ALU Verification Plan

CHAPTER 1 – DESIGN OVERVIEW

1. ALU:

The ALU (Arithmetic Logic Unit) we're working with is essentially the computational heart of any digital system. Think of it as a highly sophisticated calculator that can handle both mathematical operations and logical decisions. This particular design stands out because it's built with flexibility in mind - it's parameterized, meaning we can configure it for different bit-widths depending on our application needs.

What makes this ALU special is its comprehensive approach to digital computation. It's not just doing basic addition and subtraction. We're talking about a unit that can handle complex operations like bit rotations, comparisons, and even has built-in intelligence for input validation and error detection. The design operates synchronously with proper clock and reset handling, making it suitable for modern digital systems.

1.2 Advantages of ALU:

- **Scalable Design:**
  You can start with 16-bit and easily scale to 32, 64, or 128-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 behavior. 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.

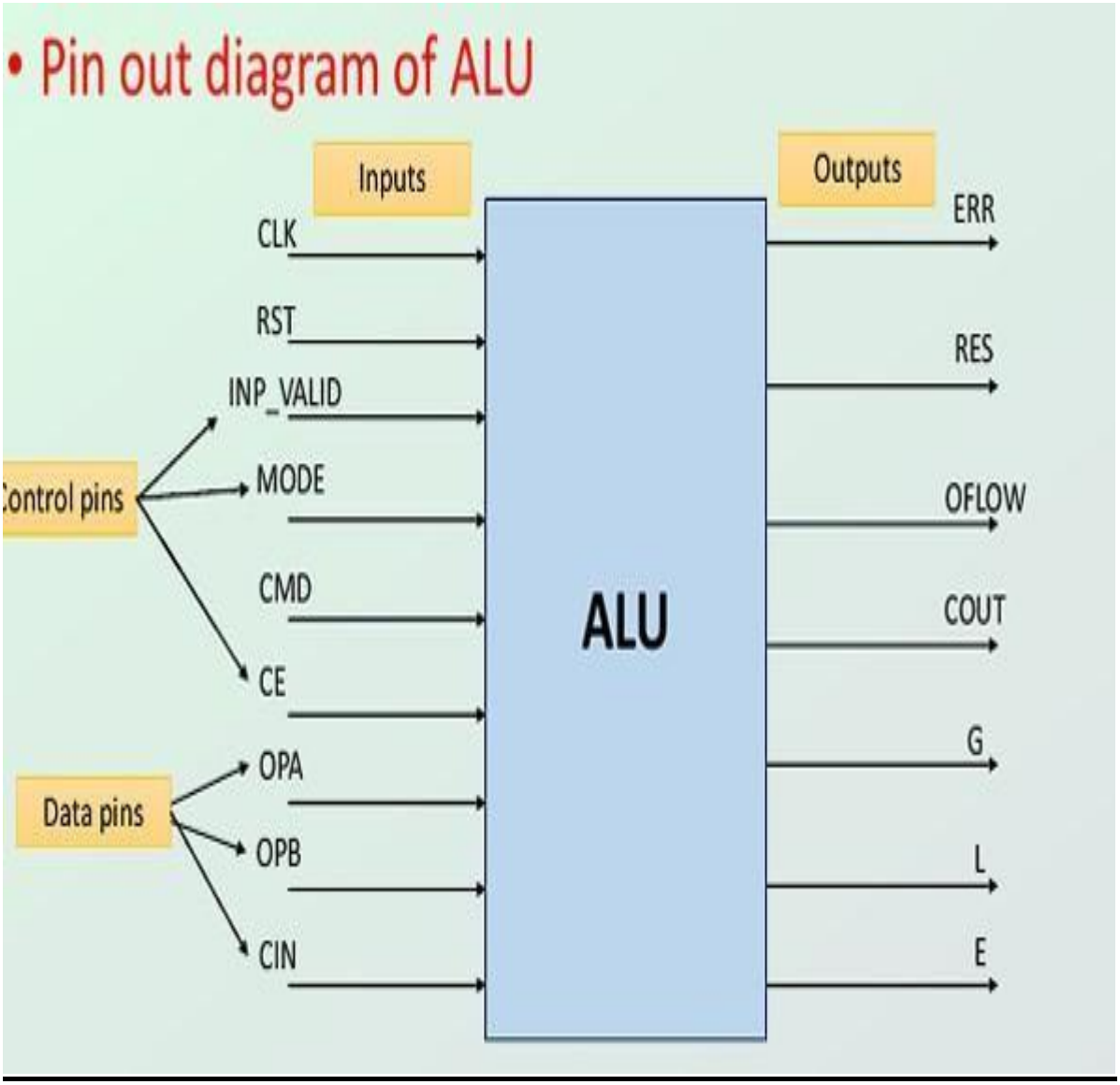
- **Limited Rotate Range**
  Rotate operations only support up to 8 positions — may not be enough for larger bit-widths.

