ALU Verification Plan

By Nishanth Gowda CJ EMP ID 6089

LIST OF FIGURES

CHAPTER 1

PROJECT OVERVIEW AND SPECIFICATIONS

1.1 ALU introduction:

The Arithmetic Logic Unit (ALU) acts as the brain of any digital processing system, executing core computational functions with both arithmetic and logical operations. In this project, the ALU is implemented as a flexible, parameterized module, allowing customization of bit-widths to suit different design needs, making it ideal for various embedded or processor-based applications.

This ALU design stand out is its comprehensive functionality , it not only performs basic operations like addition, subtraction, and logical comparisons, but also includes features like bitwise rotation**,** input validation**,** and error detection. Designed for synchronous operation**,** it uses proper clock and reset logic to ensure stable performance in real-time digital system

1.2 Advantages of ALU:

- **Customizable Design**

  With parameterization, the bit-width can be adjusted to meet specific system requirements, enhancing flexibility and reuse.

- **Supports Both Arithmetic and Logical Operations**

  Capable of executing a wide range of tasks, from simple addition to bitwise logical functions, making it a core component in digital systems.

- **Efficient Computation**

  Performs operations quickly, often within a single clock cycle, ensuring fast data processing in real-time applications.

- **Synchronous Operation**

  Proper integration with clock and reset signals ensures timing reliability and predictable behavior in synchronous digital systems.

- **Input Validation and Error Detection**

  Includes mechanisms to check input validity and detect illegal or unexpected conditions, improving the overall robustness of the system.

- **Scalable and Reusable**

  The modular and parameterized nature allows the ALU to be easily reused across different projects with varying complexity.

1.3 Disadvantages of ALU:

- **Limited Instruction Set**

  The ALU can only perform operations that are explicitly defined in the design. Complex functions like floating-point arithmetic require separate units.

- **No Memory Capability**

  The ALU operates purely on inputs and doesn't store data; it must rely on external registers or memory for data storage.

- **Increased Complexity with More Bits**

  As the data width increases (e.g., from 4-bit to 32-bit), the design becomes more complex, potentially affecting performance and power consumption.

- **Fixed Operation Control**

  The set of operations is usually hardcoded; modifying them later requires changes in the HDL code and re-synthesis.

- **Propagation Delay in Certain Operations**

  Some operations, especially those involving carry propagation (like addition or multiplication), can introduce delay depending on how the ALU is implemented.

1.4 Use cases of ALU:

- **Microprocessors and Microcontrollers**

  ALUs are the core computational units in processors. They handle arithmetic calculations, logical decisions, and flag generation for instruction execution.

- **Digital Signal Processing (DSP)**

  ALUs perform operations like addition, multiplication, and shifting — essential in filtering, encoding, and decoding signals. Efficient ALU design enhances real-time processing in audio, video, and communication systems.

- **Embedded Systems**

  Used in smart devices (IoT, sensors, controllers) for decision-making and control logic. Performs fast calculations in real-time automation tasks.

- **Graphics Processing Units (GPUs)**

  Specialized ALUs in GPUs perform parallel arithmetic operations required for rendering graphics, transformations, and shading

- **Cryptographic Algorithms**

  ALUs are used in bitwise logic and modular arithmetic — essential in implementing encryption and decryption logic in secure systems.

- **Control Units in Robotics**

  ALUs assist in decision-making based on sensor inputs and feedback, enabling autonomous movement and intelligent control.

- **Scientific and Engineering Calculations**

High-performance computing applications use powerful ALUs to perform complex mathematical operations with speed and accuracy.

- **Real-Time Operating Systems**

  ALUs work with schedulers and system cores to evaluate timing conditions and trigger tasks based on logic and arithmetic decision

1.5 Project Overview of ALU:

This project involves the design and implementation of a parameterized ALU capable of supporting a wide range of bit-widths and digital operations, making it suitable for both low-end embedded systems and high-performance processors. The verification is done using UVM to ensure functionality, scalability, and reliability across different configurations.

- **Parameterized Design**

  Supports variable bit-widths (e.g., 16, 32, 64, 128 bits). Enables integration into systems

with diverse computational needs, from simple microcontrollers to complex processors.

❖ **Dual Mode Operation**

Operates in two distinct modes:

- Arithmetic Mode (MODE = 1)
- Logical Mode (MODE = 0)

Each mode includes 14 unique operations, selected via a 4-bit command input.

❖ **Supported Operations**

- Arithmetic Mode: ADD, SUB, MUL, INC, DEC, etc.
- Logical Mode: AND, OR, XOR, NAND, NOR, NOT, bitwise rotations, etc.

❖ **Input Control Mechanism**

INP_VALID signal ensures inputs are synchronized, especially important in asynchronous or pipelined systems.

Timeout logic prevents indefinite waiting for missing inputs, improving system responsiveness.

❖ **Comprehensive Status Reporting**

Generates flags for:

- Overflow detection
- Carry-out monitoring
- Comparison results (Equal, Greater Than, Less Than)

Useful for decision-making in CPUs, control units, and FSMs.

❖ **Built-In Error Handling**

Detects and flags invalid operations (e.g., improper bit-rotation commands or out-of-range inputs).

Enhances debuggability and system integration by avoiding silent failures.

❖ **High Reusability and Modularity**

Designed to be easily integrated into larger digital systems.

Modular and scalable for use in both educational projects and industry-level SoC

architectures.

