ALU VERIFICATION DOCUMENT

VERIFICATION DOCUMENT ALU

CHAPTER 1 – PROJECT OVERVIEW

1.1 ALU Introduction:

This project aims to design a flexible and high-performance Arithmetic Logic Unit (ALU) suitable for a variety of digital systems. 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. 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 12 operations and Logical mode supports 14 operations, covering everything from basic arithmetic, such as addition, subtraction, and multiplication, to logical operations like AND, OR, XOR, and advanced functions, including bit rotations. 4-bit command input, which ensures logical separation and efficient decoding of operations.

The INP_VALID signal helps manage scenarios where operands may arrive at different times, which is common in pipelined or asynchronous systems. 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.

Advantages of ALU:

- **Flexible bit width support :**
  The design is scalable it can start as a Parameterized-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.
- **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).
- **Detection:**

  The ALU is capable of catching invalid operation scenarios, particularly during rotate operations where improper input formats can lead to malfunction.

Disadvantages of ALU:

- **Fixed Timeout Duration:**

  The input waiting mechanism is hardcoded to wait for exactly 16 clock cycles. This may not be ideal for all systems 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
- **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

Use Cases of ALU:

- **Arithmetic Operations:**

  Handles essential arithmetic functions such as addition, subtraction, multiplication, and division.
- **Logical Operations**

Executes standard logic operations like AND, OR, XOR, and NOT. These are widely used for value comparisons, and applying bitwise masks in digital systems.

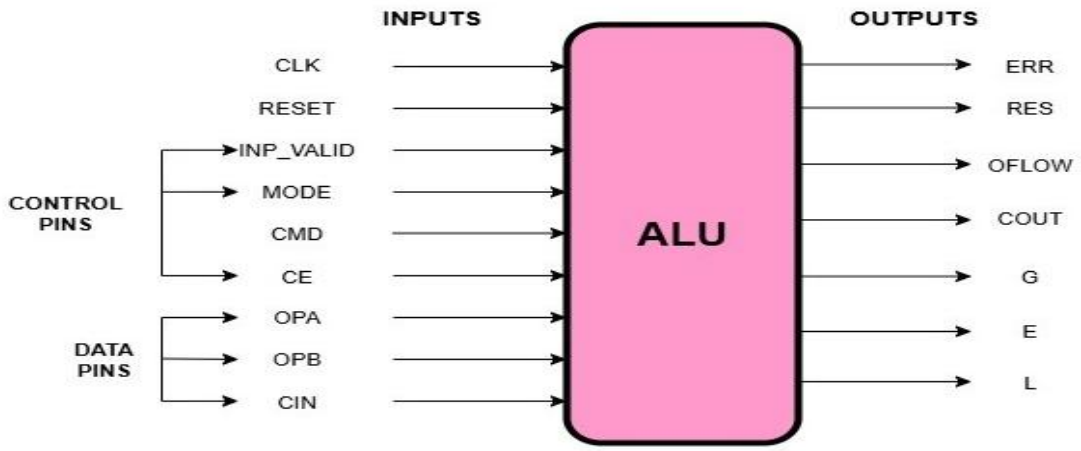
- **Bit Manipulation (Shifting & Rotation):**
  Supports left and right shifts, as well as bit rotations, which are vital in tasks such as data encoding, cryptographic algorithms, and efficient bit-level operations.
- **Data Comparison:**
  Compares two operands to determine equality, greater than, or less than relationships. This functionality supports conditional logic, loop control, and decision branching in both software and hardware.
- **Carry and Overflow Detection:**
  Monitors for carry-out or arithmetic overflow, ensuring proper handling of unsigned data and maintaining operation accuracy during calculations.

1.2 Verification Objectives

- Review design documentation: analyze functional requirements, supported operations, inputs/outputs, operating modes and edge-case behaviors.
- Construct the verification plan.
- Develop functional coverage and assertion plan.
- Frame the testbench architecture for the alu design.
- Creation of template codes for testbench components
- Implement and enhance testbench for coverage, integrating the functional coverage and SystemVerilog assertions.
- Validating the functional correctness of the ALU, considering the corner cases as well.
- Validating the timing of the operations.
- Checking its robustness against errors.

1.3 DUT Interfaces:

These are the signals present in the interface which is to be shared between the Test and Design.

