.00%
alu_environment/start	100.00%	100.00%

Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Covergroups	142	139	3	1	97.88%	87.50%
Statements	539	512	27	1	94.99%	94.99%
Branches	127	120	7	1	94.48%	94.48%
FEC Expressions	8	8	0	1	100.00%	100.00%
FEC Conditions	51	40	11	1	78.43%	78.43%
Toggles	294	276	18	1	93.87%	93.87%