VERIFICATION TEST PLAN

ECE-593: Fundamentals of Pre-Silicon Validation

Maseeh College of Engineering and Computer Science

Winter, 2025

Project Name: UMayVmyISA

Members: Riley Cox, Nathan Fraly, Jeff McHale

Date: January 19, 2026

Project Link: https://github.com/nfraly/UMayVMyISA

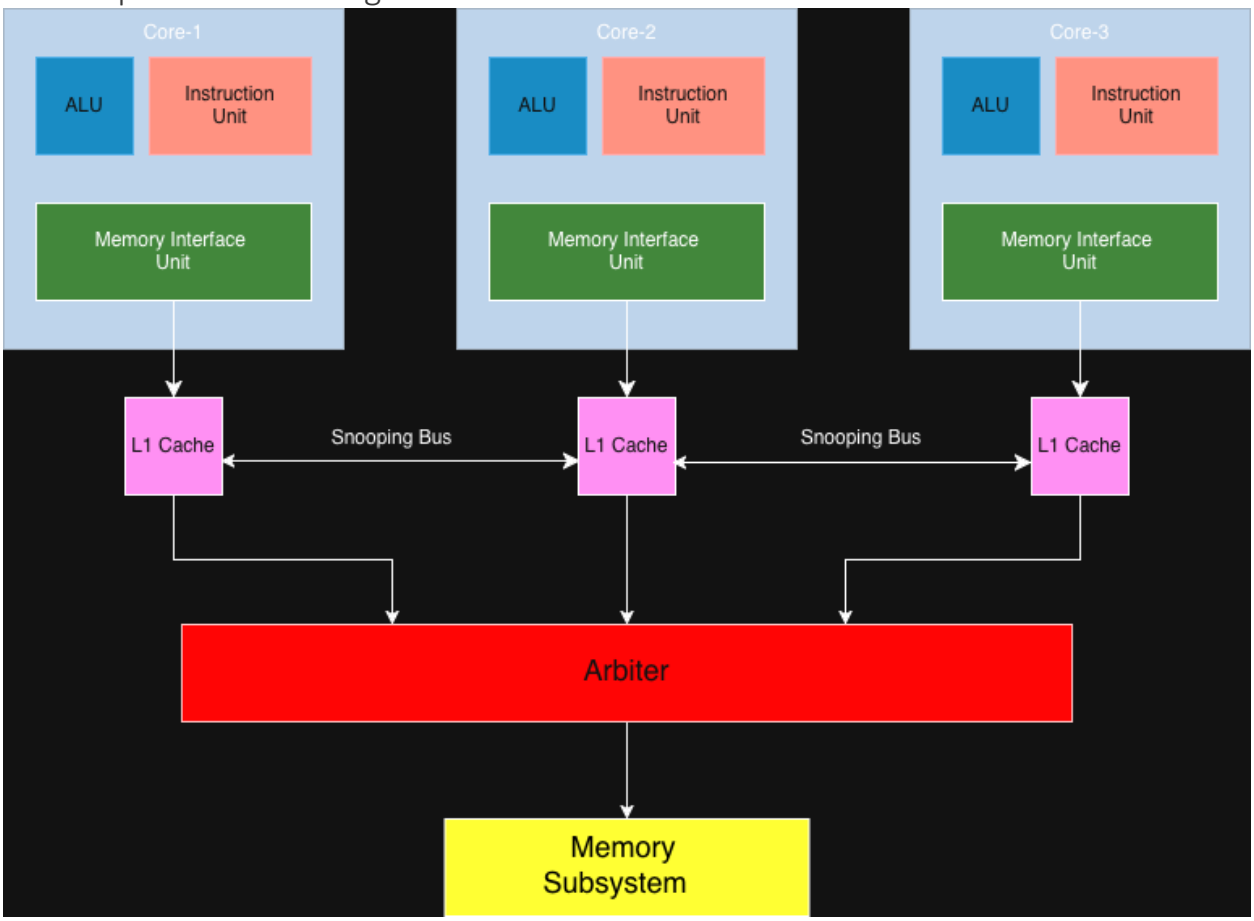
1 Table of Contents

2 Introduction:

2.1 Objective of the verification plan

This verification test plan will describe the way in which the project will be tested and confirmed that it meets the specifications. The test plan provides a clear path for the team to follow when verifying the design and allows others to see the methods we used to verify.

2.2 Top Level block diagram



2.3 The MultiProcessorSystem design has the following specifications:

- Memory Size: 2KB
- Width of Memory: 1 Byte
- Depth of Memory: 2K
- Number of Processors: 3

3 Verification Requirements

3.1 Verification Levels

3.1.1 What hierarchy level are you verifying and why?

The design will be verified at an L2 level to ensure that bugs can be identified early in the design cycle, and the cause of the bugs can be fixed easily before implementing in the larger overall design.

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

Verifying at this level allows us to catch and fix bugs early before implementing them in the larger blocks

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

4 Required Tools

4.1 List of required software and hardware toolsets needed.

Siemen's QuestaSim software is required to compile and simulate the design

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

Our directory structure

5 Risks and Dependencies

5.1 List all the critical threats or any known risks. List contingency and mitigation plans.

Nathan's military obligations come at very inconvenient times. To mitigate task fall-through, Nathan is a backup on all project-critical tasks until after his Annual Training ends

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

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

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

7 Tests and Methods

7.1.1 Testing methods to be used: Black/White/Gray Box.

For this design we have decided to use the gray box testing method.