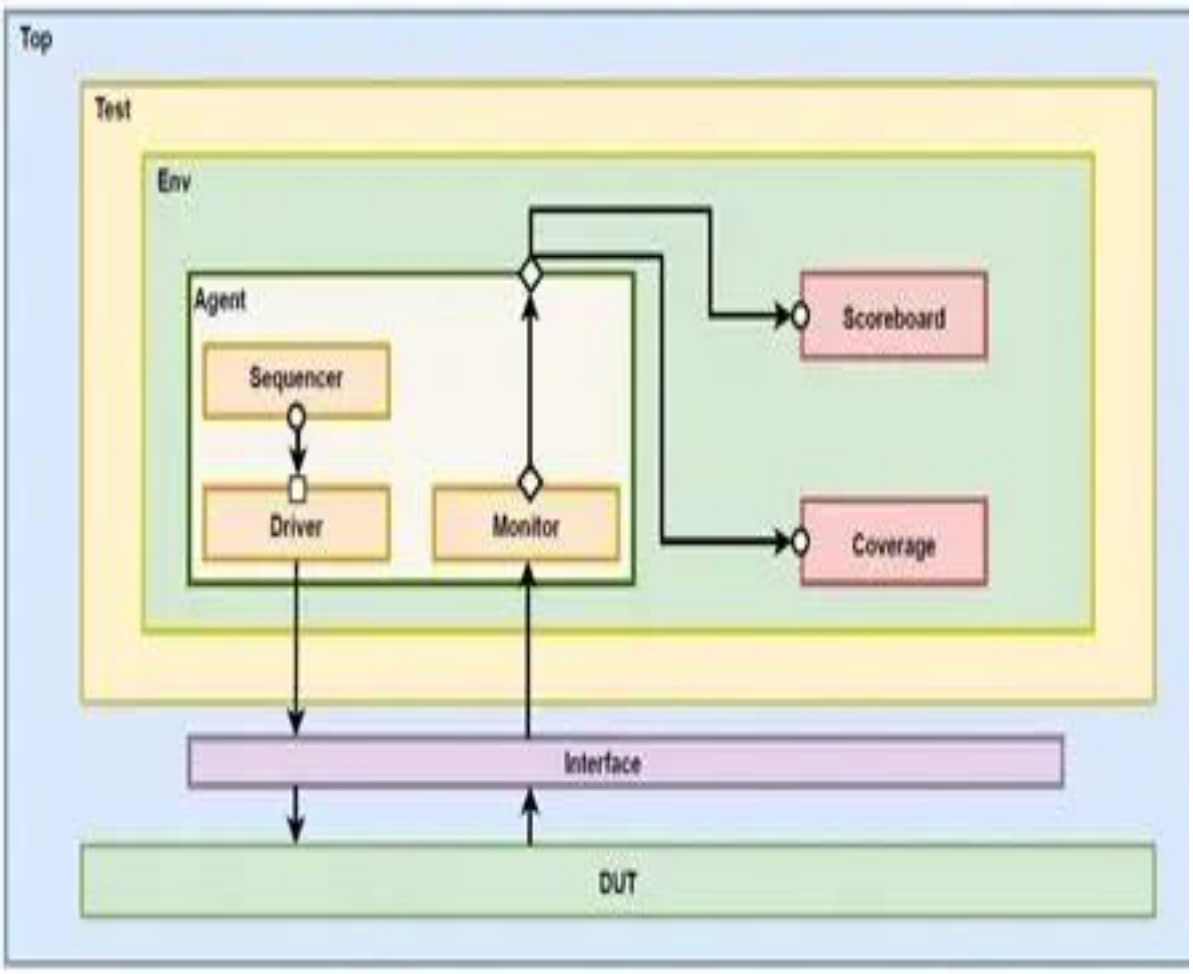


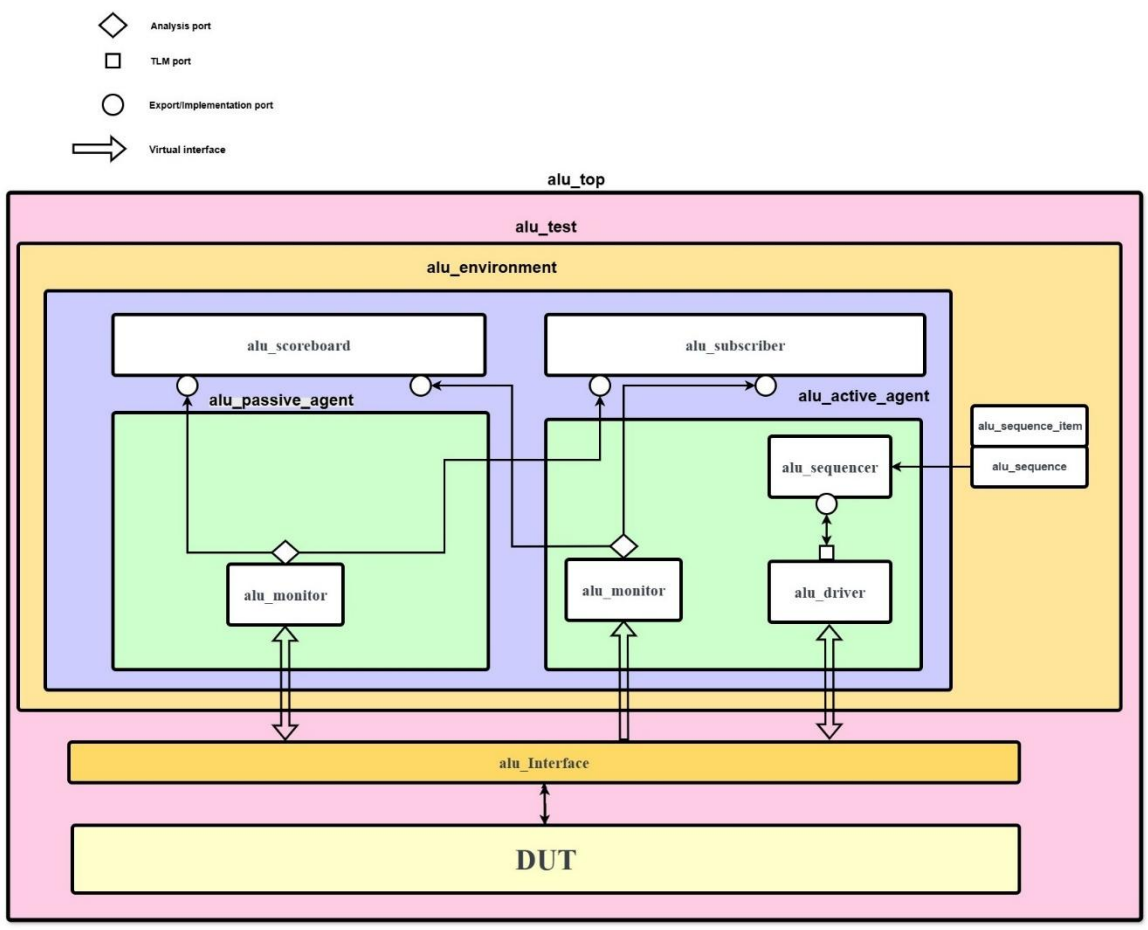
Single Name	Type	Size(bits)	Description
CLK	INPUT	1	Clock signal which is used for sample the values
RESET	INPUT	1	Asynchronous Active High RESET
CE	INPUT	1	Clock enable used to enables the design
INP_VALID	INPUT	2	Shows the Validity of the Operands(active high) MSB shows the validity of the OPB and LSB shows the validity for OPA.
MODE	INPUT	1	If the value is 1 the ALU is in Arithmetic Mode else it is in Logical Mode
CMD	INPUT	Parameterized	Commands for the Operation
OPA	INPUT	Parameterized	Operand 1
OPB	INPUT	Parameterized	Operand 2
CIN	INPUT	1	Carry in signal used for add_cin and sub_cin

ERR	OUTPUT	1	Active High error flag
RES	OUTPUT	Parameterized	Result of the instruction Performed by the ALU
OFLOW	OUTPUT	1	Overflow flag
COUT	OUTPUT	1	Carry out flag
G	OUTPUT	1	Greater than flag
E	OUTPUT	1	Equality flag
L	OUTPUT	1	Lesser than flag

CHAPTER 2 – TESTBENCH ARCHITECTURE AND METHODOLOGY