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

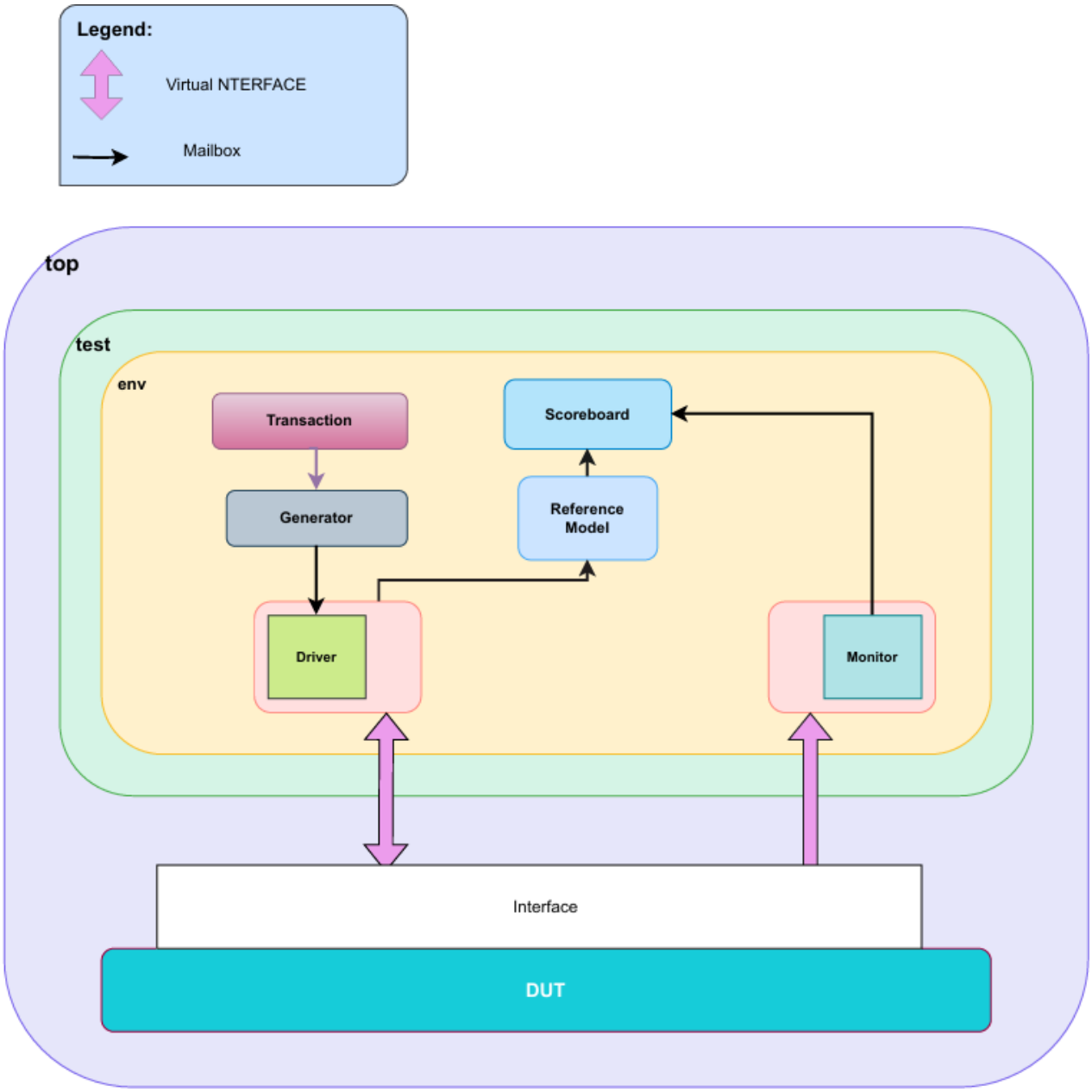
- **No Pipelining**
  It executes one operation at a time with no pipeline support, which can slow down performance in fast systems.

