ALU Verification Plan

VERIFICATION DOCUMENT- ALU

CHAPTER 1 – DESIGN OVERVIEW

1.1 ALU introduction:

The Arithmetic Logic Unit (ALU) is the core component responsible for performing all arithmetic and logical operations in a digital system. You can think of it as the brain of a processor capable of making calculations and decisions.

What makes this ALU design unique is its flexibility and versatility. It's parameterized, meaning its bit-width can be customized based on the specific requirements of different applications — whether 8-bit, 16-bit, or more.

This ALU isn't limited to simple operations like addition and subtraction. It supports a wide range of functions, including bitwise rotations, comparisons, and even error checking and input validation to ensure reliable operation.

The entire design is synchronous, working in step with a clock signal and reset control. This ensures predictable, stable behavior making it well-suited for integration into modern, high-performance digital systems.

1.2 Advantages of ALU:

- **Flexible bit width support**
  The design is scalable it can start as a 16-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), enabling everything from basic math to advanced bit manipulations like rotation reducing the need for additional external processing units.

- **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), and error indicators providing the system with rich feedback for better decision-making.

- **Power Efficiency:**
  Through a clock enable (CE**)** feature, the ALU can be disabled when idle. This minimizes power consumption, which is especially beneficial in battery-powered or energy-sensitive applications.

- **Built-in Error Detection:**
  The ALU is capable of catching invalid operation scenarios, particularly during rotate operations where improper input formats can lead to malfunction. This built-in check helps prevent such issues

1.3 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

- **Hardware Overhead:**
  While feature-rich, the ALU's complexity means it consumes more logic resources. For applications with simple requirements, this design might be unnecessarily large and inefficient

- **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

1.4 Use cases of ALU:

- **Arithmetic Operations**
  Handles essential arithmetic functions such as addition, subtraction, multiplication, and division. These are fundamental for tasks like incrementing counters, calculating memory addresses, and performing general computations in software or hardware.

- **Logical Operations**

Executes standard logic operations like AND, OR, XOR, and NOT. These are widely used for value comparisons, implementing decision-making structures, 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.

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

- **Checksum and CRC Computation**
  Can be used to compute checksums or cyclic redundancy checks (CRC), which are widely applied in error detection for communication protocols and data transmission systems.

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

- **Carry and Overflow Detection:**
  Monitors for carry-out or arithmetic overflow, ensuring proper handling of signed and unsigned data and maintaining operation accuracy during calculations.

- **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 aims to design a flexible and high-performance Arithmetic Logic Unit (ALU) suitable for a variety of digital systems, ranging from simple embedded devices to complex processors. 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. This scalability makes it adaptable to both low-resource and high-performance environments. 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. Arithnetic mode supports 12 operations and Logical mode supports 15 operations covering everything from basic arithmetic like addition, subtraction, and multiplication to logical operations like AND, OR, XOR, and advanced functions such as bit rotations. The dual-mode system efficiently leverages a 4-bit command input, which ensures logical separation and efficient decoding of operations.

To meet the demands of real-world system integration, the ALU includes intelligent control features. 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. These flags allow external systems like control units or processors to make decisions based on the outcomes of ALU operations, such as branching or exception handling.

Another highlight of this design is its robust error-handling capability. For operations like bit rotations, where operand B must follow strict rules, the ALU is equipped to detect and flag invalid inputs instead of failing silently. This proactive error detection simplifies debugging and integration, especially in complex systems. Overall, the project focuses on creating a reusable, scalable, and reliable ALU design that balances computational power with smart system-level features, making it highly suitable for modern digital applications.

1.6 Design Features:

- **Synchronous Operation with Asynchronous Reset**

  Triggers on the rising edge of CLK and can reset immediately via an async reset — ideal for reliable start-up and emergency recovery.

- **Flexible Bit-Width via Parameterization**

  Data width is set by a parameter (16, 32, 64, 128 bits, etc.) so you can scale the ALU to your application without touching the core logic.

- **Smart Operand Control**

  A 2-bit INP_VALID input indicates whether none, one, or both operands are ready perfect for handling staggered inputs in pipelined or asynchronous designs.

- **Advanced Rotate Operations**

  Supports variable rotate left/right based on operand B. Includes error checks to catch bad input values.

- **Comprehensive Comparison Outputs**

  Simultaneously provides Greater (G), Less (L), and Equal (E) flags to speed up control-logic decisions.

