# VERIFICATION TEST PLAN

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

**Project Name:** Design, Implementation and Verification of a MIPS-Lite 5-Stage In-Order CPU.

**Members:**

Mathumitha Rajan - 985876262

Srinivasan Venkatavardhan - 986982335

**Date:** 27th Jan 2026

**GIT REPO:**

https://github.com/MathumithaRajan/Mips_Lite_CPU

## Table of Contents

**2. Introduction:** ........................................ **Error! Bookmark not defined.**

**2.1. Objective of the verification plan** ........................................ **Error! Bookmark not defined.**

**2.2. Top Level block diagram** ........................................ **Error! Bookmark not defined.**

**2.3. Specifications for the design** ........................................ **Error! Bookmark not defined.**

**3. Verification Requirements** ........................................ **Error! Bookmark not defined.**

**3.1. Verification Levels** ........................................ **Error! Bookmark not defined.**

*3.1.1. What hierarchy level are you verifying and why?* ........................................ **Error! Bookmark not defined.**

*3.1.2. How are the controllability and observability at the level you are verifying*?**Error! Bookmark not defined.**

*3.1.3. Are the interfaces and specifications clearly defined at the level you are verifying. List them.* ........................................ **Error! Bookmark not defined.**

**4. Required Tools** ........................................ **Error! Bookmark not defined.**

**4.1. List of required software and hardware toolsets needed.** ........................................ **Error! Bookmark not defined.**

**4.2. Directory structure of your runs, what computer resources you will be using.** ............... **Error! Bookmark not defined.**

**5. Risks and Dependencies** ........................................ **Error! Bookmark not defined.**

**5.1 Critical threats, risks, and mitigation plans** ........................................ **Error! Bookmark not defined.**

 **Risk:** Incomplete hazard handling logic **Mitigation:** Begin verification with basic instruction sequences and incrementally add hazard-stress tests. ........................................ **Error! Bookmark not defined.**

 **Risk:** Ambiguity in instruction behavior **Mitigation:** Use a simple reference model to compare architectural state. ........................................ **Error! Bookmark not defined.**

 **Risk:** Debug complexity due to pipelining **Mitigation:** Verify pipeline stages incrementally and use waveform inspection. ........................................ **Error! Bookmark not defined.**

**6. Functions to be Verified.** ........................................ **Error! Bookmark not defined.**

**6.1. Functions from specification and implementation** ........................................ **Error! Bookmark not defined.**

*6.1.1. List of functions that will be verified. Description of each function* .........**Error! Bookmark not defined.**

*6.1.2. List of functions that will not be verified. Description of each function and why it will not be verified.* ........................................ **Error! Bookmark not defined.**

*6.1.3. List of critical functions and non-critical functions for tapeout* **Error! Bookmark not defined.**

**7. Tests and Methods** ........................................ **Error! Bookmark not defined.**

**7.1. Testing methods to be used:** ........................................ **Error! Bookmark not defined.**

**7.2. State the PROs and CONs for each and why you selected the method for this DUV.** . **Error! Bookmark not defined.**

**7.3. Testbench Architecture** ........ **Error! Bookmark not defined.**

**7.4. Verification Strategy:** ........ **Error! Bookmark not defined.**

**7.5. What is your driving methodology?** ........ **Error! Bookmark not defined.**

**7.6.What will be your checking methodology?** ........ **Error! Bookmark not defined.**

**7.7. Testcase Scenarios (Matrix)** ........ **Error! Bookmark not defined.**

**8. Coverage Requirements** ........ **Error! Bookmark not defined.**

8.3. Assertions ........ **Error! Bookmark not defined.**

9. Resources requirements ........ **Error! Bookmark not defined.**

**9.1 Team Members and Responsibilities** ........ **Error! Bookmark not defined.**

The project is executed by a two-member team. Responsibilities are divided to ensure balanced workload across design, verification, documentation, and integration. .. **Error! Bookmark not defined.**