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

A Black box would allow us to focus solely on the functionality of the design laid out by the specification. Though not knowing about the internal workings of the design may lead to redundancy in verification.

White box testing allows us to peer into the design and base our testing on the internal workings. The disadvantage of this is if the design has any changes, the testing could break.

Gray box testing combines aspects of black box and white box testing. The testing done can focus on functionality while also using knowledge of internal workings to reduce testing redundancy. By choosing gray box testing, we can verify our designs functionality and use knowledge of internal logic to cut out wasteful testing and time.

7.1.3 Testbench Architecture; Component used (list and describe Drivers, Monitors, scoreboards, checkers etc.)

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

This project will use dynamic simulation as its primary verification strategy. Formal verification does not fit this project due to the complexity of the design.

7.1.5 What is your driving methodology?

7.1.5.1 List the test generation methods (Directed test, constrained random)

Both directed test cases and constrained random test cases will be used. Directed test cases will be used for corner cases, while constrained random will handle the bulk of the testing.

7.1.6 What will be your checking methodology?

7.1.6.1 From specification, from implementation, from context, from architecture etc

7.1.7 Testcase Scenarios (Matrix)

7.1.7.1 Basic Tests

Test Name / Number	Test Description/ Features
1.1.1 Fetch Testing	Verify that the fetch module fetches the correct instruction. If given many instructions, fetch module should fetch the first instruction and increment the PC to fetch the next instruction.

1.1.2 Decode Testing	Verify that the decode stage is grabbing the correct registers and accessing the correct modules needed for executing the instruction.
1.1.3 ALU Testing	ALU performs all the operations required by the specification
1.1.4 Arbiter Testing	The round robin arbiter should work properly; assertions can be easily implemented to ensure proper functionality.
1.1.5 Memory Subsystem Testing	CPU core should be able to properly read to and write from memory
1.1.6 Cache Testing	Private L1 caches should adhere to MESI protocol

7.1.7.2 Complex Tests

Test Name / Number	Test Description/ Features
1.2.1 CPU sequence	CPU core should be able to fetch, decode, and execute a series of instructions
1.2.2	

7.1.7.3 Regression Tests (Must pass every time)

Test Name / Number	Test Description/Features
1.3.1	Tests that should always pass
1.3.2	

7.1.7.4 Any special or corner cases testcases

Test Name / Number	Test Description
1.4.1	Special Case testing tests and conditions
1.4.2	Bug injection and testing scenario

8 Coverage Requirements

8.1.1.1 Describe Code and Functional Coverage goals for the DUV

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

8.1.2 Assertions

8.1.2.1 Describe the assertions that you are planning to use and how it will help you improve the overall coverage and functional aspects of the design.

Assertions can be easily implemented for the round-robin arbiter to ensure only one core is granted at a time for example.

9 Resources requirements

9.1 Team members and who is doing what and expertise.

All responsibilities are subject to change based on new information learned about the project and methodologies learned in class.

Nathan Fraly - Implementation of classes, DVV plan. Verilog-based project experience, Assertion-based verification course complete. Has used covergroups in TB's previously

Riley Cox - Theory of Verification plan; what bins, coverpoints, covergroups, etc. Verilog-based project experience, Assertion-based verification course complete. Has used covergroups in TB's previously

Jeff McHale – RTL Implementation, HLDS. Has completed intro to SystemVerilog

All – Communicate early and often about blocks, knowledge gaps, and continue to evolve testbench and RTL design as new methodologies are discovered (through Venkatesh the Great One)

10 Schedule

10.1 Create a table with a plan of completion. You can use milestones as a guide to fill this.

Who→ When↓	Nathan Fraly	Riley Cox	Jeff McHale	Everyone	
Milestone 1 Due Date	Logistics Portion of Verification Plan, Create simple ALU testbench, Assist with early RTL design	Generate Long-Term Verification Plan, Generate Fetch/Decode Testbench	Generate first draft HLDS, Create fetch/decode/ALU RTL	Use revision control through github to ensure proper documentation IAW project specs	
Milestone 2 Due Date	Generate testbenches for new RTL models, update DVV plan and schedule, assist with RTL	Revise Coverpoints, groups, and bins, generate run.do, format print statements,	Create memory/writeback RTL, Update HLDS, fix bugs as found	^	
Milestone 3 Due Date	Begin and finish class-based TB dev based on DVV Spec	Class-based TB dev w/ Nathan	Generate Cache and Arbiter RTL, generate system-level RTL	^	
Milestone 4 Due Date	Implement UVM TB	Update DVV plan with UVM requirements	Fix bugs as found, help with UVM implementation	^	
Milestone 5 Due Date	Finish UVM TB. Add bug-injection	Finish UVM TB, Finalize test document and submit any waivers	Finish UVM TB, prepare project presentation	Present to Venkatesh the Great One	

11 References Uses / Citations/Acknowledgements