2.1 Testbench Architecture:



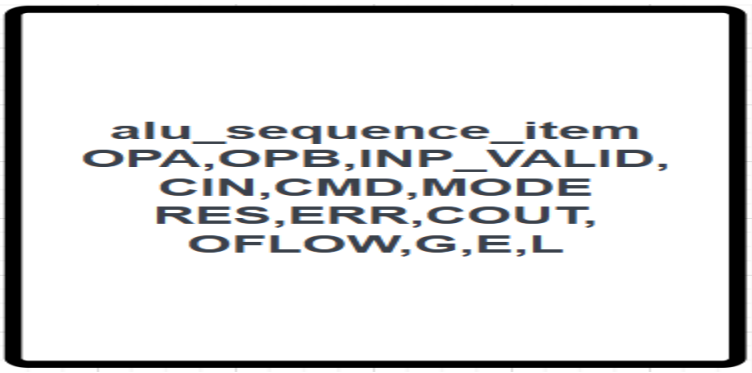


2.2 Components Details and Flowchart:

Sequence_item:

A sequence item in UVM is a user-defined class that acts like a container for all the signals that the ALU testbench needs to drive and sample.

It represents one transaction of data passed between the testbench components (like sequencer, driver, monitor, scoreboard) using TLM ports.



Input Fields (stimulus signals that drive the ALU)

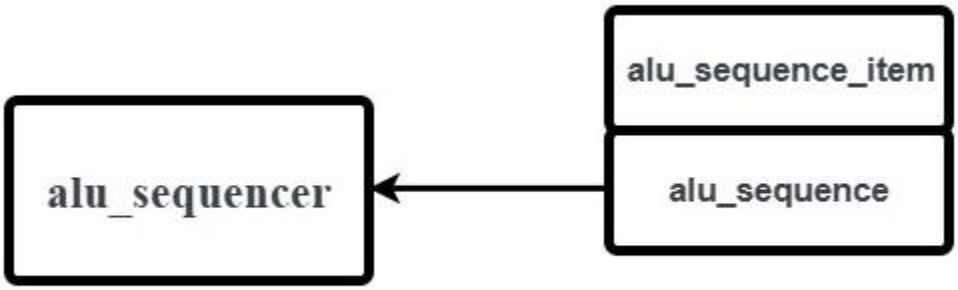
- **OPA, OPB** - The two operands on which the ALU will perform operations.
- **CIN** - Carry-in signal, used for arithmetic operations like addition.
- **CE** - Clock Enable, controls whether the ALU operation should be active.
- **MODE** - Defines the ALU's operation type (e.g., arithmetic vs logic).
- **CMD** - Specifies the exact operation (e.g., ADD, SUB, AND, OR).
- **INP_VALID** - Specifies the Validation of Operands.

