

# VERIFICATION TEST PLAN

ECE-593: Fundamentals of Pre-Silicon Validation  
Maseeh College of Engineering and Computer Science  
Winter, 2025



**Project Name: RISC-V ALU**

Members: Angelo Maldonado-Liu, Saishree Lnu, Niko Nikolov

Date: 01/31/2025

[Introduction](#)

[Top Level Block Diagram](#)

[Specifications for the Design](#)

[Design Summary:](#)

[● Outputs:](#)

[● Internal Logic:](#)

[● Key Functions:](#)

[Verification Requirements](#)

[Hierarchy Level Verification:](#)

[Controllability and Observability:](#)

[Interfaces:](#)

[Design Verification Methodology](#)

[Risks and Dependencies](#)

[Risks](#)

[Dependencies](#)

[Tests and Methods](#)

[Arithmetic Operations](#)

[Control Signal Tests](#)

[Advanced Arithmetic](#)

[Regression Tests](#)

[Critical Functionality](#)

[System Integration](#)

[Special Cases and Corner Cases](#)

[Special Conditions](#)

[Error Injection](#)

[Testbench Components](#)

[Verification Tools](#)

[Black Box Testing](#)

[White Box Testing](#)

[Gray Box Testing](#)

[Selected Approach:](#)

[Gray Box Testing with White Box elements](#)

[Testbench Architecture](#)

[1. Driver](#)

[2. Monitor](#)

[3. Scoreboard](#)

[4. Checker](#)

[5. Coverage Collector](#)

[Coverage Requirements](#)

[Functional Coverage](#)

[Code Coverage](#)

[Selected Approaches](#)

[Dynamic Simulation](#)

[Formal Verification](#)

[Team Responsibilities:](#)

[Schedule](#)

[References Uses / Citations/Acknowledgements](#)

## Introduction

The objective of this verification plan is to ensure the correct functionality, performance, and integration of the Arithmetic Logic Unit (ALU) within the execute stage of a RISC-V core. This document outlines the verification strategy, design specifications, and requirements to validate the ALU's operations.

## Top Level Block Diagram

While not explicitly provided, the ALU's interaction within the broader RISC-V pipeline is implied, involving connections with the fetch, decode, and memory stages for seamless instruction processing.

## Specifications for the Design

The ALU module, named `rv32i_alu`, is designed to choose between the Program Counter (PC) or `rs1` for Operand A, and either `rs2` or an immediate value for Operand B, based on the instruction's opcode.

Additionally, it supports a range of arithmetic, logical, and comparison functions including ADD, SUB, SLT (Set Less Than), SLTU (Set Less Than Unsigned), XOR, OR, AND, SLL (Shift Left Logical), SRL (Shift Right Logical), SRA (Shift Right Arithmetic), EQ (Equal), NEQ (Not Equal), GE (Greater or Equal), and GEU (Greater or Equal Unsigned). Results are captured in the `y_d` register.

It also handles jumps computing the next PC for control flow instructions, utilizing signals like `o_change_pc` to indicate when the PC should be updated.

Furthermore, it manages register writeback by calculating values for the destination register (`rd`), controlling writeback operations with `o_wr_rd` and `o_rd_valid`, with writeback disabled for branches or stores.

Stalling mechanisms for memory operations using `o_stall_from_alu` and manage pipeline flushing through control signals like `i_stall`, `i_force_stall`, and `i_flush` will NOT be validated.