**9.2 Workload Distribution** ........ **Error! Bookmark not defined.**

Schedule ........ **Error! Bookmark not defined.**

References Uses / Citations/Acknowledgements ........ **Error! Bookmark not defined.**

# 2. Introduction:

## 2.1. Objective of the verification plan

The objective of this verification plan is to validate the functional correctness of the MIPS-Lite 5-stage in-order CPU design. The plan focuses on verifying correct instruction execution, pipeline behavior, data flow, and basic hazard handling. Verification will be performed using a traditional SystemVerilog testbench with directed and simple randomized test programs. The goal is to ensure architectural correctness rather than performance optimization.

## 2.2. Top Level block diagram

The top-level design consists of a 5-stage pipelined datapath including Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Write Back (WB) stages. Pipeline registers are present between each stage. The verification environment interacts with the DUT at the top-level CPU interface, driving clock, reset, and observing architectural state such as registers and memory.

## 2.3. Specifications for the design

The design under verification is a simplified MIPS-Lite CPU with the following characteristics:

- In-order, single-issue execution
- 5-stage pipeline (IF, ID, EX, MEM, WB)
- Fixed 32-bit instruction format
- Support for arithmetic, logical, immediate, load/store, and branch instructions
- Single clock domain
- No speculative or out-of-order execution

# 3. Verification Requirements

## 3.1. Verification Levels

### *3.1.1. What hierarchy level are you verifying and why?*

Verification will be performed at the top-level CPU (system level). This level is chosen to validate correct end-to-end instruction execution, pipeline interaction between stages, and architectural state updates such as register file and memory contents.

### *3.1.2. How are the controllability and observability at the level you are verifying?*

Controllability is achieved by driving instruction sequences through instruction memory and controlling clock and reset signals. Observability is achieved by monitoring architectural state including register file contents, data memory, and selected internal pipeline signals exposed for debug purposes.

### *3.1.3. Are the interfaces and specifications clearly defined at the level you are verifying. List them.*

Yes. The top-level CPU interface includes clearly defined clock, reset, instruction memory interface, and data memory interface. Architectural behavior is defined by the MIPS-Lite specification documented in the design specification.

# 4. Required Tools

## 4.1. List of required software and hardware toolsets needed.

- SystemVerilog
- Questa / ModelSim simulator
- Git for version control
- Text editor (VS Code)
- draw.io (for diagrams)

## 4.2. Directory structure of your runs, what computer resources you will be using.

The project follows the directory structure provided by the course Makefile. RTL files are stored under the `rtl/` directory and testbench files under the `TRAD_TB/` directory. Simulations will be run on a local development machine using the provided Makefile and run scripts.

# 5. Risks and Dependencies

## 5.1 Critical threats, risks, and mitigation plans

- **Risk:** Incomplete hazard handling logic
  **Mitigation:** Begin verification with basic instruction sequences and incrementally add hazard-stress tests.
- **Risk:** Ambiguity in instruction behavior
  **Mitigation:** Use a simple reference model to compare architectural state.
- **Risk:** Debug complexity due to pipelining
  **Mitigation:** Verify pipeline stages incrementally and use waveform inspection.

# 6. Functions to be Verified.

## 6.1. Functions from specification and implementation

### *6.1.1. List of functions that will be verified. Description of each function*

- Instruction fetch and program counter update
- Register file read and write operations
- ALU arithmetic and logical operations
- Load and store memory operations
- Branch instruction behavior (BEQ/BNE)
- Pipeline register propagation
- Basic data hazard handling (forwarding, stalls)

### *6.1.2. List of functions that will not be verified. Description of each function and why it will not be verified.*

- Performance optimization
- Cache behavior
- Interrupts and exceptions
- Advanced branch prediction mechanisms

These are excluded to maintain focus on core pipeline functionality.