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

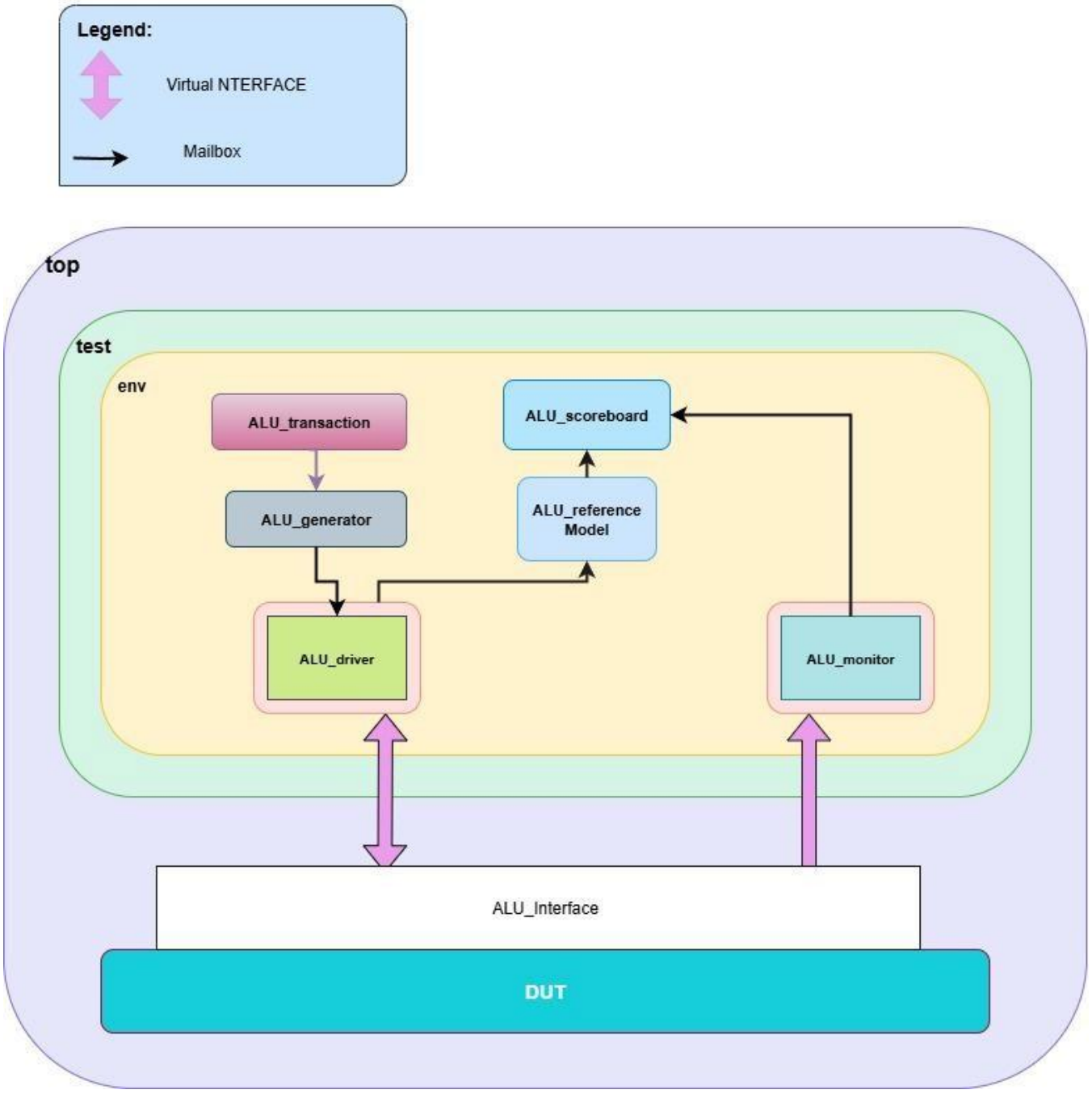
1.8 Design diagram with interface signals:



CHAPTER 2 - Verification Architecture

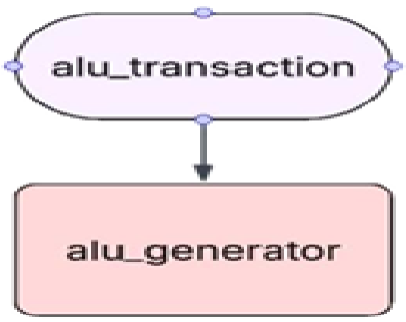


2. Verification Architecture for ALU:



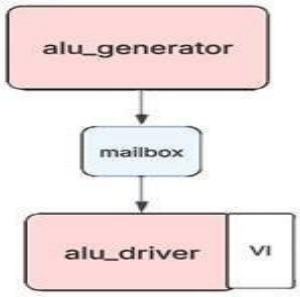
2.3 FLOW CHART OF SV COMPONENTS :

1. Transaction Class



The Transaction class serves as the fundamental data container that standardizes communication across all testbench components, defining fields like address, data, and control signals in a unified format. It implements constrained randomization capabilities, allowing verification engineers to specify realistic boundary conditions and valid data ranges that reflect actual usage scenarios. The class provides essential utility methods including copy constructors for safe data sharing, comparison functions for result verification, and display methods for debugging purposes. This abstraction eliminates the need for components to handle low-level signal details directly, promoting modularity and enabling higher-level test description. The design supports inheritance for protocol-specific extensions while maintaining clean interfaces that can be reused across different verification projects. By encapsulating all stimulus and response information in this standardized format, the Transaction class enables seamless data flow between Generator, Driver, Monitor, Reference Model, and Scoreboard components.

