,

VERIFICATION TEST PLAN

ECE-593: Fundamentals of Pre-Silicon Validation

Maseeh College of Engineering and Computer Science

Winter, 2025

Project Name: MicroProcessor System

Members: Frezewd Debebe, Janvier Rutihunza, Sal Esmael

Date: February 19, 2026

Contents

1 Introduction

1.1 Objective of verification plan

The objective of this verification plan is to ensure the correctness, functionality, and performance of the multi-processor system with shared memory architecture. This plan outlines a comprehensive strategy to verify that the Design Under Test (DUT) works according to the specifications and meets all functional requirements.

Specific objectives include:

1. Functional Verification: Verify that all components (processor cores, memory subsystem, arbiter, cache, and directory-based coherency protocol) function correctly both individually and as an integrated system.
2. Cache Coherency Validation: Ensure the directory-based coherency protocol maintains data consistency across all three processor caches under various read/write scenarios and concurrent access patterns.
3. Arbiter Performance: Verify that the arbiter correctly handles simultaneous memory access requests from multiple processors, implements the chosen arbitration scheme (round-robin/fixed priority), and ensures fair or prioritized access without deadlocks.
4. Memory Access Verification: Confirm that the 2KB memory subsystem correctly handles read and write operations from all processors, with proper addressing (11-bit) and data integrity (1-byte width).
5. Interface Validation: Verify that all interfaces between components (Memory Interface Units, shared bus, arbiter-to-cache, cache-to-memory) operate correctly with proper handshaking and data transfer protocols.
6. Corner Case Testing: Identify and test edge cases including simultaneous write conflicts, cache invalidation scenarios, bus contention, and maximum load conditions.

A successful verification considers to have:

- All functional test cases pass with 100% success rate
- Cache coherency is maintained across all tested scenarios
- Code coverage reaches minimum 90% (line and functional coverage)

- All corner cases are tested and pass
- Timing and performance requirements are met
- No critical or major bugs remain unresolved