1.6 Design Features:

- **Parameterized Bit-Width Support**
  - Supports multiple data widths: 16-bit, 32-bit, 64-bit, and 128-bit
  - Enhances reusability for low-end to high-end applications
- **Dual Mode Operation**
  - MODE = 1: Arithmetic operations, MODE = 0: Logical operations
  - Clear separation between arithmetic and logic functionality
- **4-bit Command Selection System**
  - Enables up to 16 distinct operations
  - Commands are reused efficiently based on the selected mode
- **Input Validity Handling (INP_VALID)**
  - Ensures operations execute only when valid inputs are received
  - Useful in pipelined or asynchronous environments
- **Timeout Mechanism**
  - Prevents the system from waiting indefinitely for inputs
  - Improves reliability in unpredictable data arrival scenarios
- **Status Flags for Control Logic**
  - Overflow flag
  - Carry-out flag
  - Comparison outputs: Greater than, Less than, Equal
  - Helps CPU or controller units in decision-making (e.g., branching)
- **Error Detection and Handling**
  - Flags invalid operations (e.g., undefined rotation patterns)
  - Improves debugging and integration in larger systems
- **Modular and Scalable Design**
  - Easily extendable for more operations or features
  - Clean separation between datapath and control logic

❖ **Support for Complex Operations**

Includes rotate left/right, shift, AND/OR/XOR, etc.

Enables broader use cases like encryption, CRC, and encoding

❖ **Robust Output Interface**

Generates output data (RESULT)

Sets all relevant status signals on completion

1.7 Design Limitation:

❖ **Fixed Timeout Duration**

The 16-cycle timeout is hardcoded in the design.

Modifying the wait period requires RTL changes, reducing flexibility for integration in different systems.

❖ **Mode-Based Command Overlap**

The same command code performs different operations depending on the mode (arithmetic vs. logical).

Increases the risk of software misconfiguration or unintended operation

❖ **Single Error Flag for Multiple Issues**

Only one ERR signal is used to indicate all types of faults (e.g., timeout, invalid command, input errors).

Debugging becomes difficult as the source of the error cannot be distinguished directly.

❖ **Limited Rotate Operation Range**

Rotate Left and Rotate Right operations support a maximum of 8 positions.

This limits applicability when scaling to wider data paths (e.g., 16-bit or 32-bit systems).

❖ **Fixed Operand Selection on Timeout**

In case of timeout, the ALU automatically chooses the latest operand.

This behavior might not align with expected logic in systems that require previous operand retention.

❖ **Lack of Pipelining**

The ALU executes one operation at a time without pipelining.

This can be a bottleneck in high-speed or parallel processing environments, limiting throughput

1.8 Design diagram with interface signals:

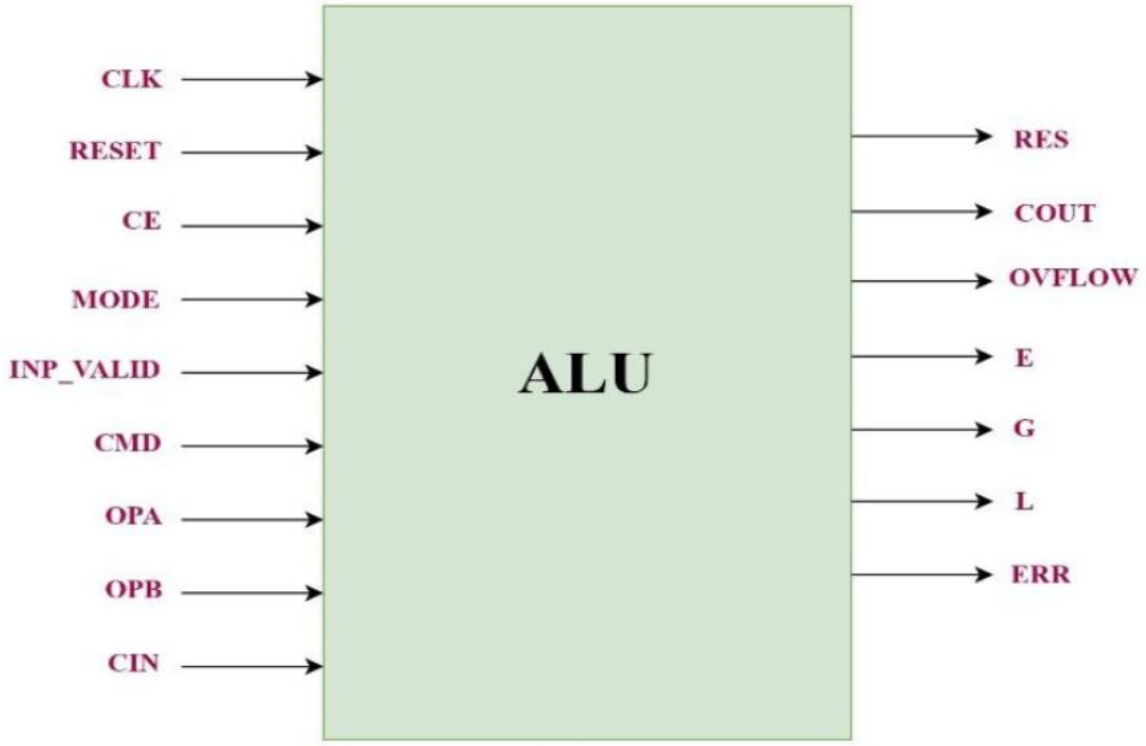


Figure 1.1 ALU block diagram

Input ports:

Signal	Direction	Size(bits)	Description
INP_VALID	INPUT	2	Shows the Validity of the Operands (active high). MSB shows the validity of 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	4	Commands for the Operation
OPA	INPUT	Parameterized	Operand A – first arithmetic/logic operand
OPB	INPUT	Parameterized	Operand B – second arithmetic/logic operand

Signal	Direction	Size(bits)	Description
CIN	INPUT	1	Carry In – input carry for arithmetic operations

Output ports:

Signal	**Direction**	**Size(bits)**	**Description**
ERR	OUTPUT	1	Active High Error Signal
RES	OUTPUT	Parameterized + 1	Result of the instruction performed by the ALU
OFLOW	OUTPUT	1	Overflow – indicates arithmetic overflow condition
COUT	OUTPUT	1	Carry out signal, updated during Addition/Subtraction
G	OUTPUT	1	Comparator output which indicates that the value of OPA is greater than the value of OPB
L	OUTPUT	1	Comparator output which indicates that the value of OPA is less than the value of OPB
E	OUTPUT	1	Comparator output which indicates that the value of OPA is equal to the value of OPB