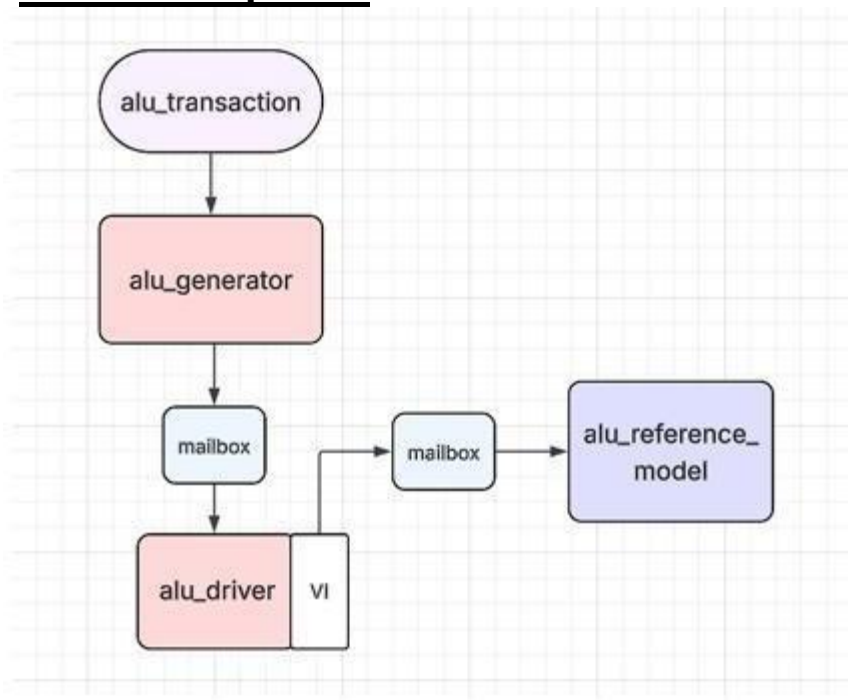
2. Generator Component



The Generator component functions as an intelligent test stimulus factory that creates comprehensive verification scenarios through sophisticated constraint-driven randomization techniques. It analyzes coverage requirements and systematically generates diverse transaction sequences that explore the complete

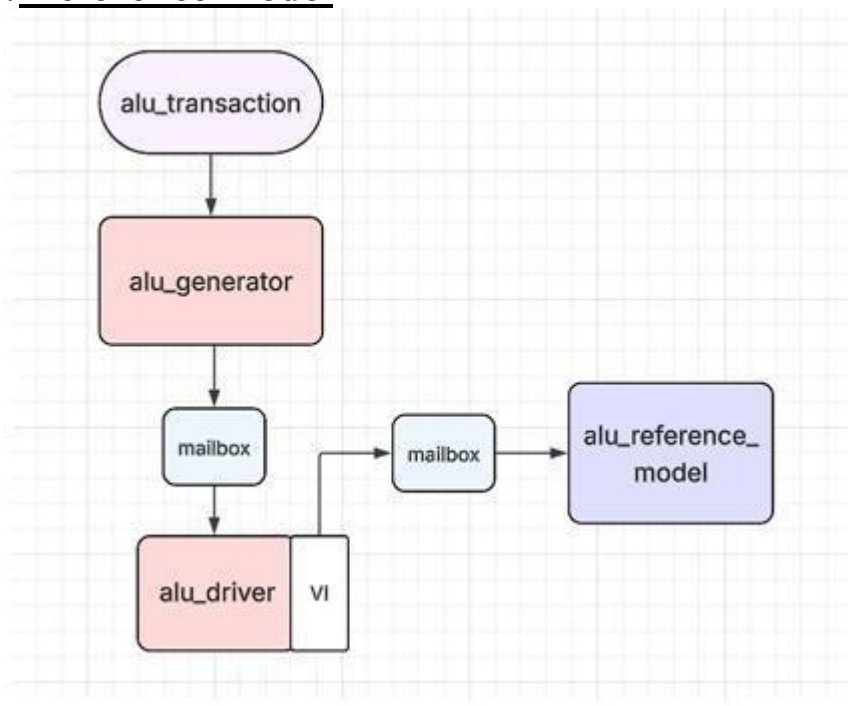
functional space of the DUT under test. The component implements multiple randomization strategies including directed testing for corner cases and weighted distributions for statistical coverage analysis. It distributes identical transaction copies to both the Driver for DUT stimulation and the Reference Model for expected result prediction, ensuring perfect synchronization. Advanced features include temporal constraints for sequence generation, cross-coverage tracking for interaction scenarios, and adaptive randomization that focuses on unexplored areas. The Generator also provides test control mechanisms like scenario transitions, configurable test phases, and seed management for reproducible verification campaigns.