1.2 Top Level block diagram

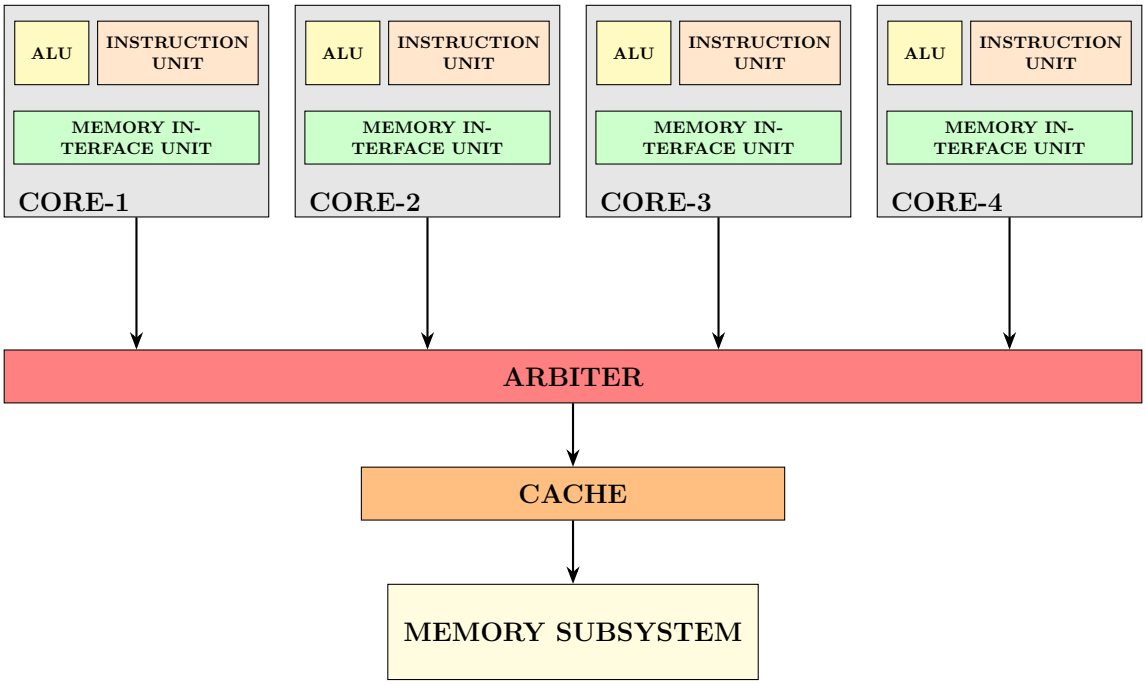

Figure 1: 4-Core Processor Architecture

1.3 Specifications for the design

- Memory Size: 2KB (1 Memory Array partitioned)
- Width of 1 Byte Memory
- 2K Depth
- 4 Processors
- Robin Arbiter

1.4 RTL Design Implementation (Generator-Based Processors – Milestone 1)

For Milestone 1 of the Multiprocessor System, processor functionality is implemented using an RTL generator-based approach rather than full instruction-level CPU cores. This design choice enables early pre-silicon validation of the system architecture while keeping the implementation complexity manageable. The primary focus at this stage is verifying correct system-level behavior, including arbitration, shared bus operation, and memory dataflow, before introducing full processor pipelines in later milestones.

Each processor is modeled as a deterministic transaction generator that interacts with the shared bus, centralized arbiter, and shared memory subsystem through a simple request–grant interface. The generator issues memory read and write transactions using explicit control and data signals, allowing validation of arbitration fairness, bus isolation, and memory correctness under concurrent access conditions.

Generator Functional Behavior

After reset, each generator executes a fixed and deterministic sequence of operations. First, the generator requests access to the shared bus. Once bus access is granted, the generator writes a unique data value to a unique memory address. The generator then issues a read request to the same memory address and captures the returned data from the memory subsystem. The read data is compared against the expected value written earlier. Based on the comparison result, the generator asserts a done signal and reports a pass or fail status. This deterministic execution model ensures repeatable simulations, simplifies debugging, and allows direct correlation between expected and observed system behavior.

Role in System Verification

The generator-based processor implementation validates several critical components of the multiprocessor system. It verifies correct operation of the centralized round-robin arbiter under simultaneous processor requests, ensures proper bus grant logic and isolation between processors, and confirms correct shared-memory read and write operations with maintained data integrity. Additionally, it validates end-to-end dataflow correctness between the processors and the shared memory subsystem. By abstracting processor behavior into generators, the design establishes a scalable and modular verification foundation. These generator modules can be replaced with real instruction-level processor cores in later milestones without requiring modifications to the shared bus, arbiter, or memory subsystem.

2 Verification Requirements

2.1 Verification Levels

2.1.1 What hierarchy level are you verifying and why?

Our verification strategy employs a bottom-up hierarchical approach, progressing from unit-level to system-level verification across three methodologies: conventional, class-based, and UVM-based verification.

1. Unit/Module Level (Conventional Testbench)
    - It is rooted in a directed testing mindset. In this approach, the testbench is usually a static module that wraps around the Design Under Test (DUT) and individual components verified in isolation.
    - Why: Catch bugs early in individual blocks before integration; easier to debug and isolate issues; verify basic functionality of each component
2. Block/Subsystem Level (Class-Based Testbench)
    - shifts the focus from "signals" to "transactions." It can wrap data into objects (like a network packet or a bus command) and use high-level methods to move that data around. Instead of writing one test for one scenario, you write a set of rules (constraints) and let the computer generate thousands of random variations.
    - Why: Verify component interactions and interfaces; test handshaking protocols; validate data flow between related modules.
3. System/Top Level (UVM Testbench)
    - It is industry-standard verification methodology for verifying digital designs. It's built on top of SystemVerilog and provides a standardized framework for creating reusable, scalable, and robust testbenches.
    - Why: Verify end-to-end functionality; test cache coherency scenarios; validate multi-core interactions; ensure system-level requirements are met; test real-world use cases.