- **Built-in Overflow Detection**

  Automatically detects overflow in arithmetic operations to help avoid errors in calculations.

- **Power-Saving Clock Enable (CE)**

  An optional CE input lets you gate the clock and shut down the ALU when idle, saving power in low-energy 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-Dependent Command Codes**

  Same command code does different things in arithmetic and logical modes — this can lead to software mistakes.

- **Single Error Signal (ERR)**

  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.

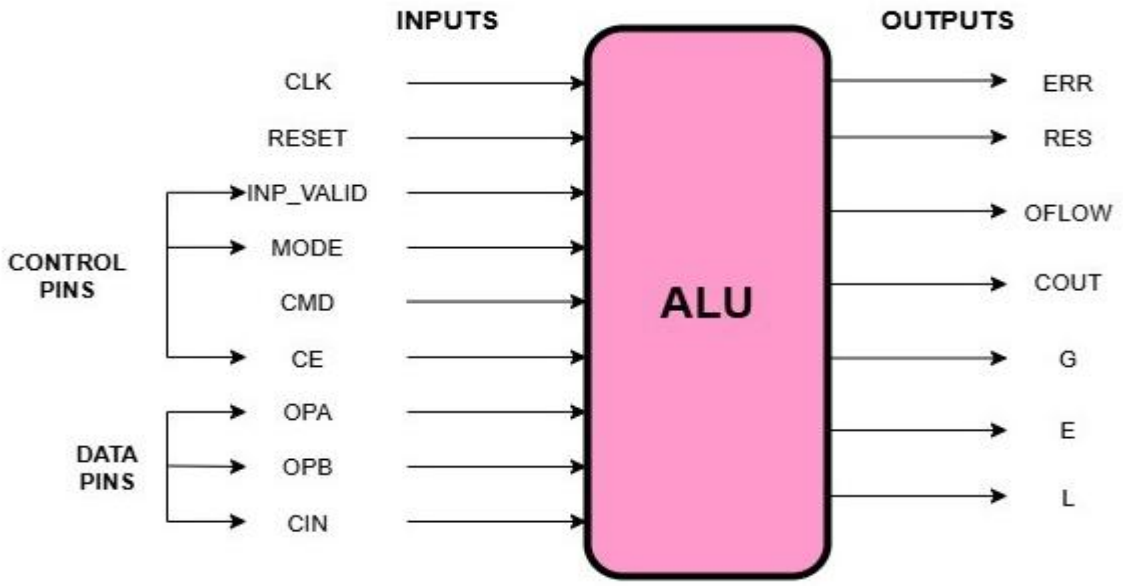
- **No Pipelining**

  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:



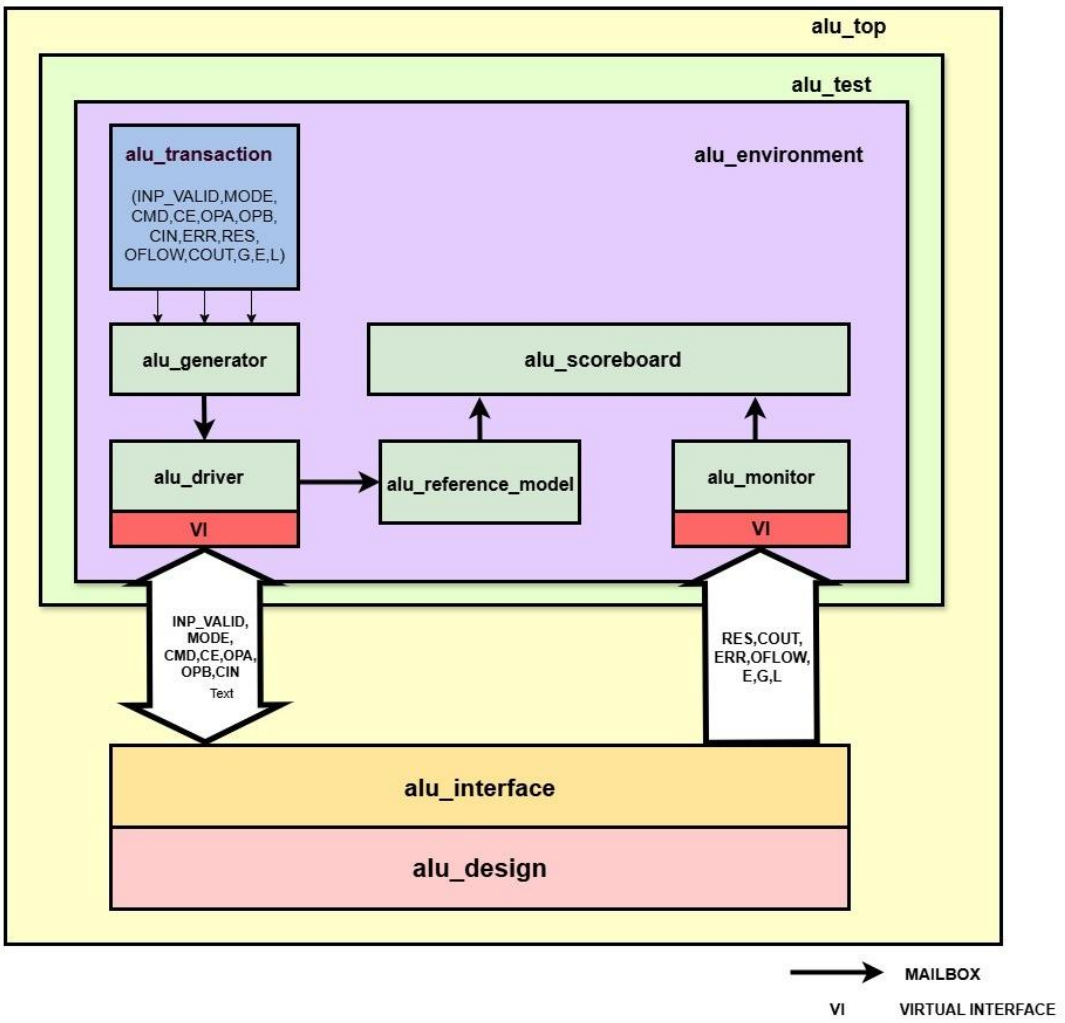
Signal Name	Type	Description
INP_VALID	INPUT	Input valid signal - indicates when input data is valid
MODE	INPUT	Mode selection signal - determines ALU operation mode
CMD	INPUT	Command signal - specifies the specific ALU operation
OPA	INPUT	Operand A - first arithmetic/logic operand
OPB	INPUT	Operand B - second arithmetic/logic operand
CIN	INPUT	Carry In - input carry for arithmetic operations
ERR	OUTPUT	Flags errors (e.g., invalid operations or shift values).
RES	OUTPUT	Final result of the ALU operation.

OFLOW	OUTPUT	Indicates arithmetic overflow.
COUT	OUTPUT	Carry-out from arithmetic operations.
G	OUTPUT	Greater than flag for comparision
E	OUTPUT	Equality flag for comparision
L	OUTPUT	Lesser than flag for comparision

CHAPTER 2 - Verification Architecture

2.1 Verification Architecture :

2.2 Verification Architecture for ALU:



The figure illustrates the ALU testbench architecture used for verifying the functionality of the Arithmetic Logic Unit (ALU) design. This layered structure promotes modularity and clarity in verification. At the bottom of the stack is the ALU Design (DUT), interfaced via a well-defined alu_interface, which acts as a bridge between the DUT and the testbench.

Above the DUT lies the alu_environment, which encapsulates all verification components. The process begins with the alu_transaction, which defines the structure of all data fields required for testing including inputs (INP_VALID, MODE, CMD, CE, OPA, OPB, CIN) and expected outputs (RES, ERR, OFLOW, COUT, G, E, L).

The alu_generator creates randomized or directed transactions and passes them to the alu_driver, which then drives these values into the DUT using a Virtual Interface (VI). This abstraction allows reusable and scalable connection to the ALU.