## Design Summary:

- *Outputs:*
  - o\_opcode: Passes the current instruction opcode to the next stage.
  - o\_exception: Propagates exceptions like illegal instructions, ecall, ebreak, or mret.
  - o\_y: ALU result for arithmetic/logical operations.
  - o\_pc: Current program counter.
  - o\_next\_pc: New program counter for jumps and branches.
  - o\_change\_pc: Indicates if the PC should be updated.
  - o\_wr\_rd, o\_rd\_addr, o\_rd, o\_rd\_valid: Control and data for writing to the destination register.
  - o\_stall\_from\_alu, o\_stall, o\_flush: Pipeline control signals for stalling and flushing.
- *Internal Logic:*
  - ALU Operations: Each operation (add, sub, slt, etc.) is handled, considering signed/unsigned where necessary.
  - Control Flow: Calculates new PC for jumps (JAL, JALR, branches) based on ALU results.
  - Register Write: Determines when to write results back, skipping for certain instructions.
  - Pipeline Control: Manages stalls and flushes to synchronize the pipeline stages.
- *Key Functions:*
  - Computes ALU results based on the current instruction.
  - Handles program counter updates for control flow instructions.
  - Manages register writes and pipeline synchronization.

## Verification Requirements

### **Hierarchy Level Verification:**

The verification is focused at the module level (ALU within the execute stage) to ensure each operation and control mechanism functions correctly in isolation before integrating into the full pipeline. This approach allows for detailed, targeted testing of the ALU's specific functionalities.

### **Controllability and Observability:**

Controllability is achieved by directly manipulating inputs like `i_rs1`, `i_rs2`, `i_imm`, and different opcodes, allowing for exhaustive testing of each ALU function. Observability is facilitated through comprehensive output signals like `o_y`, `o_next_pc`, and `o_rd`, which provide insights into the ALU's internal logic and decision-making process.

### **Interfaces:**

Interfaces are clearly defined with inputs such as `i_clk`, `i_rst_n`, `i_alu`, and outputs like `o_y`, `o_pc`, `o_next_pc`. Specifications include adherence to RISC-V ISA standards, ensuring that all operations align with the expected behavior for each opcode.

### **Design Verification Methodology**

Testbench (`rv32i_alu_tb`) is established to simulate ALU behavior, including clock generation and initial state setup for all inputs. This allows for dynamic testing of the ALU's response under various conditions.

Formal methods are employed to verify that no more than one operation or opcode is active at any time, ensuring logical integrity. Specific assertions check for the correct handling of signed versus unsigned operations.

## **Risks and Dependencies**

### **Risks**

- Incomplete Functional Coverage
- Resource Constraints
- Test Bench Complexity
- Accuracy of Reference Models
- Overlooking Custom or Rarely Used Operations
- Missed Edge Cases
- Timing and Performance Issues
- Regression Failures
- Validation Tool Limitations
- Human Error in Test Case Design
- Documentation Gaps

### **Dependencies**

- The success of the validation plan hinges on the team's proficiency in using UVM to create and manage the verification environment effectively.
- The ALU must be designed and validated against the RISC-V ISA specifications. Any deviation could lead to validation failures or incorrect ALU behavior.
- The validation effort depends on the availability and capability of simulation tools to accurately simulate ALU operations and provide detailed feedback.
- The validation process relies on having an accurate golden model or reference implementation to compare against. This ensures that the ALU's behavior is correct according to the specifications.
- The verification environment's effectiveness depends on well-designed drivers, monitors, scoreboards, etc., which must accurately interact with the ALU under test.
- The ability to assess and drive verification based on coverage metrics requires dependable tools for tracking and analyzing coverage data.
- The skill and experience of the validation team are critical for designing effective test cases, interpreting results, and managing the verification process.
- The validation schedule must be realistic, allowing enough time for thorough testing, regression runs, and adjustments based on findings.

## Tests and Methods

### Arithmetic Operations

Test			
ID	Test Name	Description	Features/Conditions
7.1.1	Basic ADD	Verify basic addition operation	- Test with positive numbers- Verify correct result in o_y
7.1.2	Basic SUB	Verify basic subtraction operation	- Test with positive numbers- Verify correct result in o_y
7.1.3	ALL ALU Ops	Test all arithmetic operations individually	- ADD, SUB, SLT, SLTU, XOR, OR, AND- Verify each operation produces correct result
7.1.4	Shift Operations	Test all shift operations	- SLL, SRL, SRA- Various shift amounts- Verify correct results