3. Driver Component



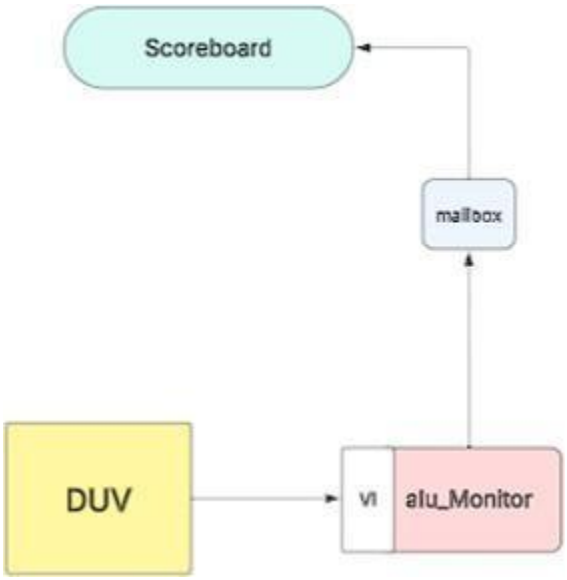
The Driver component serves as the critical interface translator that converts high-level transaction objects into precise, protocol-compliant signal sequences for DUT stimulation. It implements complex timing logic to handle setup and hold requirements, multi-phase protocols, and sophisticated flow control mechanisms including backpressure and handshaking. The component maintains strict adherence to interface specifications while supporting advanced features like configurable delay insertion and protocol violation injection for comprehensive testing. It utilizes virtual interface abstraction to ensure clean separation between verification logic and RTL implementation, promoting portability across different DUT versions. The Driver incorporates comprehensive error detection and recovery mechanisms, monitoring for protocol violations and timeout conditions while providing detailed debug information. This active verification component ensures that all stimulus applied to the DUT follows proper timing relationships and protocol requirements for reliable verification results.

4. Reference Model



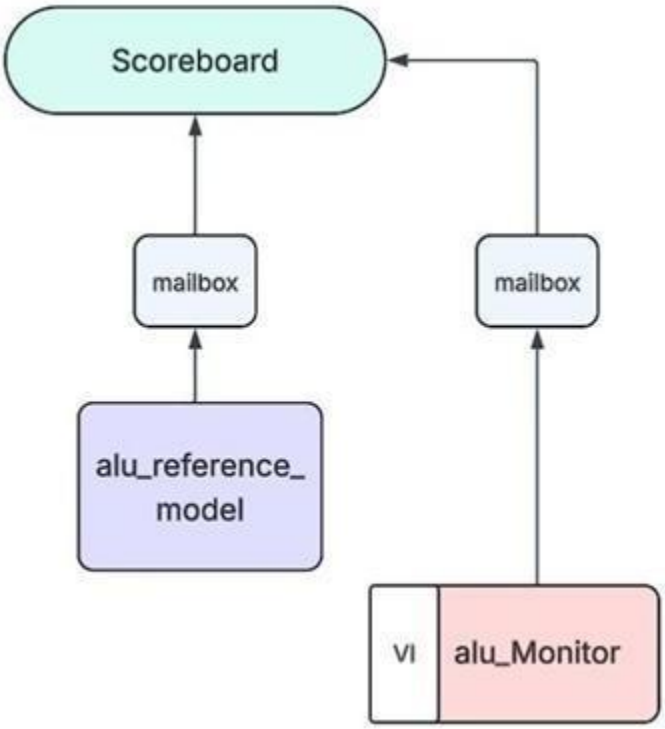
The Reference Model component implements an independent behavioral representation of DUT functionality, serving as a golden reference that generates expected responses for verification comparison. It processes identical input transactions from the Generator using high-level algorithmic approaches rather than hardware description languages, ensuring true independence from RTL implementation. The component supports multiple operational modes and configuration parameters, enabling verification of different DUT operating conditions while maintaining cycle-accurate timing where required. It incorporates sophisticated state management capabilities, tracking internal DUT states, memory contents, and configuration registers across complex test sequences. The Reference Model typically develops from specification documents using behavioral modeling techniques that prioritize accuracy and maintainability over implementation efficiency. This independent prediction engine provides the expected results that the Scoreboard uses to determine whether the actual DUT responses are correct or incorrect.