CHAPTER 2
TESTBENCH ARCHITECTURE AND METHODOLOGY

2.1 Verification Architecture for ALU:

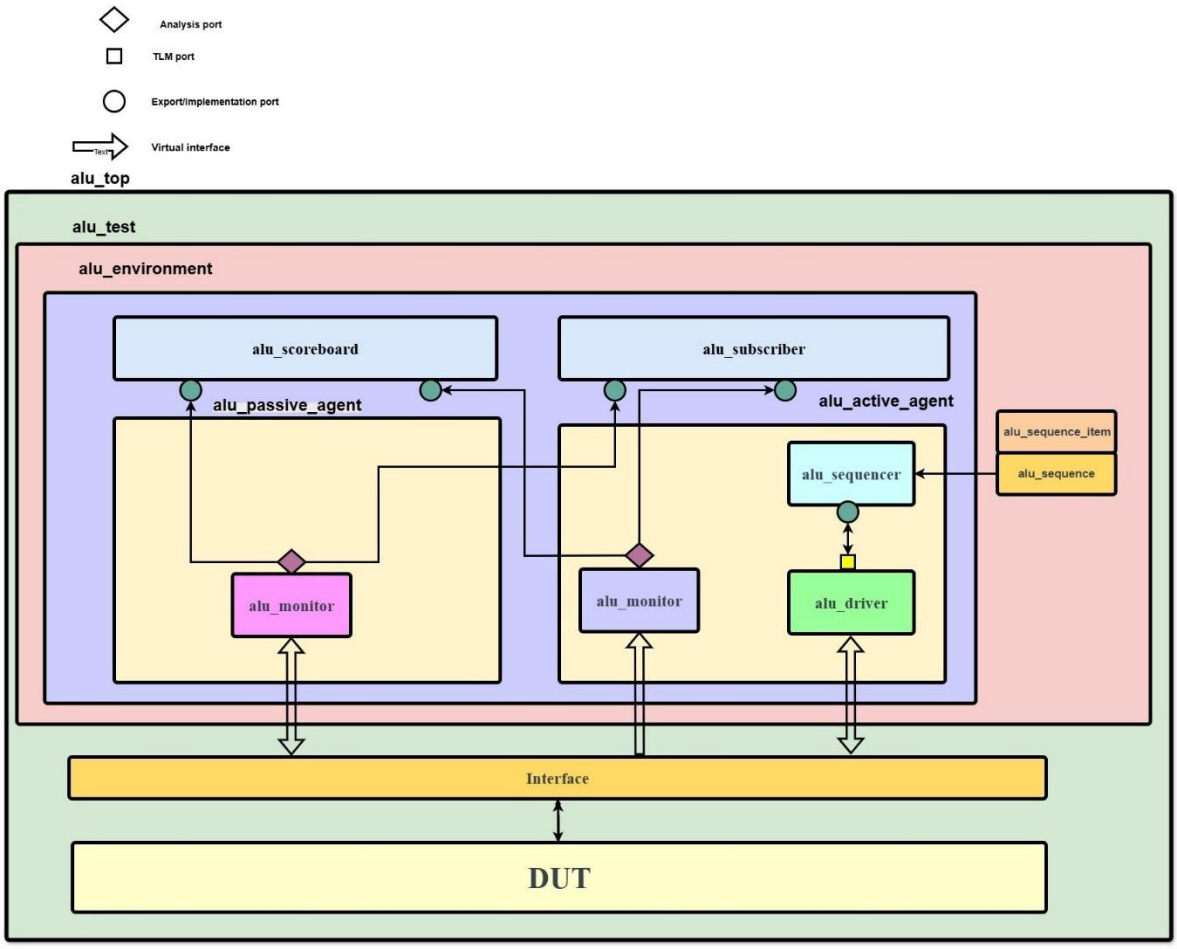


Figure 2.1 :verification architecture for ALU

Key components:

- **Sequence Item:** Defines a transaction with input data and control signals. It is the basic unit of stimulus for the DUT.
- **Sequence:** Generates multiple sequence items. It creates meaningful patterns of stimulus for verification.
- **Sequencer:** Controls the flow of sequence items. It passes transactions from the sequence to the driver.
- **Driver:** Converts transactions into pin-level signals. It drives the DUT through the interface.
- **Monitor:** Observes DUT signals passively. It converts them into transactions for checking.
- **Scoreboard:** Compares DUT output with the expected result. It ensures correctness of functionality.
- **Coverage:** Tracks which scenarios have been tested. It ensures completeness of verification.
- **Agent:** Groups sequencer, driver, and monitor. It acts as a reusable verification component.
- **Environment:** Integrates agents, scoreboard, and coverage. It forms the complete testbench setup.
- **Test:** Configures the environment and runs sequences. It verifies DUT under different scenarios.
- **Top Module:** Connects DUT, interface, and UVM testbench. It acts as the entry point for simulation.
- **Reference Model:** Provides golden output for comparison. It checks DUT's functional correctness.
- **DUV (ALU):** The design under verification. It is tested against functional requirements.
- **Interface:** Connects DUT and testbench. It carries both stimulus and observed signals.

