ALU Verification Plan

VERIFICATION DOCUMENT- ALU

CHAPTER 1 – DESIGN OVERVIEW

1.ALU:

The Arithmetic Logic Unit is at the core of digital systems and is able to handle multiple arithmetic or logical operations simultaneously. This project centers around the design and verification of a parameterized synchronous ALU supporting both arithmetic and logic functions with configurable bit widths in 16, 32, 64, and 128 bits. The ALU has two inputs with one control interface containing a mode select and a 4-bit opcode to select one of the operations.

It performs unsigned addition, subtraction, increment, decrement, comparison, and several types of multiplication, as well as logical operations including AND, OR, XOR, NOT, and their respective complementary operations: NAND, NOR, and XNOR. Furthermore, it supports shift and rotate operations of both directions with variable shift amounts.

To make the ALU both reliable and efficient, the design includes several advanced features. It supports synchronous operation with clock enable, allows for an asynchronous reset, and includes built-in mechanisms to detect errors during operation. One key feature is a 16-cycle timeout system that keeps track of how long the ALU waits for input operands. If the operands don’t arrive in time, the system automatically uses the most recent values and raises an error signal, helping to prevent unexpected behavior.

Overall, this ALU is built to be flexible and scalable, making it easy to integrate into larger digital systems like embedded platforms, custom processors, or other hardware projects that require a reliable computation unit.

1.2 Advantages of ALU:

- **Scalable Architecture:**
  The ALU is designed to scale effortlessly across different bit-widths — whether it's 8, 16, 32, 64 bits, or more — without needing to modify the core logic. This saves development time and reduces the risk of introducing new bugs.

- **Extensive Operation Support:**
  With 11 arithmetic and 14 logical operations built-in, the ALU handles everything from basic addition to advanced functions like rotate. This reduces the need for extra components or processing elsewhere.

- **Smart Input Management:**
  The ALU uses an INP_VALID signal to manage inputs that arrive at different times. A built-in timeout mechanism ensures the system doesn't hang if inputs are delayed or missing.

- **Informative Status Flags:**
  After each operation, the ALU provides useful flags — such as carry, overflow, greater-than, less-than, equal, and error — giving the system helpful insights for decision-making.

- **Power Efficiency:**
  A clock enable (CE) feature allows the ALU to shut down when not in use, making it ideal for low-power or battery-operated devices.

- **Built-in Error Detection:**
  The design actively detects invalid operations, especially in cases like rotate, where incorrect inputs can lead to unexpected behavior — helping ensure robust and predictable performance.

1.3 Disadvantages of ALU:

- **Fixed Timeout Limit:**
  The ALU always waits exactly 16 clock cycles for input operands—no more, no less. This rigid timeout might be too long for fast systems or too short for slower ones, and unfortunately, it’s not configurable.

- **Dual-Purpose Commands Can Be Confusing:**
  The same operation code (CMD) can mean different things depending on the mode. For example, CMD = 0 might trigger ADD in arithmetic mode but AND in logic mode. If you're not paying close attention, it's easy to make mistakes.

- **Generic Error Flag:**
  There's only one error signal, no matter what goes wrong. Whether it's a timeout, an invalid command, or a wrong input format — you won't know the exact cause. This makes debugging and fixing issues more difficult.

- **Overdesigned for Simple Use-Cases:**
  The ALU is packed with features, which is great for complex systems — but it comes at the cost of increased hardware usage. For applications that only need basic arithmetic or logic, this ALU might be more than necessary.

- **Rotate Operation Requires Careful Inputs:**
  The rotate-left and rotate-right commands have strict input requirements. For instance, certain bits in operand B must be zero (like bits [7:4]). If the software doesn't handle this properly, it can easily cause errors during rotation.

1.4 Use cases of ALU:

- **Arithmetic Operations:**
  The ALU handles essential math tasks like addition, subtraction, multiplication, and division. These operations are vital for things like updating counters, calculating memory addresses, or performing basic computations in software.