### Control Signal Tests

Test			
ID	Test Name	Description	Features/Conditions
7.2.1	Reset Behavior	Verify proper reset functionality	- Check all outputs are properly reset- Verify o_ce is cleared
7.2.2	Clock Enable	Verify clock enable functionality	- Test i_ce behavior- Verify pipeline stalling



7.2.3	Basic Pipeline	Test basic pipeline operation	- Verify signals propagate correctly- Check output timing
-------	----------------	-------------------------------	---

### Advanced Arithmetic

Test

ID	Test Name	Description	Features/Conditions
7.3.1	Overflow Cases	Test arithmetic overflow scenarios	- ADD with large numbers- SUB with negative results
7.3.2	Sign Extension	Verify proper sign extension	- SRA with negative numbers- SLT with mixed signs
7.3.3	Corner Cases	Test boundary conditions	- Maximum/minimum values- Zero handling

### Regression Tests

#### *Critical Functionality*

Test

ID	Test Name	Description	Features/Conditions
7.5.1	Basic ALU Suite	Comprehensive ALU operation test	- All ALU operations- Various input combinations

## System Integration

Test			
ID	Test Name	Description	Features/Conditions
7.6.1	Instruction Flow	Test complete instruction execution	- Multiple instruction types
7.6.2	Exception Handling	Verify exception propagation	- Various exception conditions- Proper exception handling

## Special Cases and Corner Cases

### *Special Conditions*

Test			
ID	Test Name	Description	Features/Conditions
7.7.1	Zero Handling	Test operations with zero	- All operations with zero operands- Zero result handling
7.7.2	Boundary Values	Test maximum/minimum values	- INT_MAX/INT_MIN operations- Overflow conditions

## Error Injection

Test			
ID	Test Name	Description	Features/Conditions
7.8.1	Invalid Operations	Test invalid operation handling	- Invalid ALU operations- Invalid opcodes

7.9.2	Timing Violations	Test timing edge cases	- Clock glitches- Setup/hold violations
-------	-------------------	------------------------	---

### **Testbench Components**

- Clock generator
- Reset controller
- Stimulus generator
- Response checker
- Coverage collector

### **Verification Tools**

- SystemVerilog simulator
- Coverage analysis tool
- Waveform viewer

### **Black Box Testing**

Pros:

- Tests functionality without internal knowledge
- Unbiased testing approach
- Simulates real-world usage
- Good for end-to-end testing

Cons:

- May miss internal corner cases
- Less efficient for coverage
- Harder to debug issues

## White Box Testing

Pros:

- Complete code coverage possible
- Direct access to internal states
- Efficient targeted testing
- Better for corner cases

Cons:

- May focus too much on implementation
- Can miss system-level issues
- Requires detailed design knowledge

## Gray Box Testing

Pros:

- Balances internal and external testing
- Good for interface testing
- Efficient coverage with reasonable effort

Cons:

- Requires both implementation and functional knowledge
- May have incomplete coverage of either aspect

## Selected Approach:

*Gray Box Testing with White Box elements*

Rationale:

- ALU is a complex module with both functional and implementation requirements
- Need to verify both internal operations and external interfaces
- Critical for pipeline integration testing

- Allows coverage-driven verification with internal state access

## Testbench Architecture

### Components

#### 1. *Driver*

- Generates clock and reset signals
- Drives input signals (i\_alu, i\_rs1, i\_rs2, etc.)
- Handles pipeline control signals

#### 2. *Monitor*

- Captures output signals (o\_y, o\_rd, etc.)
- Tracks pipeline states
- Logs transactions for analysis

#### 3. *Scoreboard*

- Validates ALU operations
- Checks pipeline behavior
- Verifies timing requirements

#### 4. *Checker*

- Validates protocol compliance
- Ensures proper pipeline stalling/flushing
- Verifies exception handling

#### 5. *Coverage Collector*

- Tracks functional coverage
- Monitors code coverage
- Reports coverage metrics

## Coverage Requirements

### Functional Coverage

- 100% coverage of all ALU operations
- 100% coverage of all opcode combinations
- 100% coverage of control signal combinations