5. Monitor Component



The Monitor component operates as a passive, non-intrusive observer that continuously samples DUT output signals and reconstructs transaction-level information without affecting DUT performance. It implements sophisticated signal processing algorithms to decode complex protocol sequences, handle variable timing relationships, and convert low-level pin activity into high-level transactions. The component features intelligent protocol decoding capabilities that manage various interface standards, burst transactions, and out-of-order response scenarios for accurate reconstruction. It incorporates advanced sampling strategies including edge-sensitive detection and configurable observation windows to capture complete transaction sequences while filtering noise. The Monitor includes comprehensive error detection mechanisms that identify protocol violations, timing infractions, and unexpected signal patterns with detailed diagnostic reporting. Additionally, it implements real-time coverage collection, tracking functional coverage points and signal toggle coverage to provide immediate feedback on verification progress.

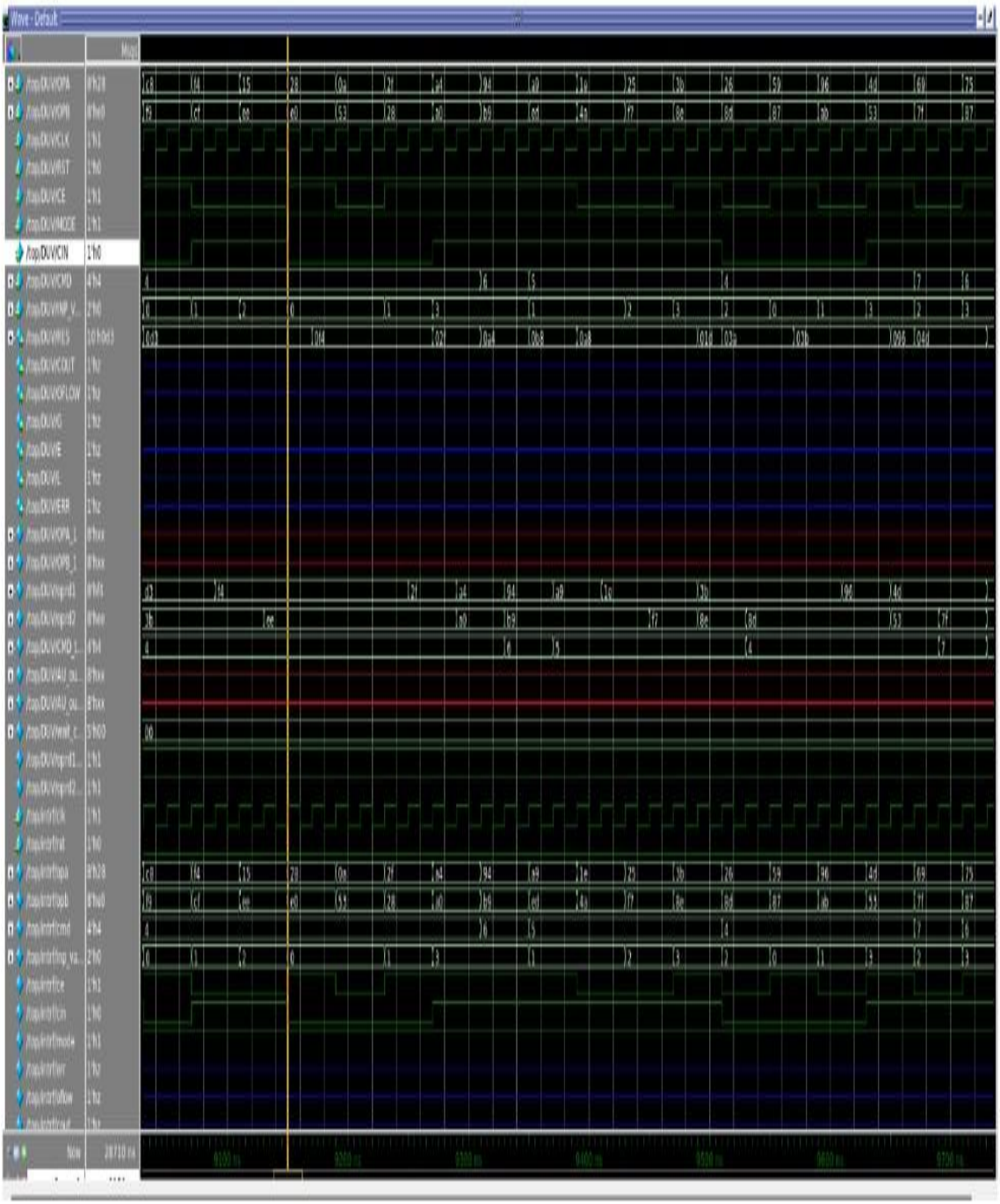
6. Scoreboard Component



The Scoreboard component functions as the central verification engine that compares expected results from the Reference Model against actual DUT responses captured by the Monitor. It implements sophisticated comparison algorithms that handle complex data correlation challenges including variable latency, out-of-order transactions, and multi-stream synchronization. The component manages intelligent matching strategies that accommodate timing variations and acceptable tolerance ranges while providing detailed mismatch analysis for failure diagnosis. It generates comprehensive test reports including statistical analysis, coverage summaries, performance metrics, and root cause analysis for any detected failures. The Scoreboard supports configurable comparison modes allowing different checking strategies for various verification phases from basic functionality to detailed performance validation. Real-time dashboard capabilities provide immediate feedback on verification progress, coverage achievement, and failure trends enabling rapid test strategy adjustments and efficient debugging workflows.