2.1.2 How is controllability and observability at the level you are verifying?

Controllability is the ability to set/control the inputs and internal states of the DUT while Observability is the ability to observe/monitor the outputs and internal states of the DUT.

Conventional testbench: At the unit level, both controllability and observability are very high, making this the easiest verification level. Controllability is maximized because we have direct access to all module inputs, allowing us to set any input combination, control clock and reset signals directly, and create any test condition without protocol constraints. For example, when testing the ALU, we can simply assign any values to operand_a and operand_b and immediately test the operation. Similarly, observability is very high because all outputs are directly visible in the testbench, internal signals can be accessed through hierarchical references, and waveform analysis provides immediate feedback on behavior. There is minimal latency between applying stimulus and observing the response, making it straightforward to identify and debug issues. The primary challenge at this level is simply ensuring comprehensive coverage of all possible input combinations, but the verification process itself is straightforward due to the complete visibility and control we have over individual components.

Class-based testbench: At the block level, both controllability and observability are high but with some additional complexity compared to unit level. Controllability remains strong because we can create specific transaction sequences through class-based drivers and use configuration objects to modify behavior, but we must now adhere to interface protocols including proper handshaking and timing requirements. For instance, testing a complete processor core requires following the instruction fetch protocol rather than simply forcing values into internal registers, and reaching specific cache states requires coordinated sequences of read and write operations. Observability is similarly high, with transaction-level monitors capturing all interface activity and protocol compliance, though some internal states must be inferred from external behavior rather than observed directly. The challenges at this level include the need for multi-step sequences to reach certain internal states, timing dependencies between components that require careful orchestration, and the fact that internal processor state like the program counter cannot be directly controlled but must be manipulated through instruction execution.

UVM testbench: At the system level, both controllability and observability are moderate, representing the most challenging verification environment but also the most realistic representation of actual system operation. Controllability is limited by the interaction of multiple components, where changing one component's state affects others through cache coherency protocols, arbiter interactions, and shared memory access. Creating specific system states requires carefully coordinated multi-step sequences across multiple cores.Observability is similarly moderate because system state is distributed across multiple components including three separate caches, the directory controller, the arbiter, and the memory subsystem, requiring correlation of observations from multiple monitors to reconstruct overall behavior. We cannot directly see cache coherency states across all cores simultaneously but must infer them from interface transactions, directory updates, and coherency protocol messages.

2.1.3 Are the interfaces and specifications clearly defined at the level you are verifying. list them?

Key interface specification:

- Memory addressing: 2KB locations
- Data width: 8-bit (1 byte per location)
- Arbitration: Request-grant protocol
- Cache coherency: Directory based coherency protocol

Conventional testbench: All unit-level interfaces are clearly defined with explicit signal names, bit widths, and timing specifications. Each module has between 3-10 signals consisting of data inputs/outputs, control signals, and bus signals.

3 Required Tools

3.1 List of required software and hardware toolset needed

For this project, we are utilizing software simulation tool to analyze the working and performance of Multi-processor system. The following are list of required tools used:

- HDL Simulation and Verification: QuestaSim
- Technical Documentation: LaTeX via Overleaf and Google Docs
- Scripting: SystemVerilog

3.2 Directory structure of your runs, what computer resource you will be using

we use git as revision control and the file structure is:

- doc/ - Design specifications, verification plan, reports, diagrams, and snapshots
- rtl/ - SystemVerilog design files including processor cores, arbiter, cache, directory controller, memory, and top-level integration
- TB/ - Testbenches organized by methodology: Concentional, Class-based, and UVM
- coverage/ - Code and functional coverage reports

Computer Resources :

- Platform: University ECE CAD server (username@phobos.ece.pdx.edu)
- Access: Remote SSH. VPN required for off-campus access.
- Local Requirements: Personal computer with SSH client, VPN client, and Git.