- **Logical Operations:**
  It performs standard bitwise logic functions such as AND, OR, XOR, and NOT. These are widely used in tasks like comparing values, bit masking, and making logical decisions in digital circuits and processors.

- **Bit Shifting and Rotation:**
  The ALU can shift bits left or right or rotate bits—operations that are common in encryption, data encoding/decoding, and low-level data manipulation.

- **Comparisons:**
  It can compare two values to determine if one is greater than, less than, or equal to the other. These results are crucial for implementing if-else conditions, loops, and branching decisions in both hardware and software.

- **Address Calculations:**
  Used in memory operations, the ALU can compute the next instruction address or the location of data, enabling smooth control flow in processors.

- **Checksum and CRC Computation:**
  The ALU supports operations that help compute checksums and cyclic redundancy checks (CRC)—important tools for error detection in data communication systems.

- **Control Signal Generation:**
  Based on comparison results, the ALU can help generate control signals for Finite State Machines (FSMs) and other control logic in digital systems.

- **Overflow and Carry Detection:**
  During arithmetic operations, the ALU can detect overflow and carry, which is crucial for handling signed and unsigned numbers correctly and ensuring computational accuracy.

- **CPU Instruction Execution:**
  As the core of the execution stage in a CPU, the ALU is responsible for carrying out the actual computations defined by machine instructions — making it a fundamental part of any processing unit.

1.5 Project Overview of ALU:

This project centers around designing a versatile and powerful Arithmetic Logic Unit (ALU) that's suitable for a wide range of digital systems — from simple embedded controllers to high-performance processors. The standout feature of this ALU is its parameterized architecture, which means it can be easily configured to work with different bit-widths like 16, 32, 64, or even 128 bits, depending on the system's needs. This flexibility makes it ideal for both resource-constrained devices and compute-intensive applications.The ALU takes in two input operands and supports two operating modes: arithmetic mode for operations like addition, subtraction, and multiplication, and logical mode for functions such as AND, OR, and XOR. A compact 4-bit command interface is used to select among 14 supported operations in each mode, making the ALU both powerful and easy to control.

The architecture is designed to handle asynchronous or staggered inputs through an input-valid signal, ensuring smooth data flow even when inputs arrive at different times. A built-in 16-cycle timeout mechanism prevents indefinite stalling by signaling an error if inputs do not arrive in time, which enhances overall system reliability. The ALU also generates useful status signals such as overflow, carry, and comparison results (greater than, less than, or equal), allowing external systems like CPUs or control units to make real-time decisions based on operation outcomes. Advanced error detection features are included to catch invalid scenarios, especially during complex operations like bit rotations, where incorrect operand formats could otherwise lead to silent failures. By flagging these issues, the ALU simplifies debugging and ensures safer integration into larger systems. Overall, the design balances performance, scalability, and robustness, making it a reliable component for modern digital hardware.

1.6 Design Features:

- **Synchronous Operation with Asynchronous Reset**
  The ALU runs on the rising edge of the clock for consistent timing, but it can reset immediately using an asynchronous reset — ideal for clean start-ups or emergency shutdowns.

- **Easily Configurable Bit Width**
  You can change the data width to 16, 32, 64, or even 128 bits just by updating a parameter — no need to rewrite the whole design. This makes the ALU highly reusable across different systems.

- **Smart Input Readiness Handling**
  The 2-bit INP_VALID signal tells you if zero, one, or both operands are ready. This helps the ALU work smoothly even when inputs don’t arrive at the same time, such as in pipelined or asynchronous setups.

- **Advanced Rotate Capabilities with Safety Checks**
  Rotate operations (left and right) are controlled by operand B, allowing flexible shifts. The ALU also checks for invalid values during rotate, helping you catch errors early.