### *6.1.3. List of critical functions and non-critical functions for tapeout*

**Critical functions:**

- Correct instruction execution
- Correct register file updates
- Correct memory access behavior

**Non-critical functions:**

- Performance metrics
- Advanced debug features

# 7. Tests and Methods

## 7.1. Testing methods to be used:

**Gray-box testing** will be used. Internal signals may be observed for debugging while verification remains functionally driven.

## 7.2. State the PROs and CONs for each and why you selected the method for this DUV.

**Pros:**

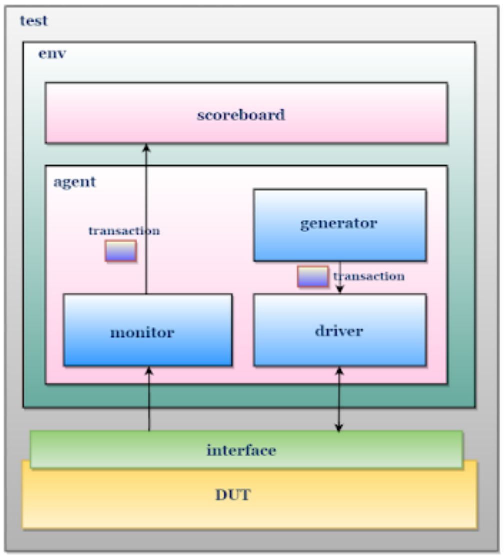
- Better visibility into pipeline behavior
- Easier debugging during early milestones

**Cons:**

- Reduced abstraction compared to pure black-box testing

Gray-box testing is selected to accelerate debug during MS1.

## 7.3. Testbench Architecture



A traditional SystemVerilog testbench will be used consisting of:

- Stimulus generator (instruction sequences)
- DUT instantiation
- Monitors to observe architectural state
- Simple scoreboard / reference model for comparison

**For Mileston-2:**

For Milestone-2, the verification environment has been enhanced from a traditional procedural testbench to a class-based SystemVerilog verification architecture.

The testbench now consists of the following components:

- Instruction generator producing directed instruction sequences and expected ALU results
- Driver interfacing stimulus to the DUT through a SystemVerilog interface
- Input monitor observing DUT inputs for debugging visibility
- Output monitor capturing DUT outputs and forwarding them to the scoreboard
- Scoreboard comparing expected vs observed results using mailbox communication
- Functional coverage collector tracking opcode, register usage, and ALU result bins

This modular class-based architecture improves scalability, reusability, and debugging efficiency compared to the initial traditional testbench.

## 7.4. Verification Strategy:

Dynamic simulation using a SystemVerilog testbench is chosen for this milestone. This approach allows step-by-step validation of pipeline behavior and debugging using waveforms.

The verification strategy transitioned to a hybrid directed and constrained-stimulus approach using a class-based testbench. Directed instruction sequences were used for initial functional validation, while mailbox-based communication enabled asynchronous checking and transaction tracking. Functional coverage was incorporated to measure instruction execution completeness.

## 7.5. What is your driving methodology?

Testcases will be driven using:

- Directed instruction sequences
- Directed instruction sequences generated through a class-based stimulus generator, with infrastructure prepared for constrained random stimulus in future milestones.

## 7.6.What will be your checking methodology?

Checking will be performed by comparing DUT architectural state against expected results derived from:

- Instruction specification
- Reference model behavior

Checking is performed through a scoreboard that compares expected ALU results generated from a reference instruction model against observed DUT outputs captured by output monitors. The scoreboard currently reports mismatches due to synchronization latency between expected and observed transactions, which will be resolved in subsequent milestones by improving pipeline alignment logic.