### Code Coverage

- 100% line coverage
- 95% branch coverage
- 90% toggle coverage
- 85% FSM coverage

## Selected Approaches

### *Dynamic Simulation*

- Primary verification method
- SystemVerilog testbench
- Coverage-driven verification
- Reason: Best for functional verification of complex sequential logic

### *Formal Verification*

- Supplementary method for critical properties
- Focus on pipeline control properties
- Reason: Essential for verifying control logic correctness

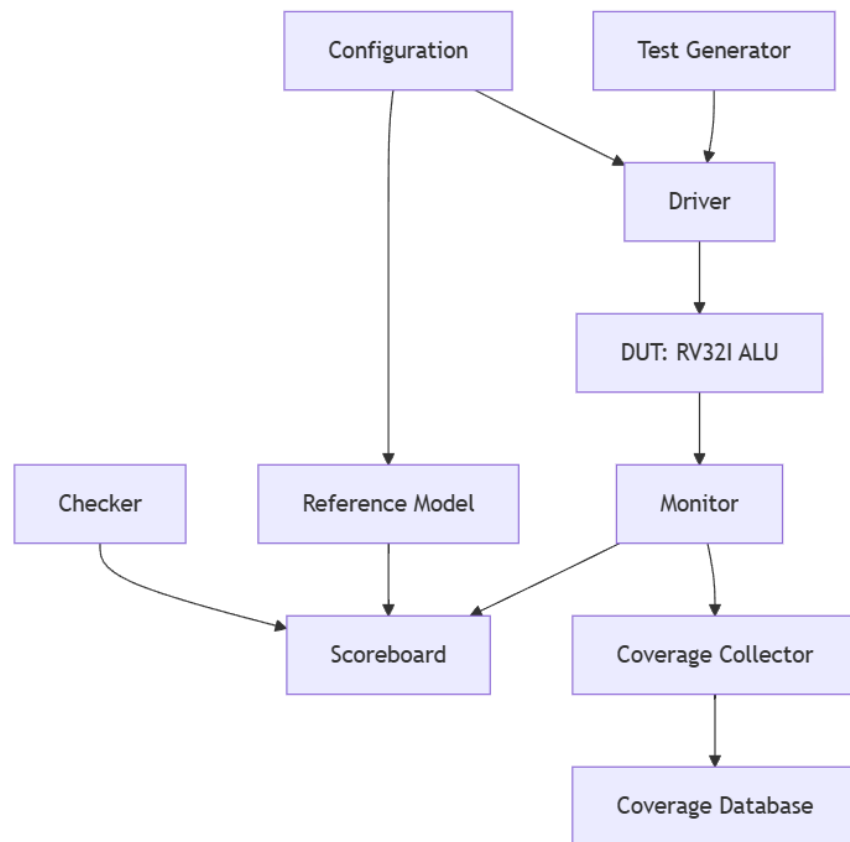


Figure 1: Validation flowchart for RISC-V ALU

## Team Responsibilities:

### Angelo:

#### ALU Arithmetic Operations

Verify the correctness of the ALU arithmetic operations.

#### Tasks:

- Test all arithmetic operations (add, sub, slt, sltu, sll, srl, sra).
- Ensure signed vs. unsigned comparisons (slt vs. sltu, sra vs. srl).
- Test edge cases, such as:
  - Overflow/underflow conditions.
  - Zero, maximum, and minimum values.
  - Signed vs. unsigned comparisons.
- Verify that the ALU produces the correct result for each operation under all possible input combinations.
- Check that the output o\_y is correctly computed and matches the expected result.

#### Tools/Skills:

- Write UVM test cases for arithmetic operations.
- Use sequence generators to cover all combinations of operands.
- Implement scoreboards to compare expected and actual results.

### Saishree:

#### Control Flow and Program Counter Logic

Validate the control flow logic, including jumps, branches, and program counter updates.