2.2 FLOW CHART OF SV COMPONENTS :

2.2.1 Sequence-Sequencer-Driver

The process begins with the seq_item, which specifies the transaction data, followed by the alu_sequence, which creates these transactions. These transactions are sent to the alu_sequencer, which manages their order and timing. The sequencer then delivers each transaction to the alu_driver through a handshake mechanism to maintain synchronization. Finally, the alu_driver translates the abstract transaction data into pin-level signals and drives them onto the DUT via the interface**.**

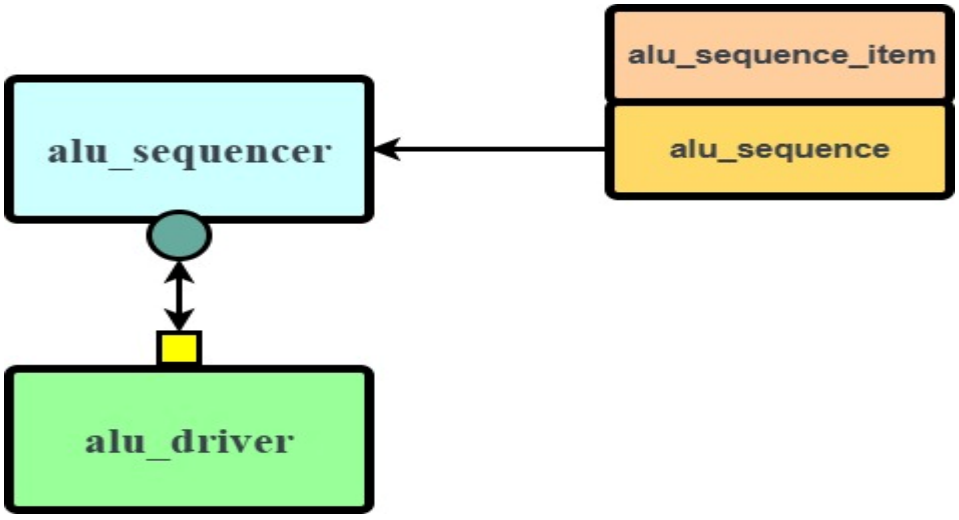


Figure 2.2.1: ALU sequence-sequencer-driver

2.2.2 Monitor-Scoreboard:

At this stage of the verification flow, the alu_monitor keeps track of the DUT interface activity and converts the observed signals into transaction-level data. These transactions are then sent to the alu_scoreboard using its TLM analysis port. The alu_scoreboard, through its analysis implementation port, receives the transactions and applies a reference model internally to compute the expected results for the given inputs. Finally, it compares the DUT's actual outputs (captured by the monitor) with the expected results (from the reference model) to ensure the functionality of the design is correct.

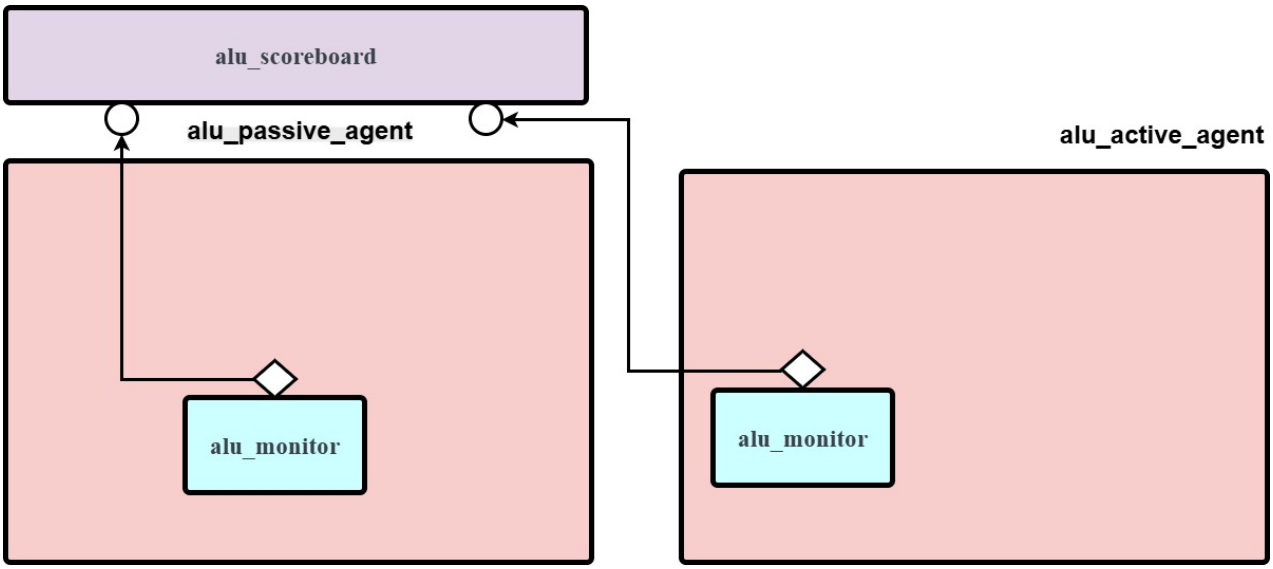


Figure 2.2.2: ALU monitor - scoreboard

`

2.2.3 Coverage

In this stage of the flow, the alu_monitor not only forwards transactions to the scoreboard but also publishes them to the alu_coverage component through its TLM analysis ports. The alu_coverage component receives these transactions via its analysis implementation ports and samples them using its covergroups. This process tracks functional coverage for both input conditions and DUT outputs, ensuring that all relevant scenarios and corner cases are exercised during verification.