## 7.7. Testcase Scenarios (Matrix)

### *7.7.1. Basic Tests*

| Test Name | Test Description/ Features |
|---|---|
| IF_Test | Verify instruction fetch and PC increment |
| RF_Test | verify register read/write functionality |

### *7.7.2. Complex Tests*

| Test Name / Number | Test Description/ Features |
|---|---|
| Pipeline_Test | This test verifies correct pipelined execution of multiple instructions<br>flowing concurrently through different pipeline stages (IF, ID, EX, MEM, WB).<br>The test includes:<br>- Back-to-back instruction execution to ensure proper pipeline overlap<br>- Multiple dependent instructions to expose data hazards<br>- Verification of correct data propagation across pipeline registers<br>- Observation of register file updates and ALU outputs under continuous instruction flow<br>This test ensures that instructions at different pipeline stages do not corrupt architectural state and that sequential execution semantics are maintained.<br>Note: Full hazard detection, forwarding, and stall behavior will be thoroughly verified in later milestones. |
| Branch_Test | This test verifies control-flow behavior for branch instructions.<br>The test includes:<br>- Taken branch scenarios where the Program Counter (PC) is redirected<br>- Not-taken branch scenarios where sequential PC progression is maintained<br>- Verification of correct instruction fetch after branch resolution<br>- Observation of pipeline behavior following a control-flow change<br>This test validates that branch decisions are handled correctly and that incorrect instructions are not committed to architectural state. |

| | |
|---|---|
| | Note: Branch prediction and advanced control hazard handling are deferred to later milestones. |

### *7.7.3. Regression Tests (Must pass every time)*

| Test Name / Number | Test Description/Features |
|---|---|
| Smoke_Test | Basic sanity test |
| ISA_Regression | Basic instruction coverage |

### *7.7.4. Any special or corner cases testcases*

| Test Name / Number | Test Description |
|---|---|
| Load_Use | Load-use hazard condition |
| Back_to_Back | Back-to-back dependent instructions |

# 8. Coverage Requirements

## 8.1. Describe Code and Functional Coverage

Functional coverage will track:

- Instruction types exercised
- Branch taken/not taken scenarios
- Load/store execution paths

Code coverage will be monitored to ensure RTL lines are exercised.

## 8.2. Formulate conditions of how you will achieve the goals. Explain the Covergroups and Coverpoints and your selection of bins.

Coverage goals for this project are achieved through a combination of directed testing, observation of architectural signals, and functional coverage defined using SystemVerilog covergroups.

At Milestone 1, the primary objective is to establish baseline functional coverage to ensure that the major datapath components are exercised and that basic instruction flow through the pipeline is observable. Coverage will be incrementally expanded in later milestones as additional features such as hazard handling and control logic are introduced.

**Covergroups and Coverpoints**

SystemVerilog covergroups are used to capture functional coverage of key architectural behaviors. Each covergroup is associated with a logical function of the design and samples signals at appropriate clock boundaries.

Coverpoints Selection

The following coverpoints are defined to measure meaningful design activity:

- Program Counter (PC) Progression
    - Covers sequential PC updates (PC + 4)
    - Ensures instruction fetch logic is exercised
- Instruction Type Coverage
    - Arithmetic instructions (ADD, SUB)
    - Logical instructions (AND, OR)
    - Immediate-type instructions (ADDI)
    - Control-flow instructions (branch – where applicable)
- Register File Activity
    - Register read access
    - Register write-back events
    - Ensures architectural state updates are occurring
- ALU Operation Coverage
    - Each supported ALU operation is covered
    - Confirms correct selection and execution of ALU control paths

**Bins Definition and Selection**

Bins are selected to represent meaningful functional scenarios rather than exhaustive value ranges.

- Instruction bins group instructions by type to ensure all supported classes are exercised.
- ALU operation bins ensure that each ALU operation is executed at least once.
- Register write bins track whether write-back occurs under different instruction conditions.

Bins are intentionally coarse-grained at this milestone to avoid over-constraining the testbench and to allow focused validation of basic functionality.

**Coverage Closure Strategy**

Coverage closure at Milestone 1 is achieved by:

- Running directed instruction sequences that explicitly target uncovered bins
- Inspecting coverage reports after each simulation run

- Refining stimulus only when necessary to hit uncovered functional paths

Advanced coverage scenarios such as cross coverage between pipeline stages, hazard conditions, and branch interactions will be introduced in later milestones when corresponding design logic is implemented.