#### Tasks:

- Verify that the new PC (o\_next\_pc) is correctly calculated for each instruction.
- Ensure that o\_change\_pc and o\_flush are correctly set when a jump or branch occurs.
- Check the behavior of the ALU when control flow instructions



Tools/Skills:

- Write test cases for control flow instructions.
- Simulate scenarios with multiple jumps and branches.
- Ensure the pipeline behaves correctly after control flow changes.

**Niko:**

ALU Logic Operations

Verify the correctness of the ALU logic, reset and signal operations.

Tasks:

- Test all logic operations
- Test edge cases
- Verify that the ALU produces the correct result for each operation under all possible input combinations.
- Check that the output `o_y` is correctly computed and matches the expected result.

Tools/Skills:

- Write UVM test cases for arithmetic operations.
- Use sequence generators to cover all combinations of operands.
- Implement scoreboards to compare expected and actual results.

## Schedule

Date	Goals
1/31 ( M1 )	<ul style="list-style-type: none"> <li>• Complete Design spec</li> <li>• Preliminary Verification Document</li> <li>• Successful compilation of design in SV</li> <li>• Simple testbench for basic functionality</li> </ul>
M2	<ul style="list-style-type: none"> <li>• Class based testbench</li> <li>• interfaces completed and tested</li> <li>• complete compilation of transactions, generator, drivers and other components</li> <li>• verified at least 20-50 random bursts of data</li> <li>• update verification document</li> </ul>
M3	<ul style="list-style-type: none"> <li>• Finalized RTL</li> <li>• complete class based verification</li> <li>• all components defined and working</li> <li>• have code coverage and functional coverage reports</li> <li>• update verification document</li> </ul>
M4	<ul style="list-style-type: none"> <li>• Develop UVM Testbench</li> <li>• add UVM verification plan and details on UVM parts</li> <li>• utilize UVM_MESSAGING, UVM_LOGGING, etc for creating log and reports</li> </ul>
3/04	<ul style="list-style-type: none"> <li>• Complete UVM architecture, environment and testbench</li> <li>• All testcases completed and reflected in the coverage reports</li> <li>• Scenarios of bug injection to verify</li> <li>• Finalized documents and presentation</li> </ul>

## References Uses / Citations/Acknowledgements

- Alan-Tony. (n.d.). *GitHub - Alan-Tony/RISC-V-Datapath-and-Control: A basic ALU and ALU Control Unit Implementation using Verilog*. GitHub.  
<https://github.com/Alan-Tony/RISC-V-Datapath-and-Control>
- AngeloJacob. (n.d.). *GitHub - AngeloJacob/RISC-V: Design implementation of the RV32I Core in Verilog HDL with Zicsr extension*. GitHub.  
<https://github.com/AngeloJacob/RISC-V>
- Deng, L. (2024). Design a 5-stage pipeline RISC-V CPU and optimise its ALU. *Applied and Computational Engineering*, 34(1), 237–244.  
<https://doi.org/10.54254/2755-2721/34/20230334>
- *Design and implementation of 32-bit RISC-V processor using Verilog*. (2024, October 18). IEEE Conference Publication | IEEE Xplore.  
<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=10750638>
- Eymay. (n.d.). *GitHub - eyamay/ALU: A fast ALU design for RISC-V hardware*. GitHub.  
<https://github.com/eyamay/ALU>
- Patterson, D. A., & Hennessy, J. L. (2020). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann.
- Robert Baruch. (2018, June 22). *LMARV-1 (Tangible RISC-V) Part 3: Designing the ALU* [Video]. YouTube. <https://www.youtube.com/watch?v=KGBUtKRBKZs>
- RISC-V Instruction Set Summary. (n.d.). In *Single-Cycle Processor* (pp. 3–14).  
<https://courses.edx.org/assets/courseware/v1/f06a2dc0c856f60ec0711e9f5e1c98cf/asset-v1:HarveyMuddX+ENGR85B+1T2023+type@asset+block/FinalReferences.pdf>