- **Full Comparison Output in One Go**
  Instead of giving just one result, the ALU outputs all three: greater-than (G), less-than (L), and equal (E). This makes control decisions faster and easier for the system.

- **Automatic Overflow Detection**
  Arithmetic operations are checked for overflow automatically. You don't have to add extra logic to catch it — the ALU flags it for you, preventing calculation errors.

- **Built-in Power Saving Feature**
  The ALU has a Clock Enable (CE) input, so you can switch it off when it's not in use. Perfect for saving power in energy-sensitive or battery-powered devices.

1.7 Design Limitation:

- **One Error Bit for All Issues**
  There's just a single ERR flag that turns high when anything goes wrong. But it doesn't tell you what went wrong — was it a timeout, a wrong command, or bad input? You're left guessing.

- **Limited Rotate Range**
  Rotate operations are restricted to only 8 positions. That's fine for small data widths but not enough for wider operations like 32-bit or 64-bit rotates.

- **Fixed Operand Priority**
  If a timeout happens, the ALU always chooses the most recent operand. While this works in many cases, it might not match the logic expected in all systems — and you can't change this behavior easily.

- **Mode-Based Command Confusion**

  The same command value means different things in arithmetic and logical modes. For example, CMD = 0 might mean ADD in one mode and AND in another. This can easily confuse developers and cause bugs.

- **No Pipelining**

  The ALU handles one operation at a time. There’s no pipelining or parallel execution, which means performance might fall short in high-speed or heavily loaded systems.

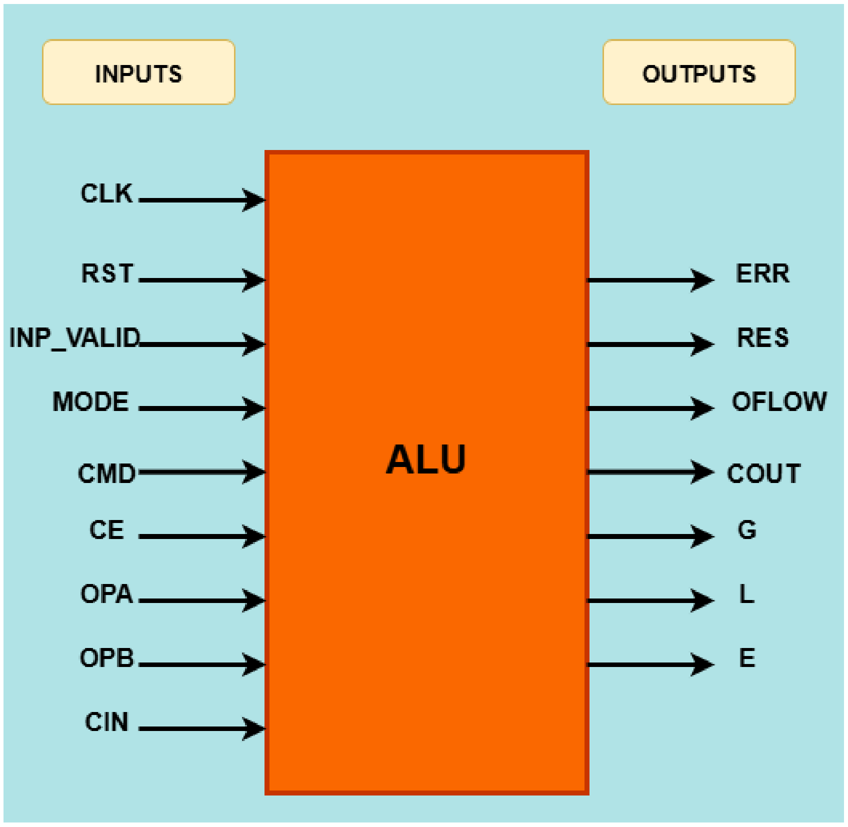
- **Fixed Timeout**

  The ALU is hardcoded to wait 16 clock cycles for both inputs. If your system runs faster or slower, you can’t tune this wait time without editing the design.