Milestone 2:

Functional coverage is implemented using SystemVerilog covergroups monitoring opcode types, register usage, and ALU result distributions. Current functional coverage is approximately 60%, reflecting execution of basic instruction classes. Coverage will improve as additional instruction types and hazard scenarios are introduced.

### 8.3. Assertions

Assertions will be added to check:

- No invalid register writes
- Correct stall behavior
- Valid pipelines register transitions

Assertions help catch protocol and pipeline errors early.

## 9. Resources requirements

### 9.1 Team Members and Responsibilities

The project is executed by a two-member team. Responsibilities are divided to ensure balanced workload across design, verification, documentation, and integration.

**Team Member 1: Srinivasan Venkatavardhan (RTL Design )**

**Responsibilities:**

- Overall project coordination and milestone planning
- RTL design of the MIPS-Lite CPU pipeline stages:
    - Instruction Fetch (IF)
    - Instruction Decode (ID)
    - Execute (EX)
    - Memory (MEM)
    - Write Back (WB)
- Design of pipeline registers and datapath control signals
- Implementation of hazard handling logic (basic forwarding and stall mechanisms)
- Integration of all pipeline stages into the top-level CPU module

- Debugging RTL functionality using waveform analysis
- Supporting verification team with internal signal visibility and debug hooks

**Expertise:**

- Digital design using SystemVerilog
- CPU microarchitecture and pipelining concepts
- RTL coding and debugging
- Computer architecture (MIPS ISA understanding)
- Git-based version control

## Team Member 2: Teammate Name (Verification Engineer & Testbench Developer)

## Responsibilities:

- Development of traditional SystemVerilog testbench (TRAD_TB)
- Creation of directed instruction test programs for:
  - ALU operations
  - Register file verification
  - Load/store behavior
  - Branch instructions
- Development of simple randomized instruction streams for stress testing
- Monitor and scoreboard implementation for architectural state comparison
- Functional coverage and code coverage setup
- Assertion development for pipeline correctness and hazard detection
- Regression test setup (Smoke, ISA Regression, Pipeline Stress Tests)
- Documentation of verification methodology and results.

**Expertise:**

- SystemVerilog testbench development
- Functional and code coverage
- CPU ISA verification techniques
- Debugging using waveform viewers (Questa/ModelSim)
- Basic reference model development

### 9.2 Workload Distribution

| Task Area | Srinivasan Venkatavardhan (%) | Mathumitha Rajan (%) |
|---|---|---|
| RTL CPU Design | 70 | 30 |
| Pipeline & Hazard Logic | 80 | 20 |
| Testbench Development | 10 | 90 |
| Testcase Creation | 30 | 70 |
| Coverage & Assertions | 40 | 60 |
| Debug & Integration | 50 | 50 |
| Documentation | 50 | 50 |

### 9.3 Collaboration and Integration Plan

- Weekly integration checkpoints to merge RTL and testbench
- Git-based version control for collaborative development
- Joint debugging sessions using waveform inspection
- Shared reference model for architectural state validation

## Schedule

| Milestone | Due Date | Points | Planned Activities |
|---|---|---|---|
| Milestone-1 | Jan 30 | 10 pts | Verification plan, CPU architecture design, initial RTL modules |
| Milestone-2 + Milestone-3 | Feb 18 | 40 pts | Complete pipeline RTL, hazard logic, directed testcases, initial debug |

| | | | |
|---|---|---|---|
| Milestone-4 | Feb 27 | 20 pts | Add randomized tests, assertions, coverage analysis, regression testing |
| Milestone-5 (Final) | Mar 14 | 30 pts | Final integration, documentation, report submission, final demo |

## References Uses / Citations/Acknowledgements

- Course lecture slides (ECE-593)
- Patterson & Hennessy, Computer Organization and Design
- MIPS Architecture Documentation