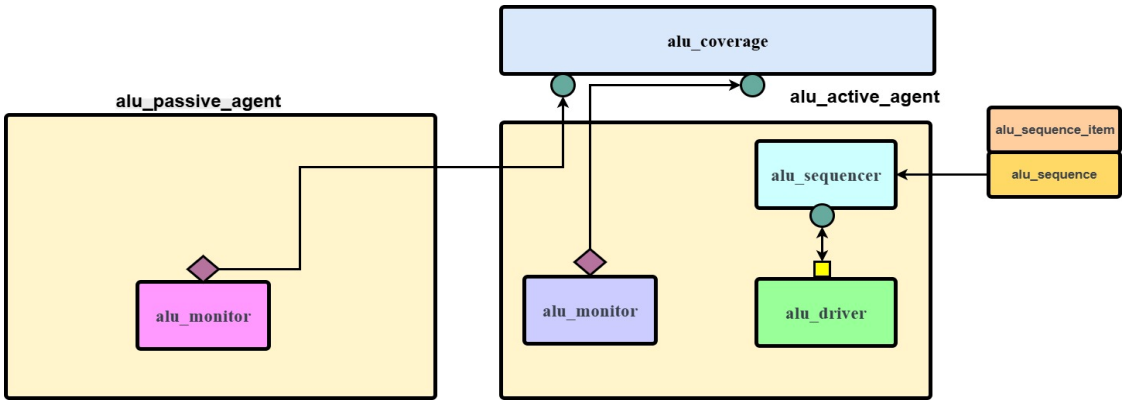


Figure 2.2.3: ALU coverage

2.2.4 ENVIRONMENT COMPONENT

The environment component in UVM serves as the top-level container that brings together all verification components such as agents, scoreboards, and coverage collectors. It establishes the necessary connections between these components through TLM ports, enabling smooth transaction-level communication. The environment is also responsible for configuring, instantiating, and managing its sub-components, ensuring a well-structured verification setup. By integrating all pieces into a reusable framework, the environment provides an organized and scalable approach to verifying the DUT.

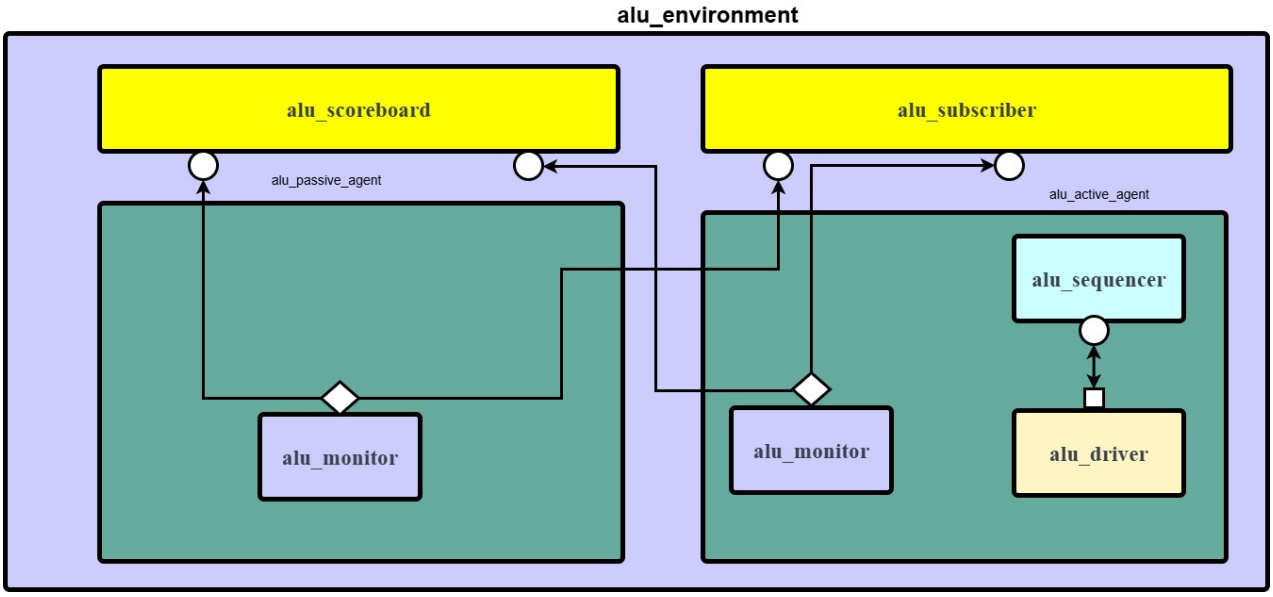


Figure 2.2.4 : ALU environment

2.2.5 TEST COMPONENT

The ALU test serves as the configuration and control layer that instantiates the test environment and defines specific test scenarios. It configures test parameters, initializes the ALU environment with appropriate settings, and controls the stimulus generation through predefined test sequences. The test class acts as the entry point for executing targeted verification scenarios and managing the overall test flow.