Output Fields (response signals from the ALU)

- **RES** - The computed result of the operation.
- **OFLOW** - Indicates if an arithmetic overflow occurred.
- **COUT** - Carry-out signal, relevant in addition/subtraction.
- **G, L, E** - Comparison flags showing whether operand A is greater, less, or equal to operand B.

- **ERR** - Error flag, raised if an invalid command/mode is given or if an unexpected condition happens.

Sequence:



Generates the stimulus containing randomized ALU inputs (OPA, OPB, CMD, CIN, MODE, CMD) by randomizing the seq_item's.

Controls test variability by applying constraints to randomization (e.g., valid opcodes, corner-case operands).

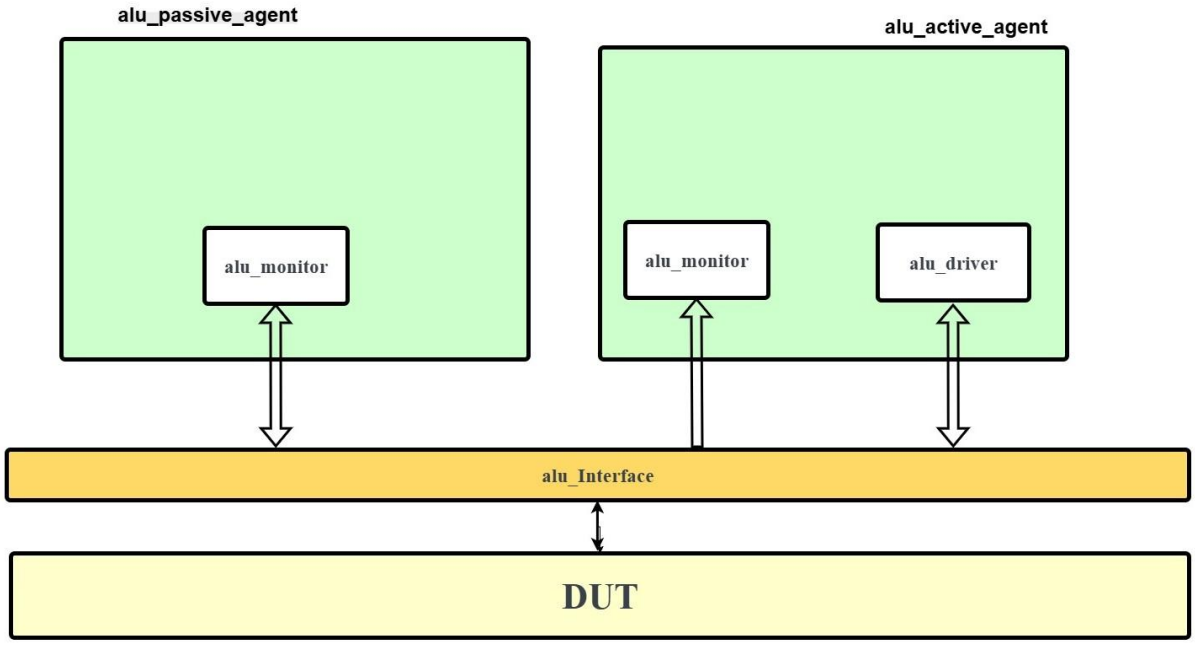
These seq_items are sent to the sequencer req fifo where they are stored.

Interface:

It is used to bundle all inputs and outputs of DUT, so we don't have to instantiate DUT every time and can directly use an interface.

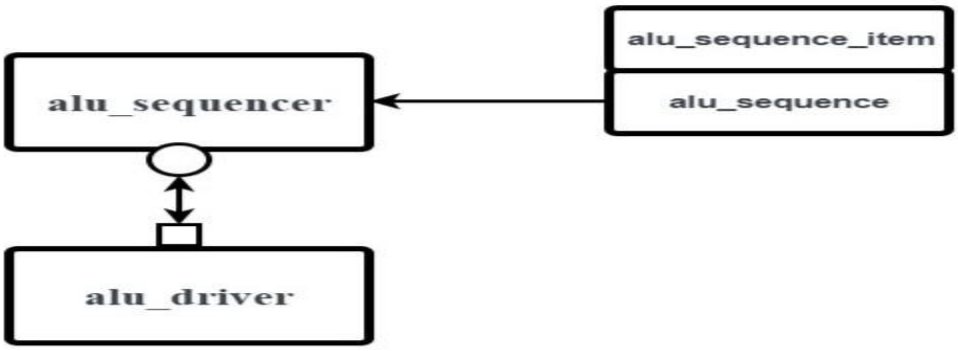
It is used to drive the inputs from the driver to the DUT, and used to access output values from DUT to the monitor.