- **Wasted Resources**

  All features and operations are implemented in hardware, even if some aren’t used. That means more area and power consumption — not ideal for small or low-power chips.

<u>1.8 Design diagram with interface signals:</u>



:

Inputs:

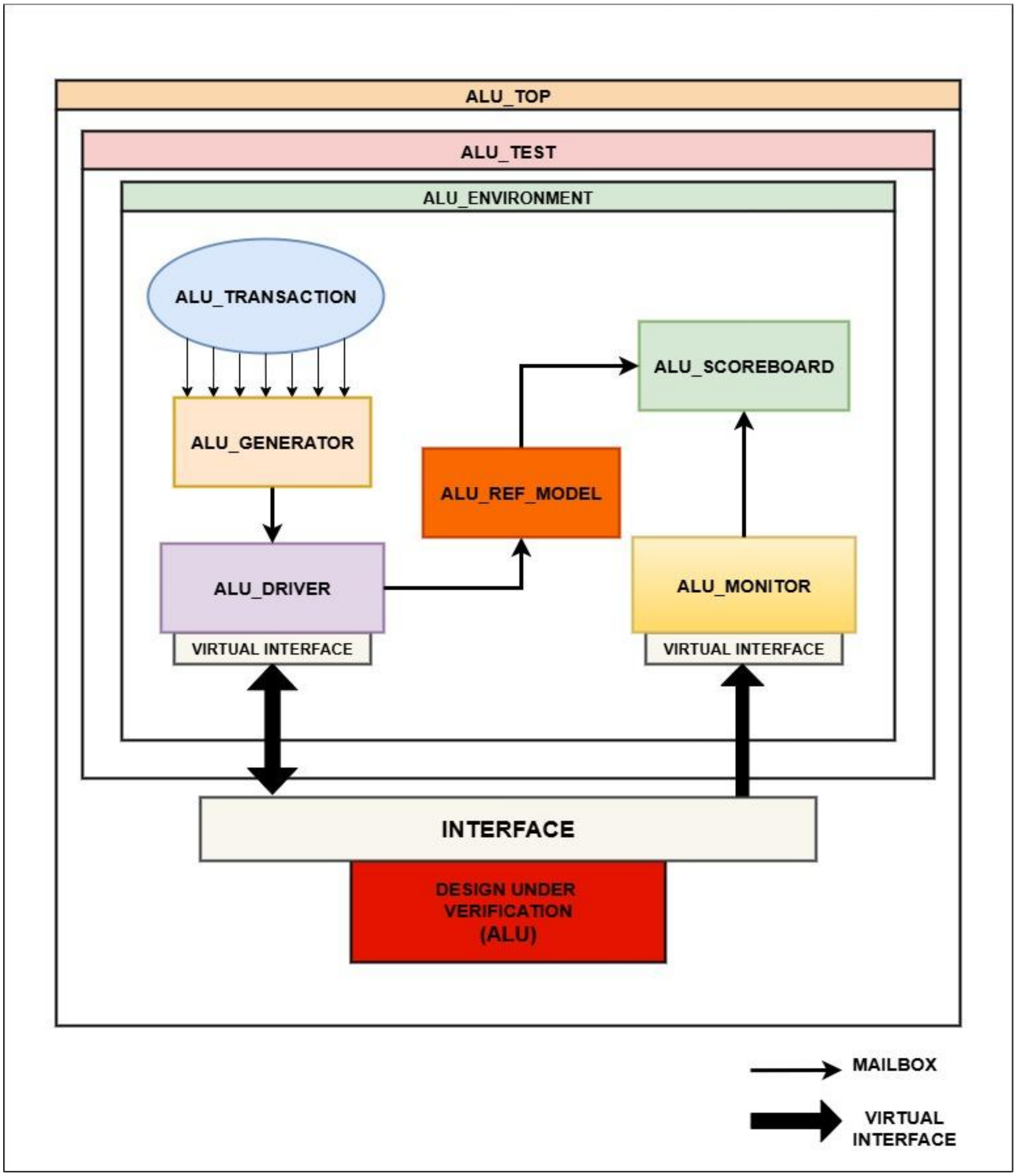
Signal Name	Type	Description
MODE	Input	Mode selection signal - determines ALU operation mode
OPA	Input	Operand A (Parameterized)
OPB	Input	Operand B(Parameterized)
CMD	Input	Command signal - specifies the specific ALU operation
INP_VALID	Input	Input valid signal - indicates when input data is valid
CIN	Input	Carry In - input carry for arithmetic operations

