ALU Verification Plan

VERIFICATION DOCUMENT- ALU

CONTENTS **Page Number**

CHAPTER 1 – DESIGN OVERVIEW

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

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.

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.

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

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.

- **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.

Design Features:

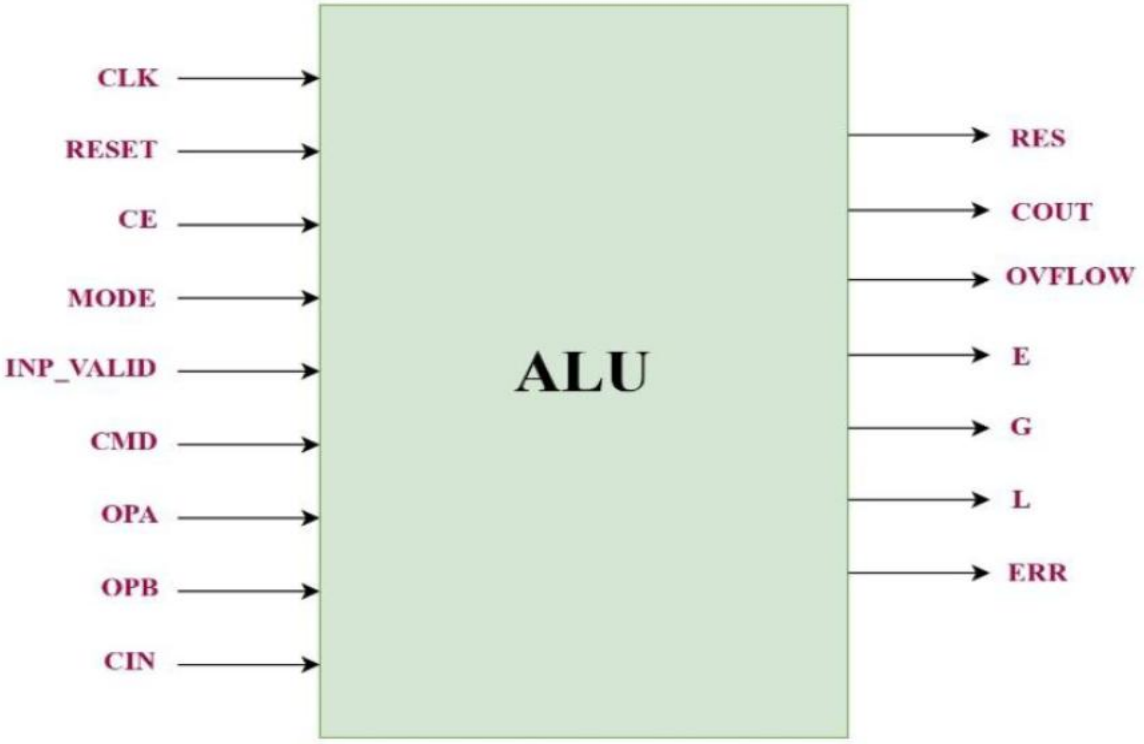
- **Parameterized Bit-Width Support**
  - Supports multiple data widths: 16-bit, 32-bit, 64-bit, and 128-bit
  - Enhances reusability for low-end to high-end applications
- **Dual Mode Operation**
  - MODE = 1: Arithmetic operations, MODE = 0: Logical operations
  - Clear separation between arithmetic 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

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

Design diagram with interface signals:



Input ports:

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

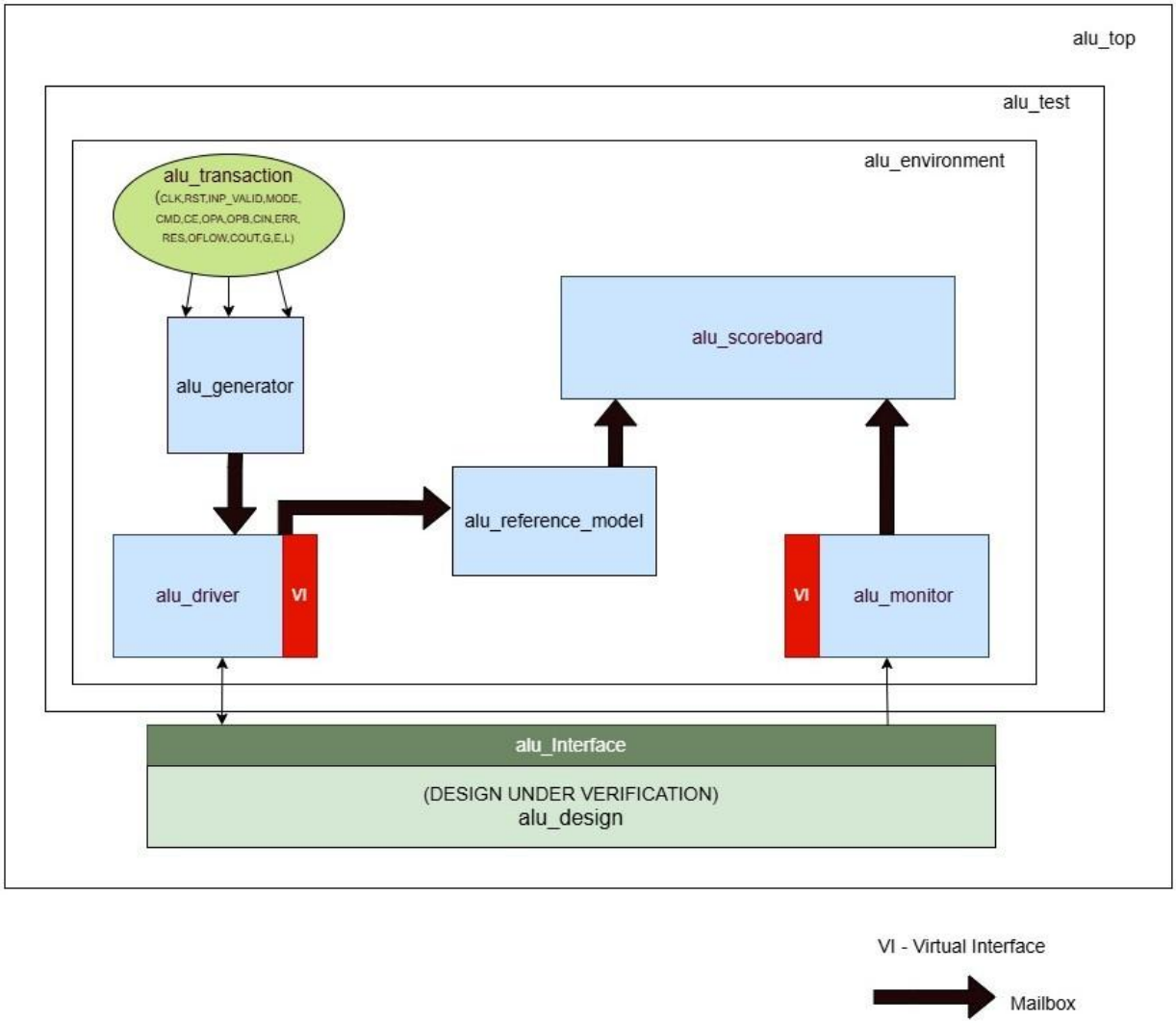
Output ports:

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:



This diagram represents the SystemVerilog testbench architecture for an ALU (Arithmetic Logic Unit) design. It is a modular verification environment, structured to enable scalable and reusable testing of the ALU using transaction-level modeling (TLM) and mailboxes for communication.

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.

alu_test

- Test configuration layer.
- Instantiates and configures the test environment (alu_environment) and provides stimulus via the test class.

alu_environment

This is the heart of the testbench. It contains generator, driver, monitor, scoreboard, and reference model:

alu_transaction

- A class that defines the transaction or stimulus data structure.
- Contains all signal fields like:
    - CLK, RST, INP_VALID, MODE
    - CMD, CE, OP_A, OP_B, CIN, ERR
    - RES_OFLOW, COUT, G, E, L
- These represent the inputs/outputs used to drive the DUT.

alu_generator

- Generates randomized or constrained stimulus (transaction objects).
- Sends transactions to alu_driver using mailbox.

alu_driver

- Converts high-level transactions into pin-level signals.
- Drives those onto the DUT via the Virtual Interface (VI).

- Acts like a proxy between transaction world and DUT signal world.

alu_monitor

- Monitors outputs from the DUT through the Virtual Interface (VI).
- Converts low-level DUT output signals back into high-level transaction objects.
- Sends these to the alu_scoreboard for checking.

alu_reference_model

- A golden model of the ALU that predicts the expected output.
- Receives the same transaction as the DUT (from generator).
- Sends the predicted output to the alu_scoreboard.

alu_scoreboard

- Compares outputs from the DUT (via alu_monitor) and the reference model.
- Logs mismatches or confirms correctness.
- Verifies functional accuracy.

alu_interface

- A SystemVerilog interface that connects DUT ports to the testbench.
- Enables access to DUT signals in a clean and controlled way.
- The VI (Virtual Interface) is passed to the driver and monitor.

alu_design (DUT - Design Under Verification)

- This is your actual RTL design — the ALU module that you want to verify.
- Driven and observed using alu_interface.

Communication

- Arrows in the diagram represent mailbox-based communication between components:
    - generator → driver
    - monitor → scoreboard
    - reference model → scoreboard

2.3 FLOW CHART OF SV COMPONENTS :