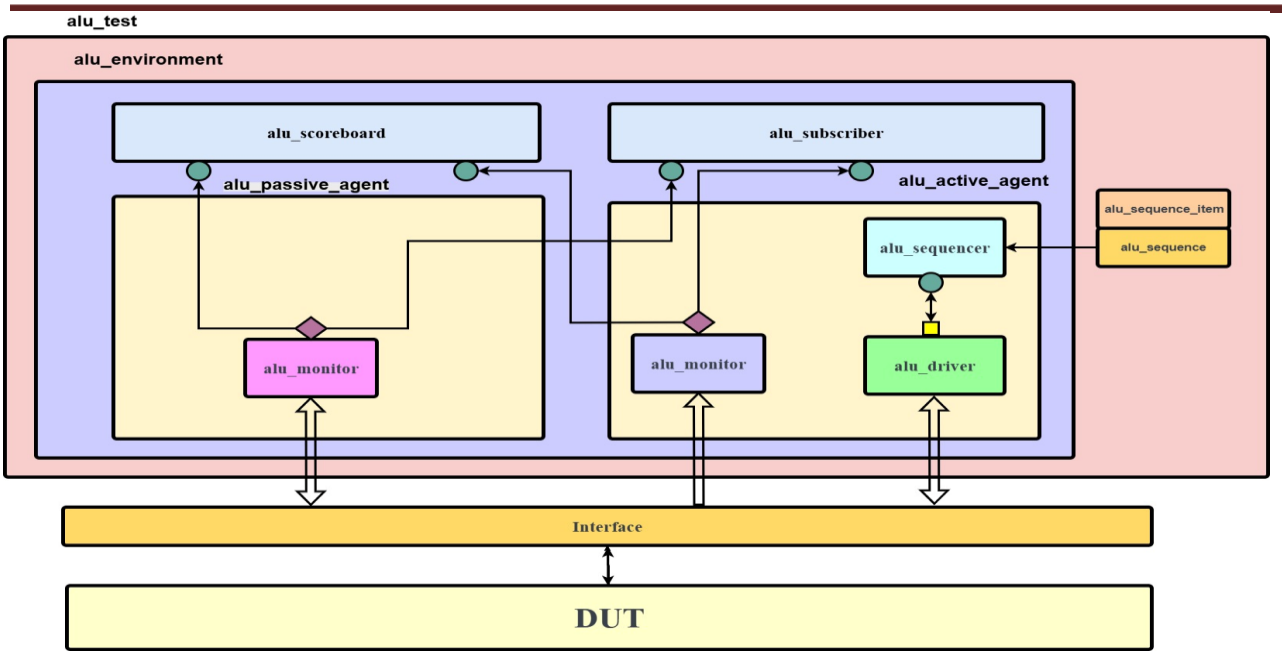


Figure 2.2.5: ALU test component

2.3.6 TOP COMPONENT

alu_top (Outer Box):

•Top-level testbench module that instantiates everything needed to verify the ALU.

•Includes the alu_test and the alu_interface, which connects the design to the testbench.

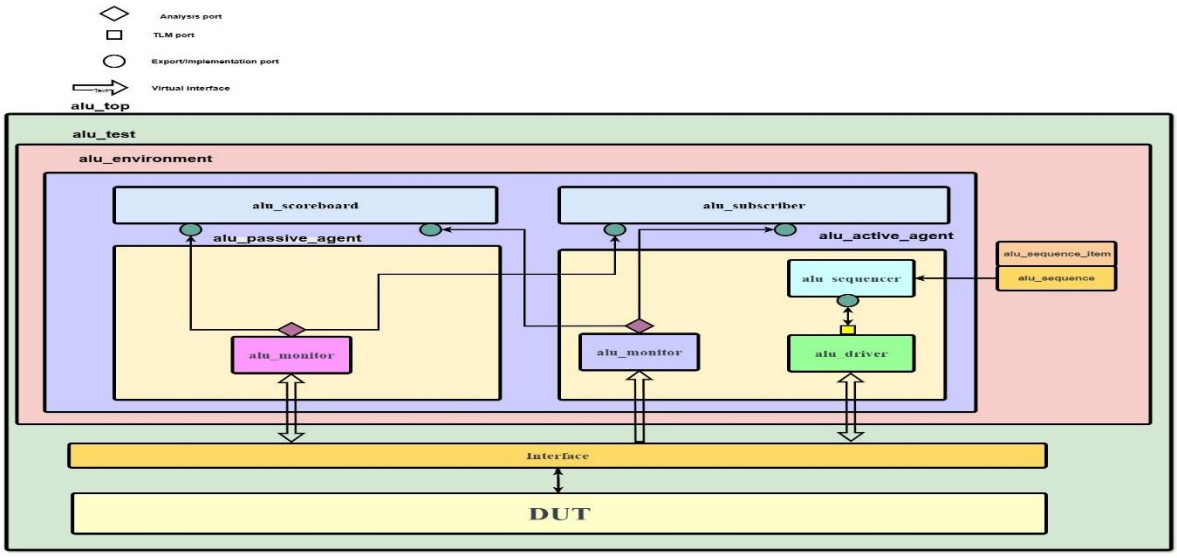


Figure 2.2.6: ALU top component

TEST PLAN

Test plan

https://docs.google.com/spreadsheets/d/1Hb9BDjKUWGku7YgYGu4m2-MxOdKwIwqMFIhu1x-F5Y4/edit?gid=0#gid=0

coverage plan

https://docs.google.com/spreadsheets/d/1Hb9BDjKUWGku7YgYGu4m2-MxOdKwIwqMFIhu1x-F5Y4/edit?gid=820551267#gid=820551267

assertion plan

https://docs.google.com/spreadsheets/d/1Hb9BDjKUWGku7YgYGu4m2-MxOdKwIwqMFIhu1x-F5Y4/edit?gid=1490673837#gid=1490673837

CHAPTER 3

VERIFICATION RESULTS AND ANALYSIS

3.1 ERRORS IN THE DUT

specification	Bugs description
ADD_IN	In ADD_IN operation, when OPA = OPB and CIN = 1, Then,there is no COUT in this condition.
SUB_IN	In SUB_IN operation, when OPA = OPB and CIN = 1, result is -1. There is no overflow in this condition.
INC_A	In INC_A operation (CMD = 4'b0100), RES = OPA is assigned without increment.This is a bug , it should be RES = OPA + 1 to perform increment correctly.
INC_B	As per spec,CMD 6 is INC_B but in design it is DEC_B
DEC_B	As per spec,CMD 7 is DEC_B but in design it is INC_B
OR	As per spec,CMD 2 is OR operation,but in design it is logical AND operation
SHR1_A	As per spec ,CMD=8 ,MODE=0, shift right operation,but in design its res=opa
SHR1_B	As per spec ,CMD=10 ,MODE=0, shift right operation,but in design it's a left shift
ROR	In CMD = 4'b1101, when any of oprd2[4] to oprd2[7] are high, ERR is set to 0.This is a bug — as per specification, ERR should be set to 1 to indicate error
ADD_IN	In ADD_IN operation, the design performs addition but does not assign COUT. This is a bug — COUT should reflect the carry-out from the MSB of the sum.
MUL_S	In CMD = 4'b1010, the design performs RES = (oprd1 << 1) - oprd2, which is incorrect. It should perform multiplication — RES = (oprd1 << 1) * oprd2 as per specification.