4 Risks and Dependencies

4.1 List all the critical threats or any known risks. List contingency and mitigation plan

5 Functions to be verified

5.1 Functions from specification and implementation

5.1.1 List of functions that will be verified. Description of each function

5.1.2 List of functions that will not be verified. Description of each function and why it will not be verified

5.1.3 List of critical functions and non-critical functions for tapeout

6 Tests and Methods

6.1 Testing methods to be used: Black/White/Gray Box

- **Milestone 1:** a Gray Box testing approach was adopted. While the Scoreboard performed Black Box functional verification of the ALU operations, the use of a SystemVerilog Interface and Monitors allowed for the observation of internal bus arbitration and core-specific handshaking. This provided the necessary visibility to debug multi-core contention while maintaining the independence of the reference model

6.2 State the PROs and CONs for each and why you selected the method for this DUV

Method	PROS	CONS
Black Box	Faster testbench development; decoupled from RTL changes; focuses strictly on user-level requirements.	Difficult to hit high code coverage; hard to verify internal arbitration logic or hidden state machines.
White Box	Provides 100% visibility into internal signals; essential for reaching 100% Toggle and Condition coverage.	Highly "brittle" (breaks if internal RTL signal names change); testbench becomes overly complex and implementation-dependent.
Gray Box	Balances visibility and stability. Allows protocol checking on internal buses while keeping the reference model independent.	Requires more initial setup; requires understanding of both architecture (specs) and micro-architecture (RTL).

Milestone 1

For this specific Design Under Verification (DUV), Gray Box was selected for three primary reasons:

- Multi-Core Arbitration: A pure Black Box approach cannot verify if the Arbiter is correctly prioritizing Core 0 over Core 3 during a collision. By using Gray Box, we can monitor the internal core_id and grant signals to ensure the hardware is following the priority rules defined in the spec.
- Protocol Integrity: The DUV uses a specific handshake (read_en, ready, rvalid). Gray Box allows us to place SystemVerilog Assertions (SVA) on these internal interface signals to catch timing violations (like a data-valid signal appearing without a request) the moment they happen.
- Bridging the Coverage Gap: As seen in our Milestone 2 results, while functional coverage (Opcode hits) was high, code coverage (Toggles) was lagging. The Gray Box method allows us to identify exactly which internal bit-slices or logical branches are "cold" and write directed constraints to target them specifically.

6.3 Testbench Architecture: Component used (list and describe Drivers, Monitors, Scoreboards, Checkers etc)

Milestone 1

- **Generator:** Handles the "Scenario" level. It uses SystemVerilog constraints to create a distribution of operations (ALU, Memory, NOPs) across all four cores. It is responsible for hitting the "Corner Cases" like maximum address ranges.
- **Driver:** The only component that "talks" to the DUV pins. It translates the high-level transaction objects into the physical signaling required by the bus protocol, managing the timing of the reset and request phases.
- **monitors:**
  - **Input Monitor:** Records what the Driver sent to ensure the DUV received the correct stimulus.
  - **Output Monitor:** Captures the DUV's response. It is crucial for Gray Box testing as it tracks which core the response belongs to.
- **Scoreboard & Reference Model:** The "Decision Maker." The Reference Model mimics the DUV's expected behavior in pure software (e.g., expected = A * B). The Scoreboard compares this "Golden" result against the Actual result from the Output Monitor and reports PASS/FAIL.
- **Coverage Collector:** A passive component that subscribes to the Monitor's data stream to update the covergroups. It tracks our progress toward the 100% Functional Coverage goal.