Outputs

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 for ALU:



The ALU design architecture depicted in the diagram follows a modular, layered structure typically used in SystemVerilog-based verification environments. At the highest level is the ALU_TOP module, which encapsulates the complete verification system. This includes the ALU_TEST block that serves as the testbench entry point, where different test scenarios are defined to stimulate and validate the ALU's functionality.

Central to the testbench is the ALU_ENVIRONMENT, which integrates all major verification components. The environment includes a Generator, which produces a stream of randomized or constrained transactions defined by the ALU_TRANSACTION class. These transactions represent the various inputs and control signals required by the ALU, such as operands, command codes, and mode selections.

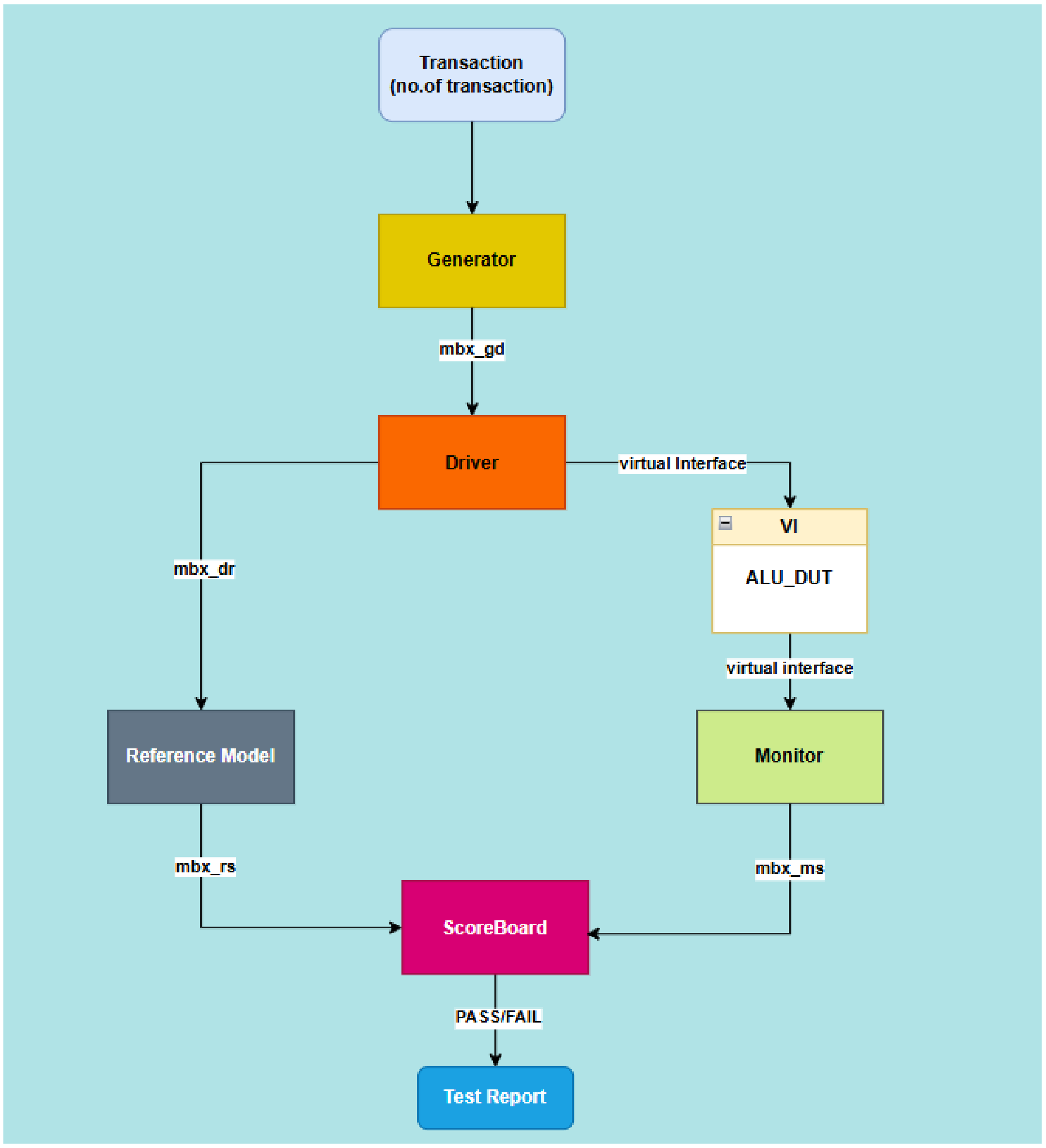
The generated transactions are passed to the ALU_DRIVER, which drives them to the Design Under Verification (DUV) using a virtual interface. The ALU itself is accessed through an INTERFACE module that provides signal-level connectivity between the driver/monitor and the DUT.