INP_VALID	When INP_VALID = 2'b00, no operation should occur and ERR should be set to 1. In the current design, ERR is not asserted, which violates input validity
CLK WAITING	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 CODE COVERAGE

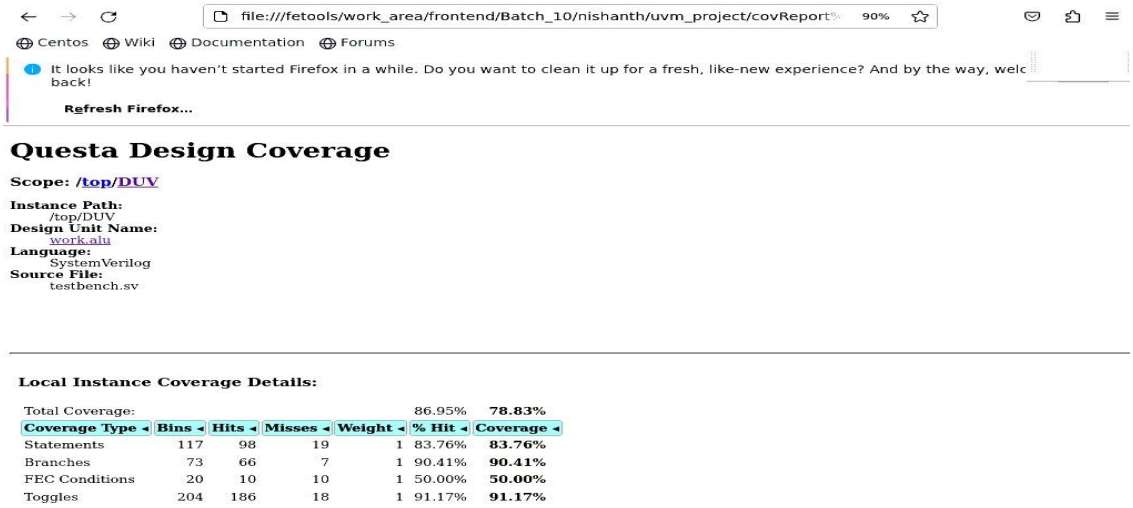


Figure 3.2: code coverage

3.3 INPUT FUNCTIONAL COVERAGE

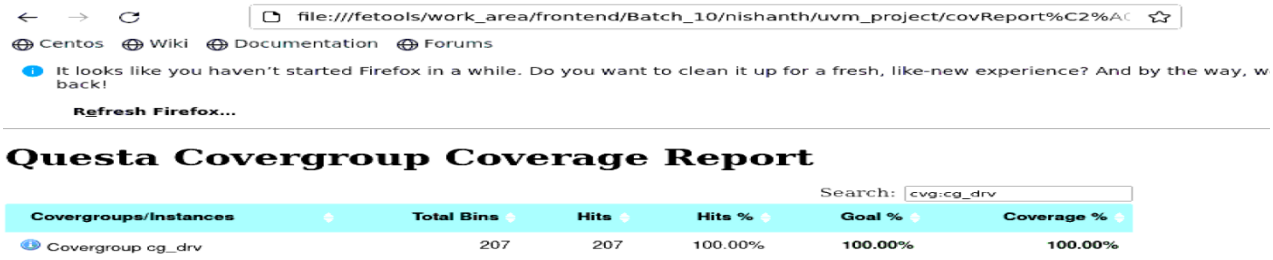


Figure 3.3: input functional coverage

3.4 OUTPUT FUNCTIONAL COVERAGE

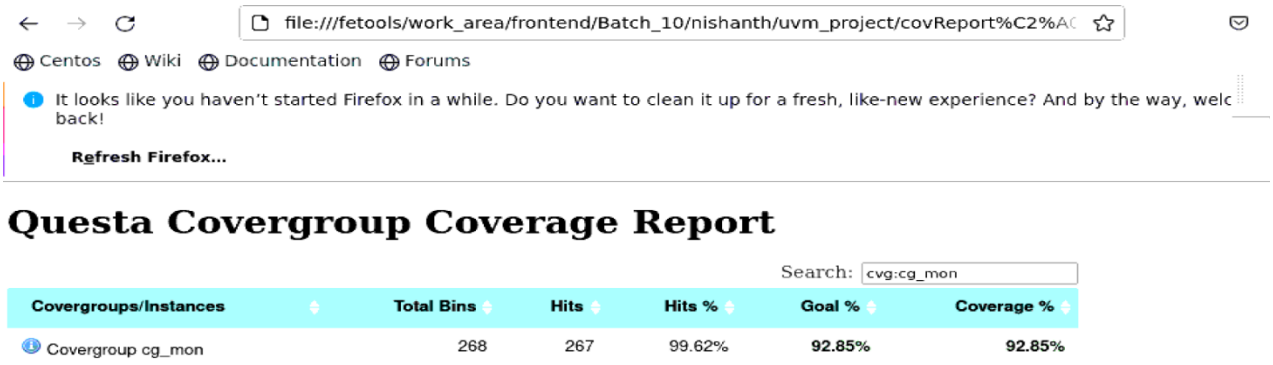


Figure 3.4: output functional coverage

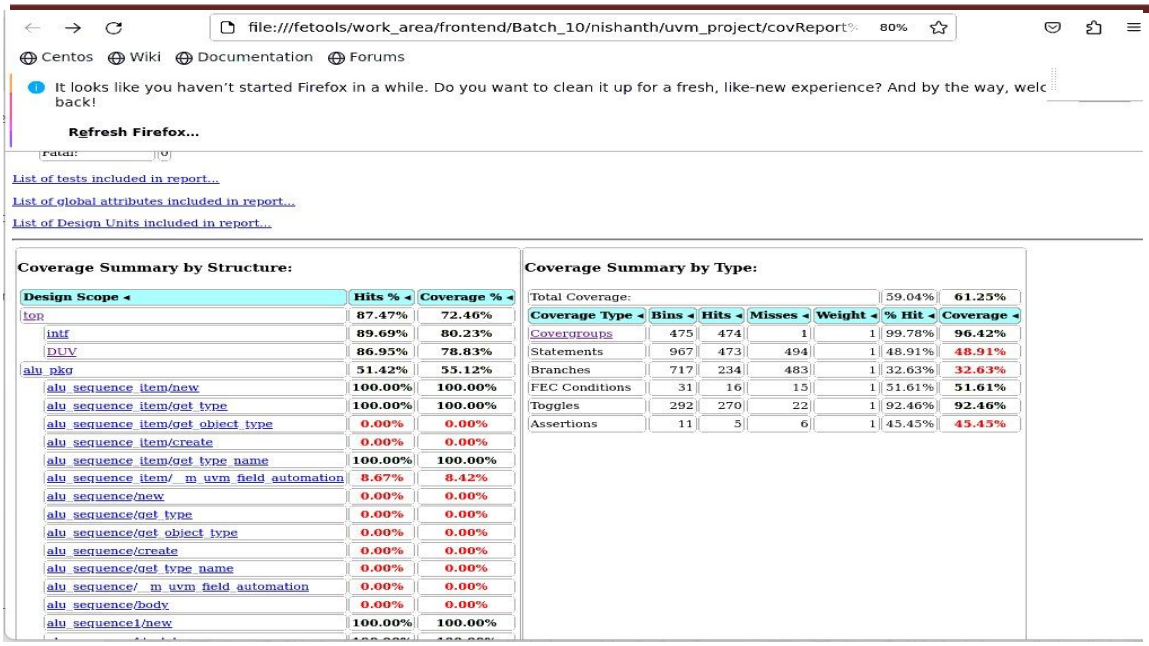
3.4 ASSERTION COVERAGE

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

3.5 OVERALL COVERAGE



file:///fetools/work_area/frontend/Batch_10/nishanth/uvm_project/covReport 80%

Centos Wiki Documentation Forums

It looks like you haven't started Firefox in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welc back!

Refresh Firefox...

List of tests included in report...

List of global attributes included in report...

List of Design Units included in report...

Coverage Summary by Structure:

Design Scope	Hits %	Coverage %