6.4 Verification Strategy: (Dynamic Simulation, Formal Simulation, Emulation etc.) Describe why you chose the strategy

Dynamic Simulation involves applying a sequence of input stimuli (test vectors) to a design over a period of time and observing the resulting output. It relies on a logic simulator to calculate the response of the RTL code based on the timing and logic defined in the source files.
For this multi-core DUV, Dynamic Simulation is the most appropriate strategy for the following reasons:

- **Handling Non-Deterministic Randomness:** Since we are using Constrained Random Verification (CRV), we need a simulator that can handle millions of different random combinations of Core IDs, Opcodes, and Addresses. Dynamic simulation is the native environment for SystemVerilog's randomization engine.
- **Protocol Timing Verification:** Unlike Formal verification (which is mathematical), Dynamic Simulation allows us to see exactly how the design behaves clock-cycle by clock-cycle. This is critical for verifying the Arbiter's timing and the handshaking logic (ready, rvalid) between the cores and memory.

- **Functional Coverage Feedback:** Dynamic simulation provides immediate feedback via covergroups. We can see exactly which "Crosses" (like Core 2 + NOP) have been hit during a run, allowing us to iterate quickly.
- **Cost and Resource Efficiency:** For a design of this scale (a 4-core processor/ALU), Emulation (using specialized hardware like Palladium or Zebu) would be overkill and prohibitively expensive. Dynamic simulation provides the necessary performance on standard workstation hardware.

6.5 What is your driving methodology?

6.6 What will be your checking methodology?

6.7 Testcase Scenarios (Matrix)

7 Coverage Requirements

7.1 Assertions

8 Resource requirements

8.1 Team members and who is doing what and expertise

Milestone 1

- **Frezewd:** Preparing Verification Plan Document.
- **Janvier:** Preparing Design Specification Document.
- **Sal Esmael:** Preparing the Design file and Testbench, Readme file.

Milestone 2 & Milestone 3

- **Frezewd:** Transaction, Generator, Driver, Scoreboard, updating Verification Plan
- **Janvier:** Monitor_in, Monitor_out, Interface, Report
- **Sal Esmael:** RTL, tb_top, functional coverage

Milestone 4

- **Frezewd:** RTL, UVM TB: top level environment,scoreboard, section for UVM in the verification Plan
- **Janvier:** RTL, UVM TB: Sequencer, driver, scoreboard
- **Sal Esmael:** RTL, UVM TB: monitor,scoreboard, testbench report

Milestone 5

- **Frezewd:** Bug injection and verification, Verification plan
- **Janvier:** Bug injection and verification, Verification plan
- **Sal Esmael:** Bug injection and verification, Verification Plan

9 Schedule

9.1 Create a table with a plan of completion.

Table 1: Project Completion Plan

Milestone	Description of Activities	Deliverables	Due	Status
MS1	• Document preparation • structuring files and git • Design implementation and conventional TB • Division of tasks	• Design Spec Doc • Initial Verification Plan • Initial Design Implementation • Conventional Testbench	Jan 30	Completed
MS2 & MS3	• Class-based verification • Defining components: Transaction, Generator, Driver, Monitors, Scoreboard and Coverage • work on code and functional coverage • Debugging error	• Complete RTL • Class-based Testbench • Coverage report • Updated Verification Plan • Report for TB	Feb 18	Completed
MS4	• Develop UVM Testbench • Study UVM architecture, hierarchy & components • Add a section for UVM verification Plan • use UVM_MESSAGING, UVM_LOGGING mechanisms to create and log reports and data.	• Complete RTL • UVM Testbench • Coverage report • Updated Verification Plan • Report for TB	Feb 27	Not Started
MS5	• Complete the UVM architecture, UVM environment and UVM testbench • Complete all the testcases and obtain coverage reports. • Create few scenarios of bug-injection and verify. • Preparing presentations and finalize documents. Row 4, Col 2	• Complete UVM Architecture • Coverage report • Finalized Documents • Presentation	Mar 14	Not Started

10 Reference uses/Citations/Acknowledgments

10.1 Project GitHub

Github link: `https://github.com/jmpfizi-rutihunza/ece593_g9_final_project`

10.2 Reference

10.3 Acknowledgment