Simultaneously, the ALU_MONITOR observes the signals sent to and from the ALU using the same virtual interface and sends the observed behavior to the ALU_REF_MODEL (reference model). The reference model computes the expected result based on the given inputs, mimicking the intended behavior of the ALU.

Finally, the ALU_SCOREBOARD receives the actual results from the monitor and compares them against the expected results from the reference model. Any mismatches are flagged, helping verify the correctness of the ALU. Communication between components like the generator, scoreboard, and reference model is achieved through mailboxes, ensuring synchronization and data integrity across different test phases.

This structured design promotes reusability, modularity, and scalability, making it easier to maintain and extend the verification environment for more complex ALU features or new test scenarios.

2.3 FLOW CHART OF SV COMPONENTS :



The SystemVerilog component flowchart shown provides a clear and structured representation of the ALU verification environment. It begins with the Transaction block, which defines the number and nature of operations to be tested. These transactions are passed to the Generator, which is responsible for creating randomized or user-defined stimulus data, such as operands, operation codes, and control signals. This data is sent to the Driver via the mbx_gd mailbox.

The Driver receives these transactions and applies them to the ALU_DUT (Design Under Test) through a Virtual Interface (VI). The virtual interface provides a channel for signal-level interaction between the driver and the ALU hardware model. Simultaneously, the Driver also forwards the transaction details to the Reference Model using the mbx_dr mailbox. The reference model computes the expected output for the given inputs using a golden algorithm that reflects the intended ALU behavior.

On the other side, the Monitor observes the signals coming from the ALU_DUT using the same virtual interface. It captures the actual output and sends it to the Scoreboard through the mbx_ms mailbox. Meanwhile, the Reference Model also sends its expected results to the Scoreboard via the mbx_rs mailbox.

The Scoreboard plays a crucial role by comparing the actual and expected outputs. It determines whether the test case has passed or failed and generates the outcome accordingly. This result is finally forwarded to the Test Report block, which consolidates the results and provides a summary of the verification status, including PASS/FAIL indications.

Overall, this flowchart demonstrates a mailbox- and interface-driven architecture for modular, reusable, and scalable ALU verification, ensuring a systematic comparison of actual vs. expected behavior under various test scenarios.