top	87.47%	72.46%
intf	89.69%	80.23%
DUV	86.95%	78.83%
alu_pkg	51.42%	55.12%
alu_sequence_item/new	100.00%	100.00%
alu_sequence_item/get_type	100.00%	100.00%
alu_sequence_item/get_object_type	0.00%	0.00%
alu_sequence_item/create	0.00%	0.00%
alu_sequence_item/get_type_name	100.00%	100.00%
alu_sequence_item/__m_uvm_field_automation	8.67%	8.42%
alu_sequence/new	0.00%	0.00%
alu_sequence/get_type	0.00%	0.00%
alu_sequence/get_object_type	0.00%	0.00%
alu_sequence/create	0.00%	0.00%
alu_sequence/get_type_name	0.00%	0.00%
alu_sequence/__m_uvm_field_automation	0.00%	0.00%
alu_sequence/body	0.00%	0.00%
alu_sequence1/new	100.00%	100.00%

Coverage Summary by Type:

Total Coverage:					59.04%	61.25%
Coverage Type	Bins	Hits	Misses	Weight	% Hit	Coverage
Covergroups	475	474	1	1	99.78%	96.42%
Statements	967	473	494	1	48.91%	48.91%
Branches	717	234	483	1	32.63%	32.63%
FEC Conditions	31	16	15	1	51.61%	51.61%
Toggles	292	270	22	1	92.46%	92.46%
Assertions	11	5	6	1	45.45%	45.45%

Figure 3.5: overall coverage

3.6 OUTPUT WAVEFORM

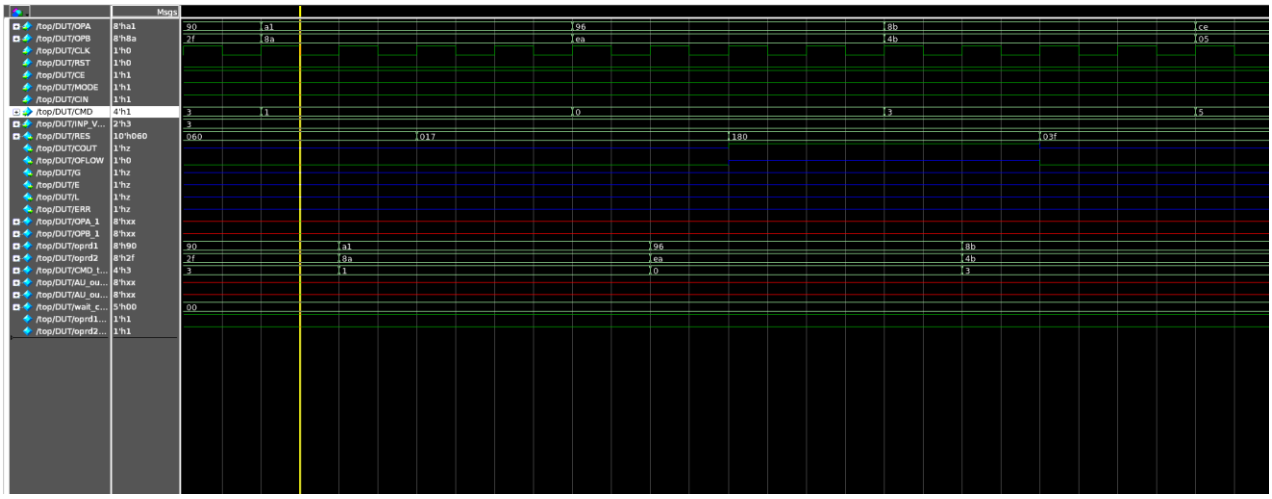


Figure 3.6: output waveform