It directly maps to the DUT's input and output ports. This interface is accessed by testbench components such as the driver and monitor using virtual interface handles, enabling structured and reusable connectivit



Sequencer:

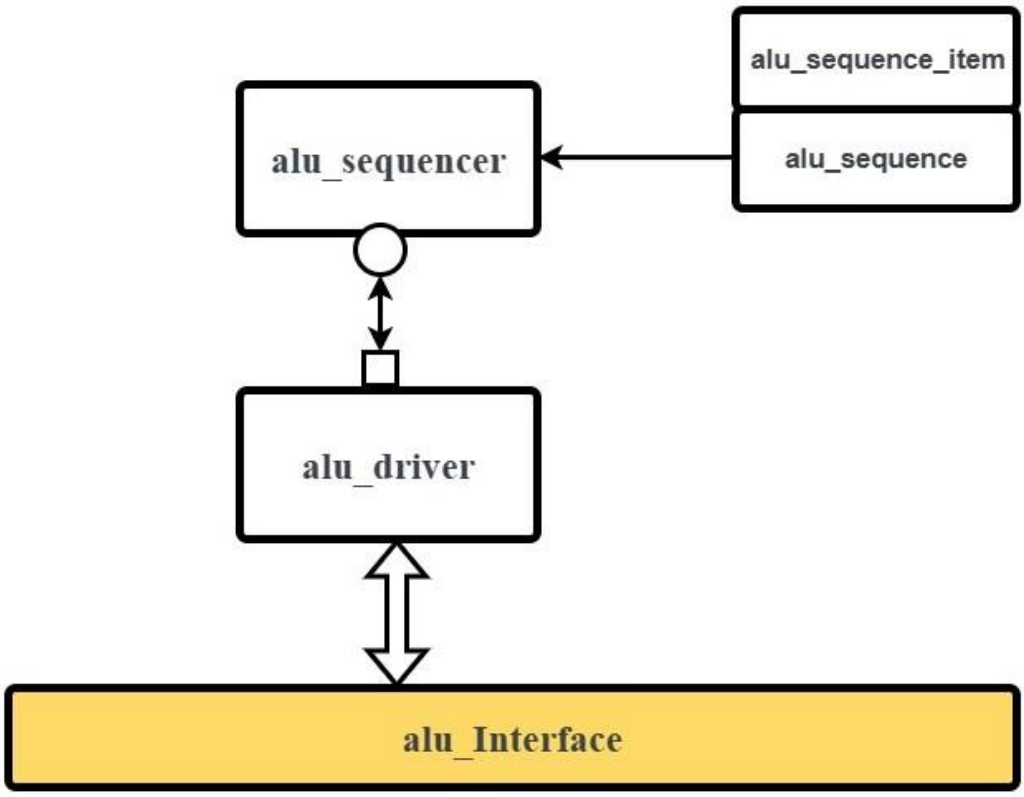
Controls the transaction level communication between sequence and driver by establishing a connection between them. Ultimately, it passes transactions or sequence items to the driver so that they can be driven to the DUT.



(seq_item_port in an in-built method of uvm_driver, which is used to request items from sequencer. seq_item_export is an inbuilt method of uvm_sequencer which is used to send seq_items to the driver.)

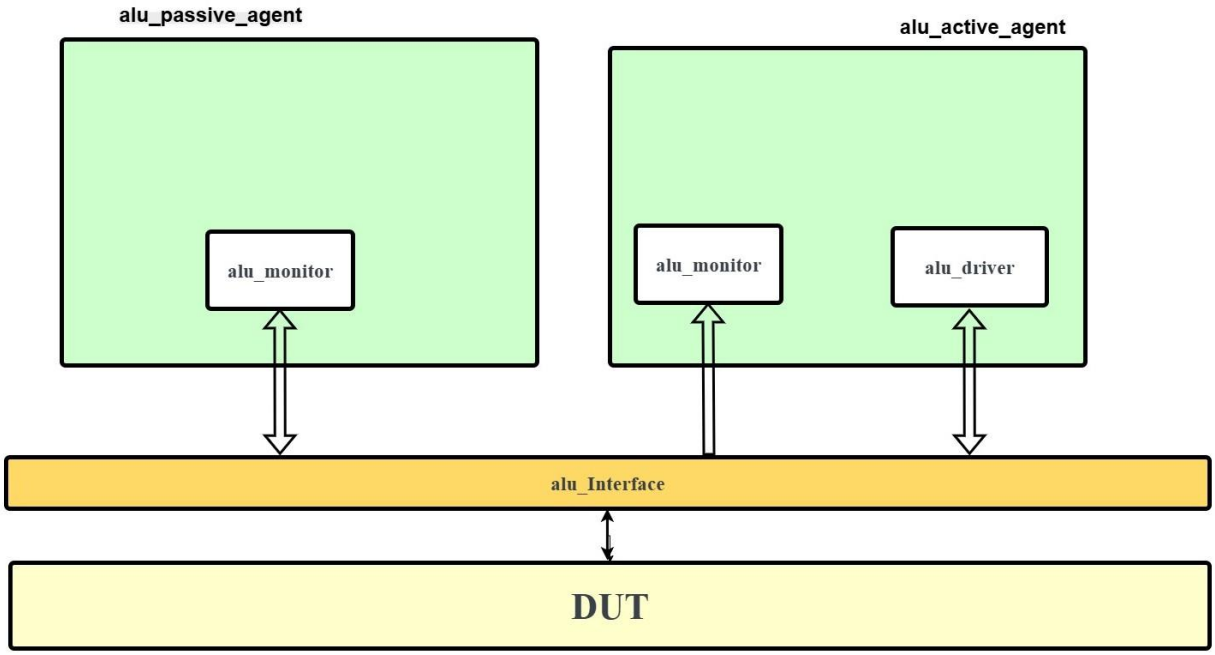
Driver:

Driver component converts high-level transactions into pin-level activity at the DUT inputs. It receives transactions from the sequencer via a TLM ports and drives them to the DUT using a virtual interface.



Driver retrieves the virtual interface from uvm_config_db.Handles special cases (single operand commands, multi operand commands with wrong inp_valid, multiplication operations).

Monitor:



Monitor component converts pin-level activity from the DUT outputs into high-level transactions. It captures the DUT's output signals via a virtual interface and packages them into transactions, which are then sent to the scoreboard and coverage through a TLM ports for functional Coverage, comparison and analysis.

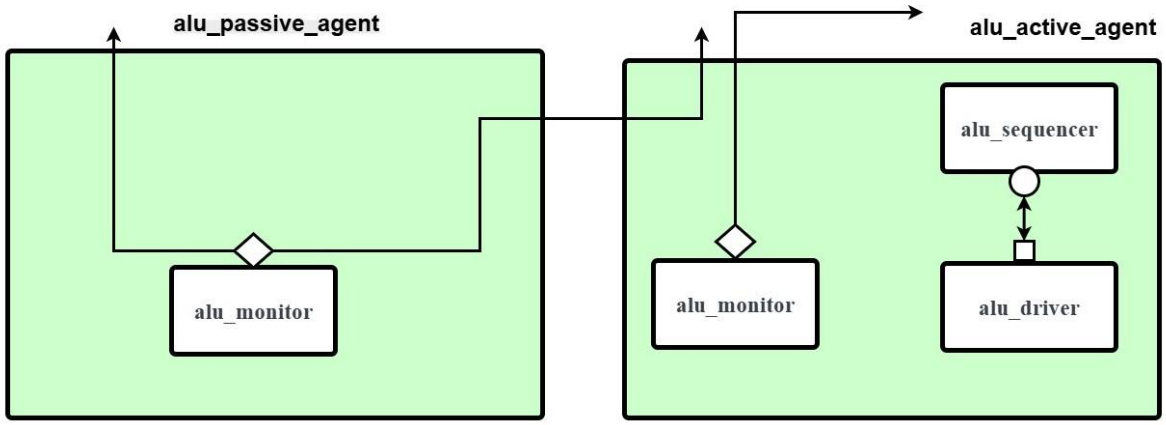
Here monitor is present in both Active and Passive agents ,Where Monitor in the Active agent takes the signal from the Interface and send to Coverage for calculate input coverage and to the Scoreboard to find expected result. Monitor in the Passive agent takes signals from interface and send to Coverage for calculate output coverage and to the Scoreboard as actual result.

Agents:

In this testbench, two agents are used to manage stimulus and data flow.

Active Agent : This agent is responsible for driving stimulus to the DUT. It contains three main components: a driver, a sequencer, and a monitor. The monitor within this agent observes the input

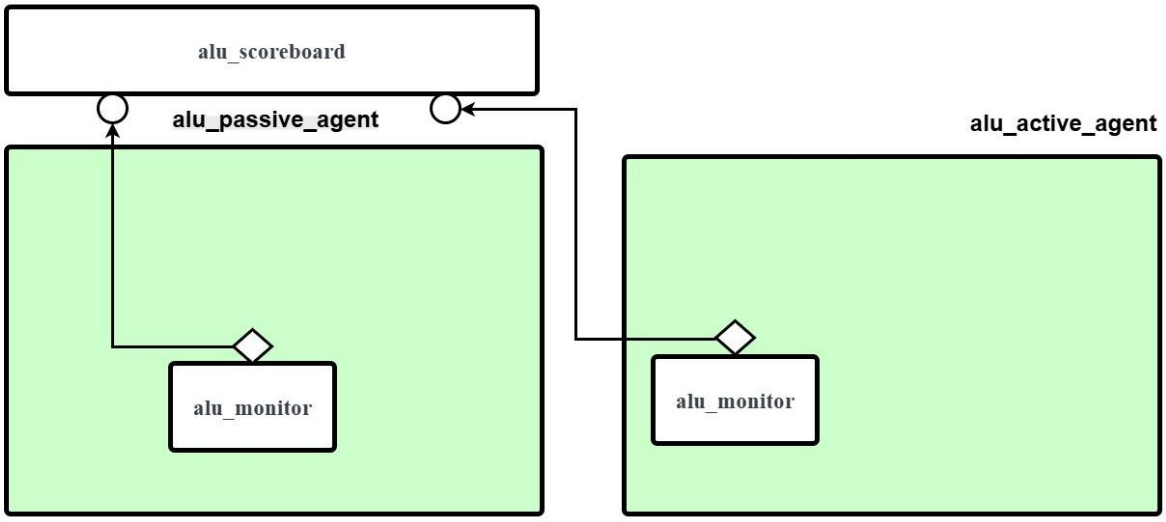
transactions being driven to the DUT and sends them to the scoreboard to be used as the expected input for the result calculation and to the coverage to calculate the input coverage.