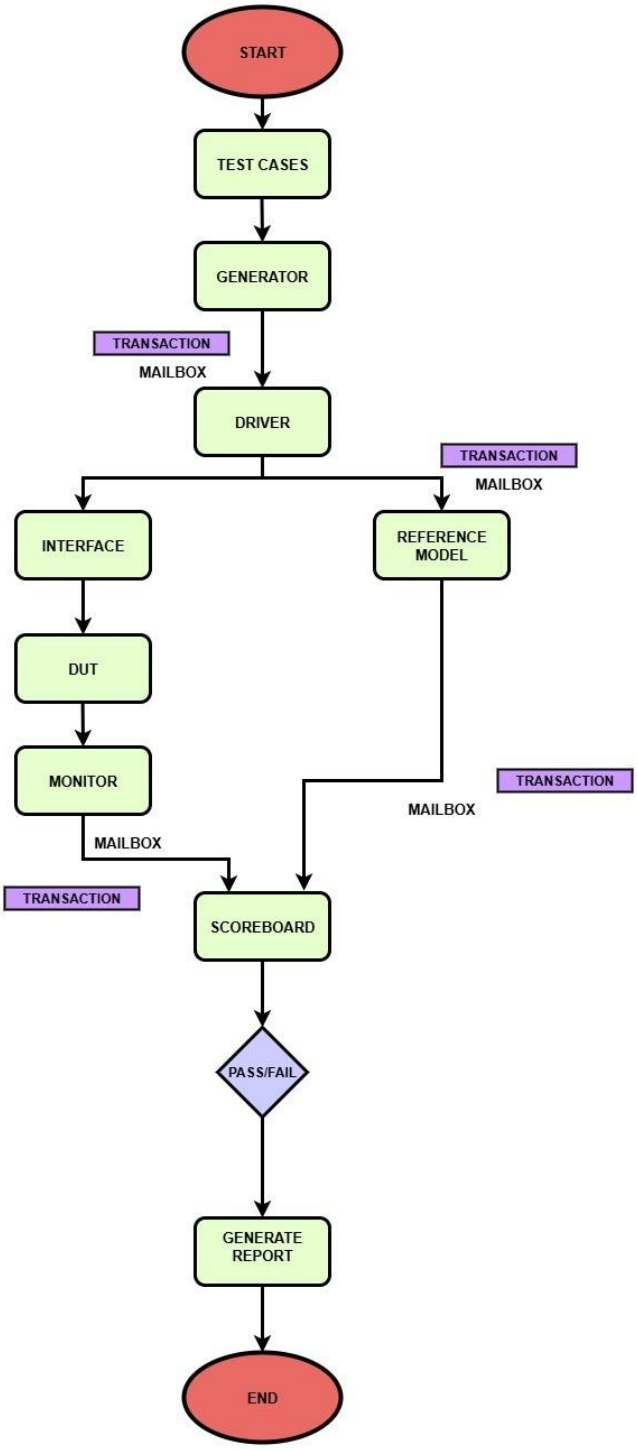
Simultaneously, the same transaction is sent to the alu_reference_model, which computes the expected output based on known functional behavior. As the DUT processes the input, its outputs are captured by the alu_monitor again through the VI and converted into transaction format.

Both the reference model's expected outputs and the DUT's actual outputs are sent to the alu_scoreboard, which performs a comparison to validate correctness. Any mismatches are flagged, allowing developers to trace issues back to specific operations or commands.

The entire testbench resides within the alu_test and alu_top layers, forming a complete self-contained verification system. Mailboxes and virtual interfaces ensure clean communication and data handling between components.

This architecture ensures that every component from stimulus generation to output checking — works together in a synchronized and reliable manner to verify the ALU thoroughly and systematically.

2.3 FLOW CHART OF SV COMPONENTS :



The SystemVerilog Verification Flow shown in the diagram represents a modular and layered testbench architecture commonly used in functional verification of digital designs, such as an ALU or any other DUT (Design Under Test).

The process begins at the START, where Test Cases define the specific input conditions and scenarios to be tested covering various corner cases, valid and invalid operations.The Generator module is responsible for creating random or directed transactions (stimulus), which encapsulate all necessary input data. These transactions are passed via a Mailbox to the Driver, which interprets them and drives corresponding low-level signals to the DUT.

The Driver communicates with the DUT (Design Under Test) through an Interface, which connects the verification environment to the DUT's ports. Simultaneously, the same transaction is sent to a Reference Model, which simulates expected behavior independently.

As the DUT processes the inputs, its outputs are captured by a Monitor, which translates the DUT’s raw outputs into meaningful transaction-level data. These monitored outputs are sent to the Scoreboard, also via a Mailbox.

The Scoreboard compares the DUT output against the Reference Model's output to detect mismatches or failures. Based on this comparison, a PASS/FAIL decision is made.

After this evaluation, the testbench moves to the Generate Report phase, where test results are compiled including information about passed tests, failed cases, and any detected issues.

The process concludes at the END block, signaling the completion of the test session.
Throughout this flow:

- Mailboxes ensure synchronized and orderly communication between blocks.
- Transactions abstract signal-level activity into high-level operations for easier generation, monitoring, and analysis.

This structured methodology ensures modularity, reusability, and thorough coverage, making it ideal for verifying complex digital systems with high reliability.