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/intf/assert__0	12338	356	999292	986597	1	0	5	Faile
/top/intf/assert__1	14796	210	999292	984285	1	0	4	Faile
/top/intf/assert__2	9861	172	999292	989257	1	1	4	Faile
/top/intf/assert__3	106504	1632	999292	891155	1	0	2	Faile
/top/intf/assert__ppt_clock_enable	0	422976	999292	576315	1	0	2	Covere
/top/intf/assert__ppt_reset	0	1	999292	999291	0	0	1	Covere
/top/intf/assert__ppt_valid_inp_valid	10	999281	999292	0	1	0	1	Faile
/top/intf/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%

Analysis:

- The 16-clock cycle timeout logic fails because the timeout flag does not assert after 16 cycles as expected.
- Single-operand operations (increment/decrement) are not executed unless both input valid signals are asserted, which violates the design requirements.
- The Carry-Out (COUT) flag does not assert during increment operations when an overflow occurs, such as incrementing from 255 to 256.
- The Increment A (INC_A) operation outputs the original OPA value instead of incrementing it, so no change is seen in the output.
- The Increment B (INC_B) operation erroneously decrements the value instead of incrementing as required.
- Shift operations are faulty: right shifts on OPA and OPB do not function, and the left shift on OPB produces incorrect results.
- The Decrement B (DEC_B) operation increases the value instead of reducing it, which is the opposite of intended behavior.
- Overflow flag logic fails for decrement operations; the OVER_FLOW indicator remains inactive even when decrementing below the minimum value.
- The Carry-In (CIN) signal has a one-cycle timing delay issue during addition and subtraction with carry-in, leading to functional inaccuracies.
- Rotate Right A by B (ROR_A_B) error detection does not assert the error flag, even when error conditions occur.
- The multiplication operation displays various unexpected functional issues.
- Logical OR operations do not complete successfully even when valid input values and proper timing are provided.