Passive Agent : This agent is responsible for observing the DUT's behaviour. It contains only a monitor. This monitor observes the output pins of the DUT and converts the pin-level activity into a data transaction, which is then sent to the scoreboard for comparison and to the coverage for calculate output coverage.

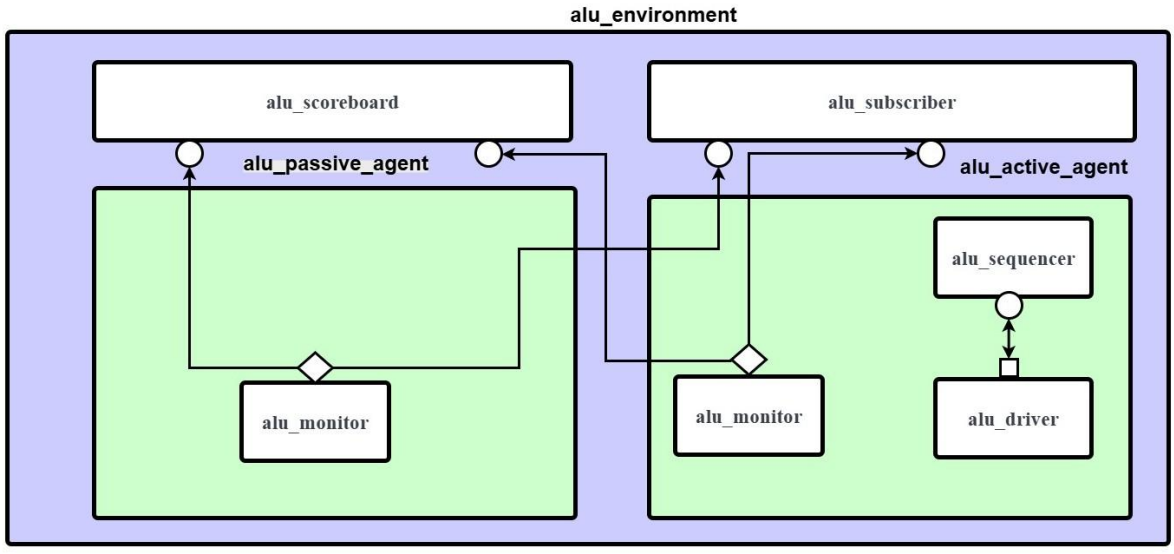
Scoarboard:

The scoreboard is a central component for checking the correctness of the DUT's operation. It receives transactions from both the active and passive agent monitors using analysis implementation-ports. It takes the input transaction from the active monitor, calculates the expected result, and then compares it with the actual output transaction received from the passive monitor. The scoreboard reports any discrepancies, identifying pass/fail conditions for each operation.



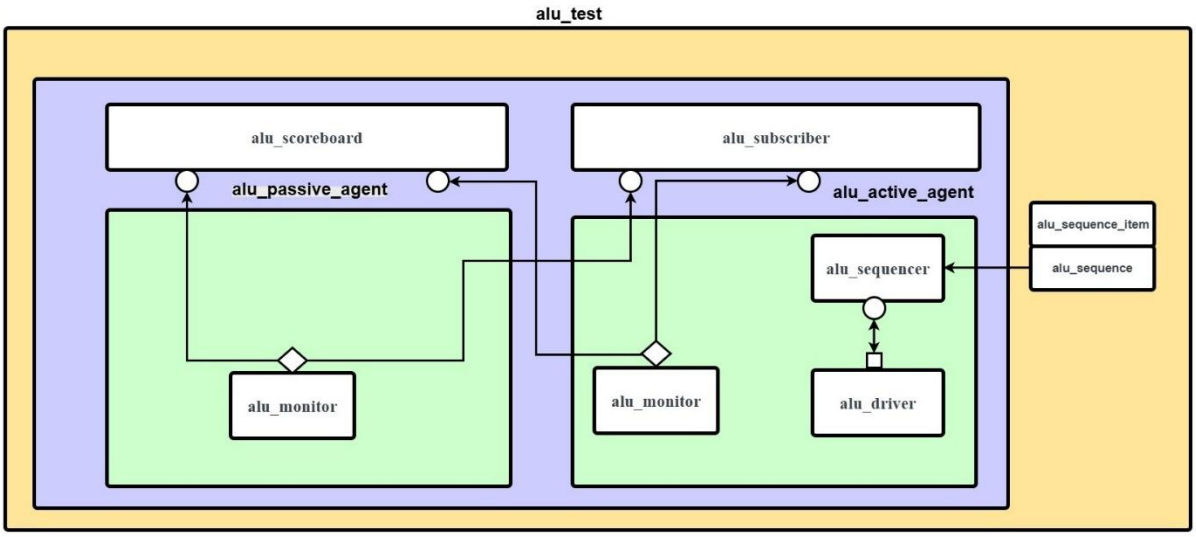
Environment:

The testbench environment serves as the top-level component that instantiates and connects all other UVM components. It encapsulates the complete verification environment, including the agents, scoreboard, and configuration objects. Its primary purpose is to manage the hierarchy and communication between different parts of the testbench.



Test:

The test component configures the environment, agents, and sequences. It orchestrates simulation runs by instantiating appropriate sequences, and managing overall verification intent.



CHAPTER 3 – VERIFICATION RESULT AND ANALYSIS

3.1 Errors in DUT:

Incorrect Output for Two Operand Operation:

SUB_CIN : When both operands are equal and cin is high, the overflow signal is not being asserted as expected.

INC_A: These operations are not functioning correctly;

INC_B: These operations are not functioning correctly;

DEC_B: These operations are not functioning correctly;

MUL_SHIFT: These operations are not functioning correctly;

OR: These operations are not functioning correctly;

SHR1_A: These operations are not functioning correctly;

SHR1_B: These operations are not functioning correctly;

ROR_A_B: Although the rotate-right operation seems to be functioning, the error signal is not being asserted in failure scenarios.

- **When a two-operand operation is issued, and INP_VALID is '11', the error is not being asserted.**
- **Similarly, when INP_VALID is '00', the expected error condition is not triggered.**
- **During the 16-cycle wait state, if INP_VALID transitions from '01' to '10' in the next cycle, the design incorrectly takes both inputs as valid and performs the operation, which leads to erroneous output.**

3.2 Coverage Report:

Input Coverage:

Covergroup type:

input_cov

Summary	Total Bins	Hits	Hit %
Coverpoints	554	554	100.00%
Crosses	64	64	100.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CARRY_IN	2	2	0	100.00%	100.00%	100.00%
CLOCK_ENABLE	2	2	0	100.00%	100.00%	100.00%
COMMAND	32	32	0	100.00%	100.00%	100.00%
INPUT_VALID	4	4	0	100.00%	100.00%	100.00%
MODE	2	2	0	100.00%	100.00%	100.00%
OPERAND_A	256	256	0	100.00%	100.00%	100.00%
OPERAND_B	256	256	0	100.00%	100.00%	100.00%

Search:

Crosses	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
MODE_CMD_	64	64	0	100.00%	100.00%	100.00%

Output Coverage:

Covergroup type:

output_cov

Summary	Total Bins	Hits	Hit %
Coverpoints	264	264	100.00%
Crosses	0	0	0.00%

Search:

CoverPoints	Total Bins	Hits	Misses	Hit %	Goal %	Coverage %
CARR_OUT	2	2	0	100.00%	100.00%	100.00%
EQUAL	1	1	0	100.00%	100.00%	100.00%
ERROR	1	1	0	100.00%	100.00%	100.00%
GREATER	1	1	0	100.00%	100.00%	100.00%
LESSER	1	1	0	100.00%	100.00%	100.00%
OVERFLOW	2	2	0	100.00%	100.00%	100.00%
RESULT_CHECK	256	256	0	100.00%	100.00%	100.00%

Assertion:

Questa Assertion Coverage Report

Show All | Show Covered | Show Missing

Assertions	Failure Count	Pass Count	Attempt Count	Vacuous Count	Disable Count	Active Count	Peak Active Count	Status
assert__VALID_INPUTS_CHECK	0	641350	810101	168751	0	0	1	Covered
assert__ppt_clock_enable	0	168750	810101	641351	0	0	2	Covered
assert__invalid_00	39291	1328	810101	769482	0	0	2	Failed
assert__out_range_logical	74347	0	810101	735754	0	0	2	Failed
assert__Invalid_command	81256	0	810101	728845	0	0	2	Failed
assert__0	10157	0	810101	799944	0	0	2	Failed
assert__ppt_timeout_logical	11761	4	810101	798336	0	0	17	Failed
assert__ppt_timeout_arithmetic	11274	210	810101	798617	0	0	17	Failed
assert__ppt_reset	0	1	810101	810100	0	0	1	Covered
assert__clk_valid_check	0	810101	810101	0	0	0	1	Covered
assert_rst_valid	0	810101	810101	0	0	0	1	Covered

Code coverage:

Local Instance Coverage Details:

Total Coverage:					86.71%	**77.58%**
Coverage Type	**Bins**	**Hits**	**Misses**	**Weight**	**% Hit**	**Coverage**
Statements	117	98	19	1	83.76%	**83.76%**
Branches	73	66	7	1	90.41%	**90.41%**
FEC Conditions	20	9	11	1	45.00%	**45.00%**
Toggles	204	186	18	1	91.17%	**91.17%**

Output Waveform: