VERIFICATION TEST PLAN

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

Project Name: Verification of AXI4 Lite Protocol

Members: Nithesh Kamireddy
Vyshnavi Julakanti
Satyam Sharma
Sarath Chandra Jampani

Date: 5/22/2025

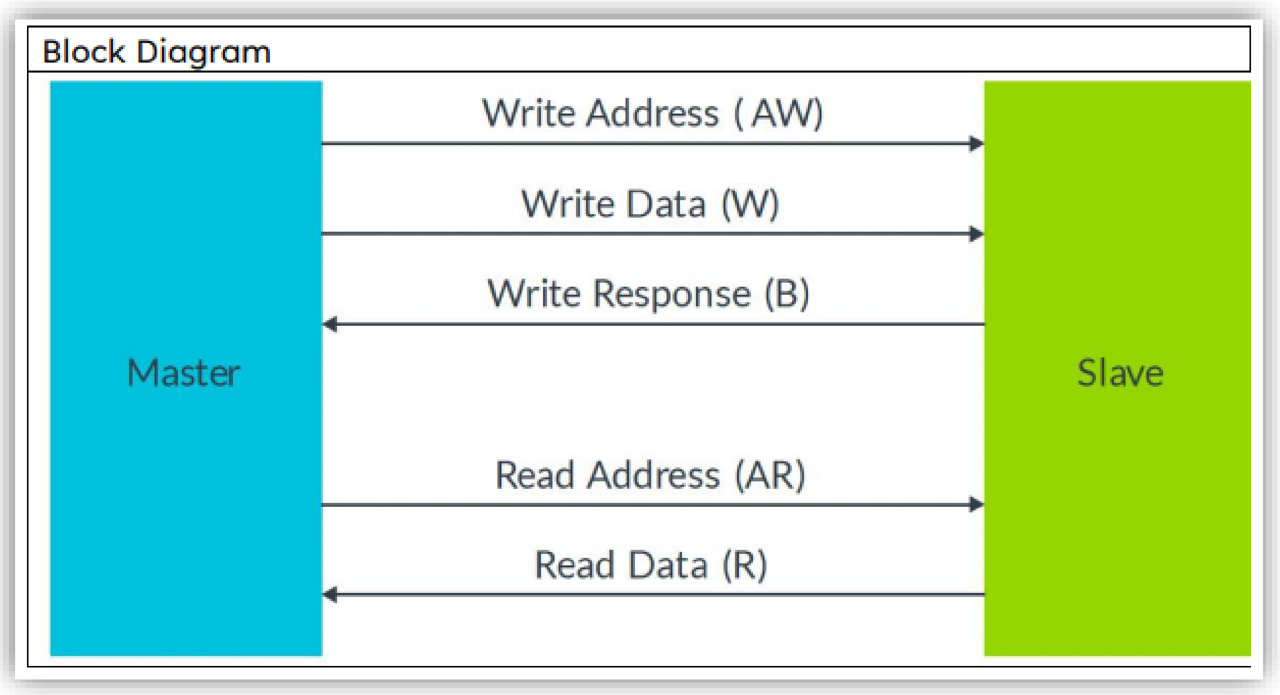
1 Table of Contents

2 Introduction:

2.1 Objective of the verification plan

To functionally verify an AXI4-Lite slave interface using a class-based SystemVerilog testbench. The focus is on validating protocol handshakes, memory-mapped register access, concurrent channel behavior, and byte-level write correctness via WSTRB.

2.2 Top Level block diagram



2.3 Specifications for the design

- **AXI4-Lite compliant (AW, W, B, AR, R channels)**
- **32-bit data path**
- **4 registers, addressable via 2-bit MSB of address**
- **Byte-wise writes via 4-bit WSTRB**
- **Active-low reset (ARESETn), synchronous ACLK**
- **Memory-mapped access**

3 Verification Requirements

3.1 Verification Levels

3.1.1 What hierarchy level are you verifying and why?

- RTL module level – to validate the complete slave protocol behavior.

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

- Full controllability via constrained-random drivers; observability via monitors and reference memory model.

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

- Clearly defined: AXI4-Lite signals – AWADDR, WDATA, WSTRB, ARADDR, RDATA, and all valid/ready signals.

4 Required Tools

4.1 List of required software and hardware toolsets needed.

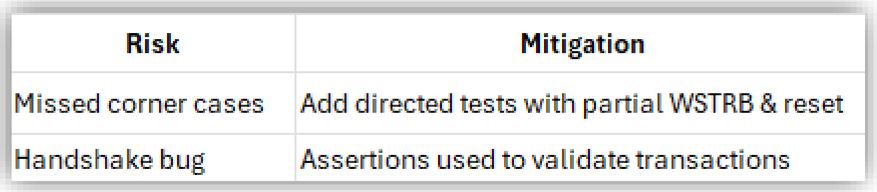
- **Simulator:** QuestaSim
- **Language:** SystemVerilog (OOP-based TB)
- **Viewer:** GTKWave
- **Version Control:** GitHub

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

- /rtl – RTL design
- /tb – testbench code
- /logs – simulation logs
- /scripts – run/compile scripts

5 Risks and Dependencies

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

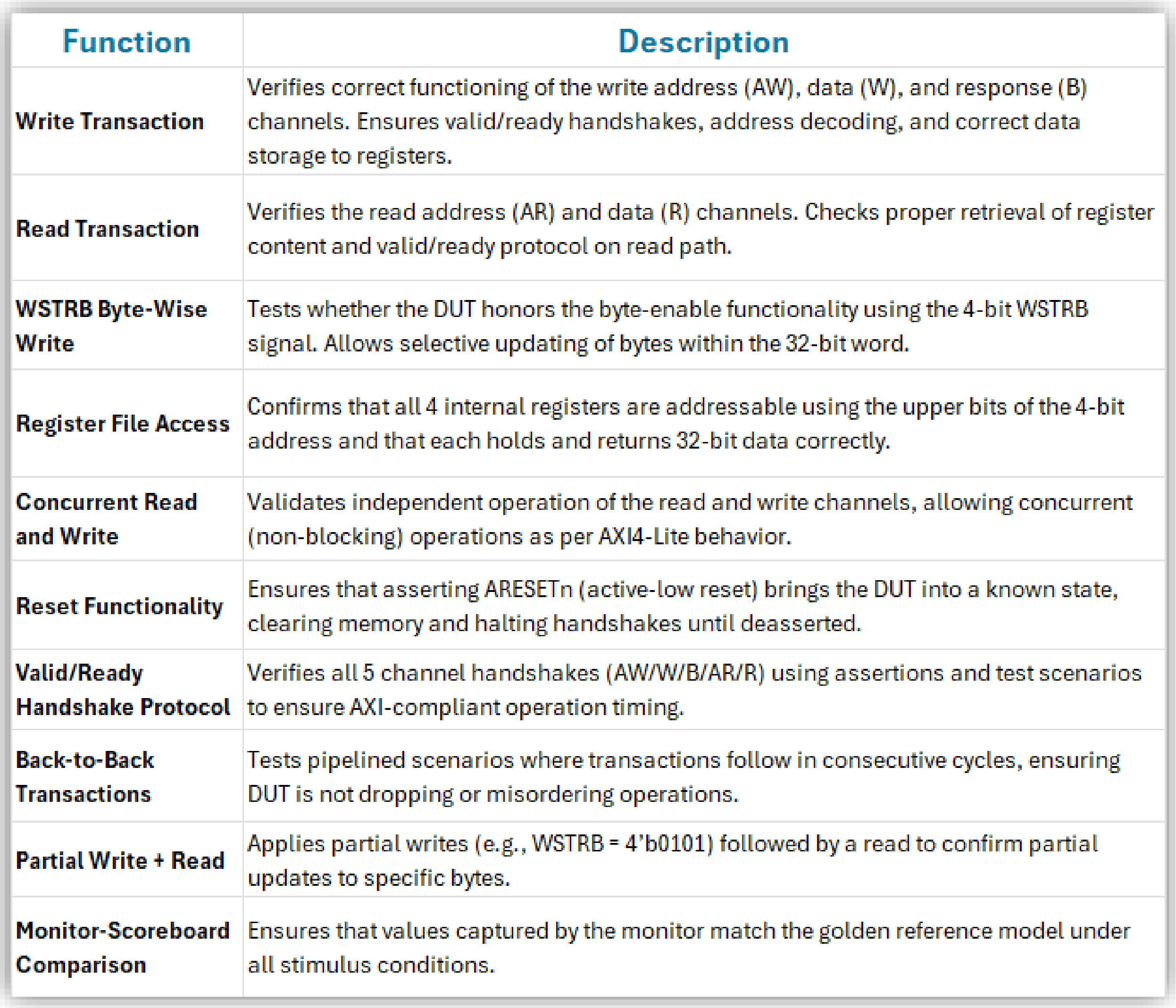


Risk	Mitigation
Missed corner cases	Add directed tests with partial WSTRB & reset
Handshake bug	Assertions used to validate transactions

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



Function	Description
Write Transaction	Verifies correct functioning of the write address (AW), data (W), and response (B) channels. Ensures valid/ready handshakes, address decoding, and correct data storage to registers.
Read Transaction	Verifies the read address (AR) and data (R) channels. Checks proper retrieval of register content and valid/ready protocol on read path.
WSTRB Byte-Wise Write	Tests whether the DUT honors the byte-enable functionality using the 4-bit WSTRB signal. Allows selective updating of bytes within the 32-bit word.
Register File Access	Confirms that all 4 internal registers are addressable using the upper bits of the 4-bit address and that each holds and returns 32-bit data correctly.
Concurrent Read and Write	Validates independent operation of the read and write channels, allowing concurrent (non-blocking) operations as per AXI4-Lite behavior.
Reset Functionality	Ensures that asserting ARESETn (active-low reset) brings the DUT into a known state, clearing memory and halting handshakes until deasserted.
Valid/Ready Handshake Protocol	Verifies all 5 channel handshakes (AW/W/B/AR/R) using assertions and test scenarios to ensure AXI-compliant operation timing.
Back-to-Back Transactions	Tests pipelined scenarios where transactions follow in consecutive cycles, ensuring DUT is not dropping or misordering operations.
Partial Write + Read	Applies partial writes (e.g., WSTRB = 4'b0101) followed by a read to confirm partial updates to specific bytes.
Monitor-Scoreboard Comparison	Ensures that values captured by the monitor match the golden reference model under all stimulus conditions.

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

- Protocol errors or BRESP codes (out of scope for AXI-Lite)
- Unaligned access (not specified in DUT)

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

- Critical: All five protocol channels, memory behavior
- Non-Critical: BRESP, debug features (not present)

7 Tests and Methods

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

- **White-box** (RTL access + internal observation)
- White-box testing was chosen because the DUT is custom RTL (developed by the same team), allowing internal signal observation and control.

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

Pros

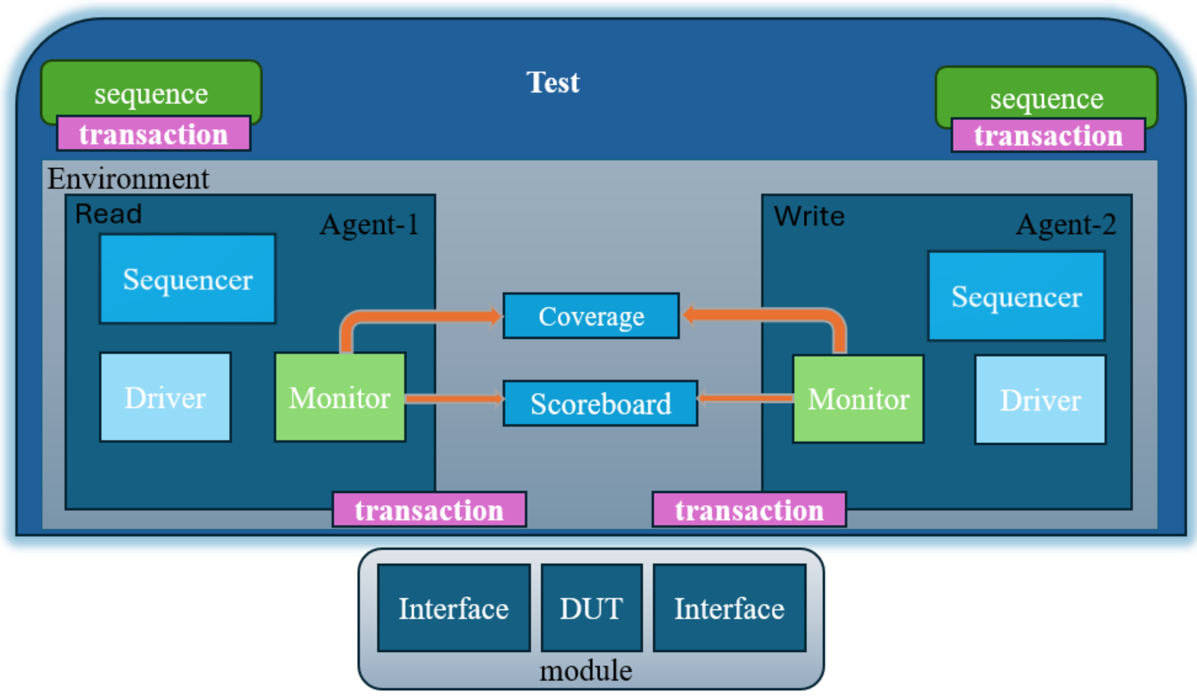
- **Precise debugging:** Internal signals such as register contents and handshake states can be directly monitored, aiding rapid identification of issues.
- **Efficient development:** Design bugs, especially those related to protocol handshakes or register access, can be isolated quickly without reliance on indirect observation.
- **Strong scoreboard alignment:** Full visibility into the design allows tight synchronization between expected behavior (reference model) and observed outputs.

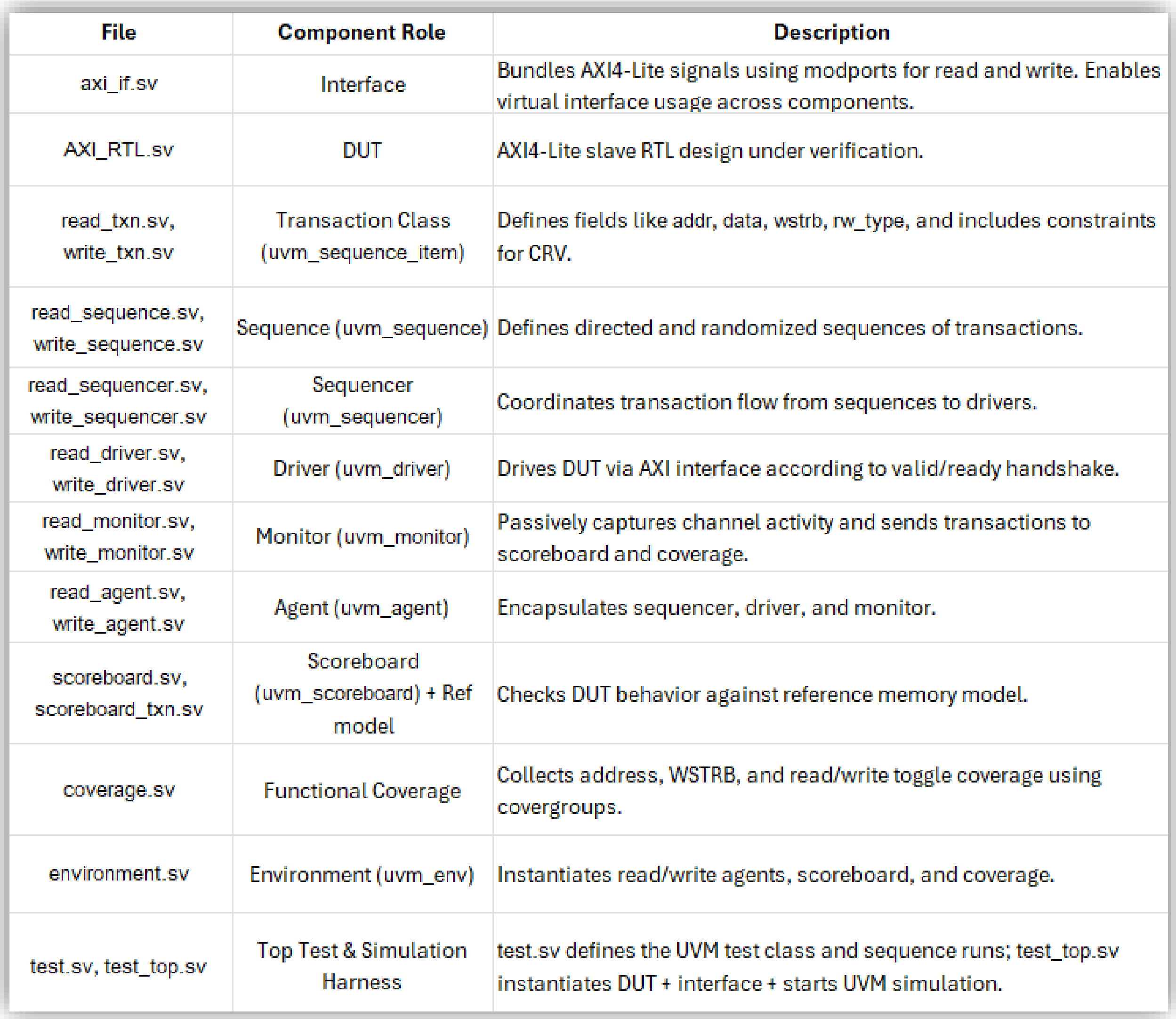
Cons

- **Requires detailed RTL understanding:** Verification engineers must be familiar with internal register mapping, reset behavior, and signal timing.
- **High testbench coupling:** Modifications to the design may necessitate updates in drivers, monitors, or scoreboard components due to tight integration.
- **Limited abstraction:** In contrast to reusable UVM Verification IP (VIP), this environment is tailored specifically to the current design, reducing portability and reuse for other DUTs.

- Selected white-box testing because we have a clear understanding of how the AXI4-Lite slave module works internally. This approach lets us directly observe key internal signals like register states and handshake behavior, which makes debugging much easier and faster. It also helped us build a reliable testbench using both directed and constrained-random tests. With most of the checking handled through a functional scoreboard, and a few targeted assertions for protocol handshakes, white-box testing gave us the visibility and control we needed to fully verify the design without adding unnecessary complexity.

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





File	Component Role	Description
axi_if.sv	Interface	Bundles AXI4-Lite signals using modports for read and write. Enables virtual interface usage across components.
AXI_RTL.sv	DUT	AXI4-Lite slave RTL design under verification.
read_txn.sv, write_txn.sv	Transaction Class (uvm_sequence_item)	Defines fields like addr, data, wstrb, rw_type, and includes constraints for CRV.
read_sequence.sv, write_sequence.sv	Sequence (uvm_sequence)	Defines directed and randomized sequences of transactions.
read_sequencer.sv, write_sequencer.sv	Sequencer (uvm_sequencer)	Coordinates transaction flow from sequences to drivers.
read_driver.sv, write_driver.sv	Driver (uvm_driver)	Drives DUT via AXI interface according to valid/ready handshake.
read_monitor.sv, write_monitor.sv	Monitor (uvm_monitor)	Passively captures channel activity and sends transactions to scoreboard and coverage.
read_agent.sv, write_agent.sv	Agent (uvm_agent)	Encapsulates sequencer, driver, and monitor.
scoreboard.sv, scoreboard_txn.sv	Scoreboard (uvm_scoreboard) + Ref model	Checks DUT behavior against reference memory model.
coverage.sv	Functional Coverage	Collects address, WSTRB, and read/write toggle coverage using covergroups.
environment.sv	Environment (uvm_env)	Instantiates read/write agents, scoreboard, and coverage.
test.sv, test_top.sv	Top Test & Simulation Harness	test.sv defines the UVM test class and sequence runs; test_top.sv instantiates DUT + interface + starts UVM simulation.

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

Dynamic Simulation using UVM

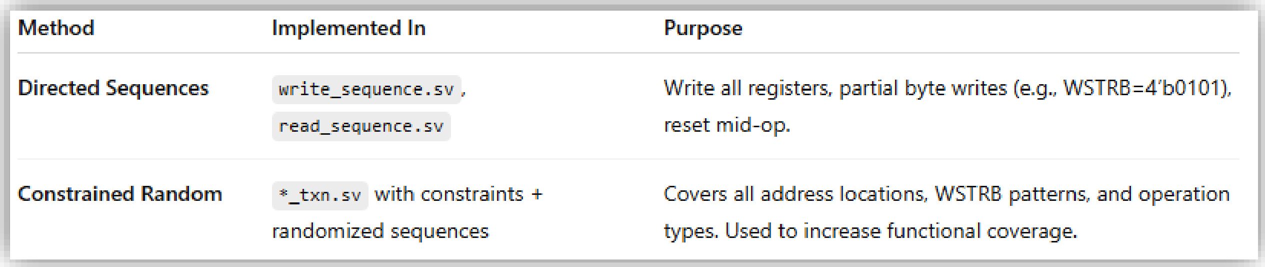
Justification:

- Enables concurrent testing of AXI read/write channels.
- Supports both directed a nd randomized stimulus using sequences.
- Allows modular component reuse and structured messaging via uvm_report_server.
- Functional coverage, scoreboard checking, and simulation control are tightly integrated into the environment.
- Provides waveform generation and debug visibility through QuestaSim and GTKWave.

7.1.5 What is your driving methodology?

- All stimulus is generated through UVM sequences and driven to the DUT via sequencers and drivers.
- Read Agent Flow:
- read_sequence.sv creates read-only transactions.
- read_sequencer.sv issues them to the read_driver.sv.
- Read driver asserts ARVALID/ARADDR and waits for handshake.
- Write Agent Flow:
- write_sequence.sv creates transactions with addr, data, and wstrb.
- write_driver.sv drives AWVALID/WVALID and waits for response via BVALID.

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



Method	Implemented In	Purpose
Directed Sequences	`write_sequence.sv`, `read_sequence.sv`	Write all registers, partial byte writes (e.g., WSTRB=4'b0101), reset mid-op.
Constrained Random	`*_txn.sv` with constraints + randomized sequences	Covers all address locations, WSTRB patterns, and operation types. Used to increase functional coverage.

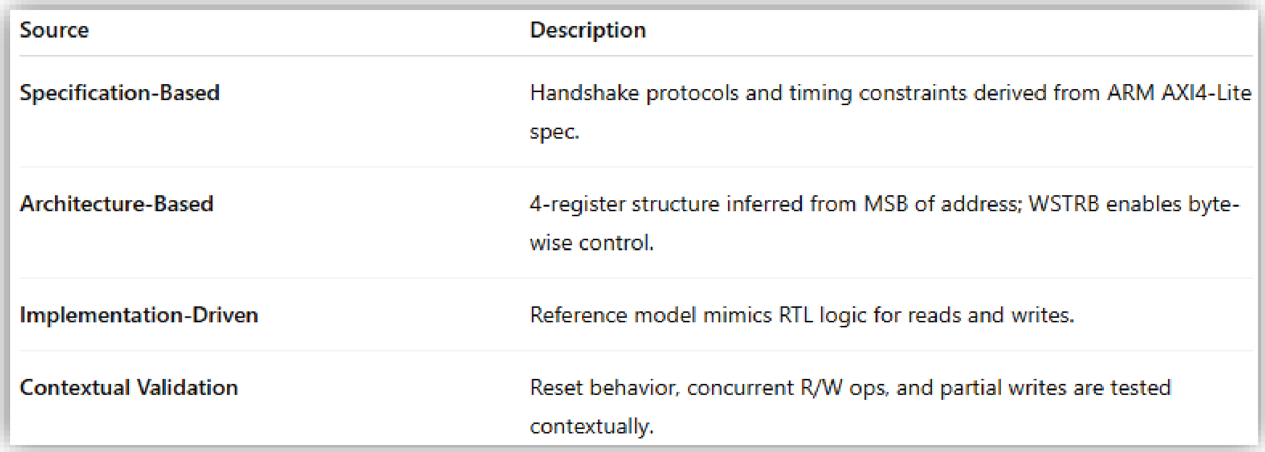
7.1.6 What will be your checking methodology?

- The DUT output is checked via a scoreboard that compares monitor-observed results to expected values generated by a reference memory model.

Data Flow:

- Monitors capture AXI channel activity and forward it to scoreboard.sv.
- scoreboard_txn.sv helps organize expected vs actual result comparison.
- Ref model updates memory on write and returns expected value on read.

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



Source	Description
Specification-Based	Handshake protocols and timing constraints derived from ARM AXI4-Lite spec.
Architecture-Based	4-register structure inferred from MSB of address; WSTRB enables byte-wise control.
Implementation-Driven	Reference model mimics RTL logic for reads and writes.
Contextual Validation	Reset behavior, concurrent R/W ops, and partial writes are tested contextually.

7.1.7 Testcase Scenarios (Matrix)

7.1.7.1 Basic Tests

Test Name / Number	Test Description/ Features
1.1.1	Perform a basic read from a single register. Validates ARVALID/ARREADY and correct RDATA output.
1.1.2	Write to a register with WSTRB = 4'b1111 (all bytes valid). Ensures full-word write is stored correctly.
1.1.3	Apply reset during operation. Check that DUT clears internal state and halts valid transactions.
1.1.4	Write and read to/from all 4 registers (address range 0x0 to 0xC). Ensures full address decoding is functional.

7.1.7.2 Complex Tests

Test Name / Number	Test Description/ Features
1.2.1	Concurrent read and write operations. Verifies independent channel operation and valid/ready handshakes on both sides. Checks for coherency when accessing the same register.
1.2.2	Perform a partial write using WSTRB = 4'b0101 followed by an immediate read. Verifies byte-level write masking and correct data merging in the target register.

7.1.7.3 Regression Tests (Must pass every time)

Test Name / Number	Test Description/Features
1.3.1	Sequence of read/write operations with randomized values. Confirms overall DUT stability and correctness under repeated simulation runs.
1.3.2	Perform reset between multiple back-to-back transactions. Verifies consistent DUT recovery and output under reset stress.

7.1.7.4 Any special or corner cases testcases

Test Name / Number	Test Description
1.4.1	Special case: Write with WSTRB = 4'b0001 (LSB byte only), then read back. Confirms that only byte[7:0] is updated and others remain unchanged.
1.4.2	Inject backpressure by holding READY low for multiple cycles. Tests DUT's protocol tolerance, especially during handshake delays.

8 Coverage Requirements

8.1.1.1 Describe Code and Functional Coverage goals for the DUV

The goal is to achieve:

- **100% code coverage** for the DUT (AXI_RTL.sv) including:
  - All if/else, case, and FSM transitions
  - Conditions around WSTRB, read/write paths, and reset logic
- **95%+ functional coverage** through covergroups focused on:
  - All valid register addresses (4 total)
  - All 4-bit WSTRB byte combinations
  - Operation type: read, write
  - Read-after-write scenarios
  - Reset followed by operation
  - Channel activity coverage (valid/ready assertions per channel)

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

- Functional coverage is implemented using a dedicated coverage.sv file. Covergroups are sampled inside monitors and optionally inside drivers for constrained-random behavior. Coverage reports are extracted post-simulation using tool-generated output.

Covergroups and Coverpoints:

Covergroup Name	Coverpoints	Bins	Description
cg_address	addr	4 bins for 0x0, 0x4, 0x8, 0xC	Ensures all 4 register addresses are accessed
cg_wstrb	wstrb	16 bins (all possible 4-bit combinations)	Checks full and partial byte-wise writes
cg_op_type	rw_type	2 bins: read, write	Confirms both operation types are exercised
cg_reset_sequence	reset followed by op	1 bin: reset → write/read	Confirms proper DUT behavior after reset
cg_interleaved	concurrent read/write trigger	1 bin: active overlap	Ensures both agents operate simultaneously at least once

How Coverage is Sampled:

- write_monitor samples WSTRB and address
- read_monitor samples address and operation type
- Reset-based covergroup sampled in test sequences

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.

- To enhance protocol validation, a **minimal set of AXI4-Lite protocol assertions is planned to be implemented** as part of the verification environment. These assertions will be embedded within the interface (axi_if.sv) or integrated into the monitor components for runtime protocol compliance checking.

Purpose of Assertions:

- **Catch protocol violations early** during simulation
- **Validate handshake behavior** between valid and ready signals
- **Improve observability** without relying solely on waveform inspection
- **Complement functional coverage** by enforcing timing relationships

Planned Assertions:

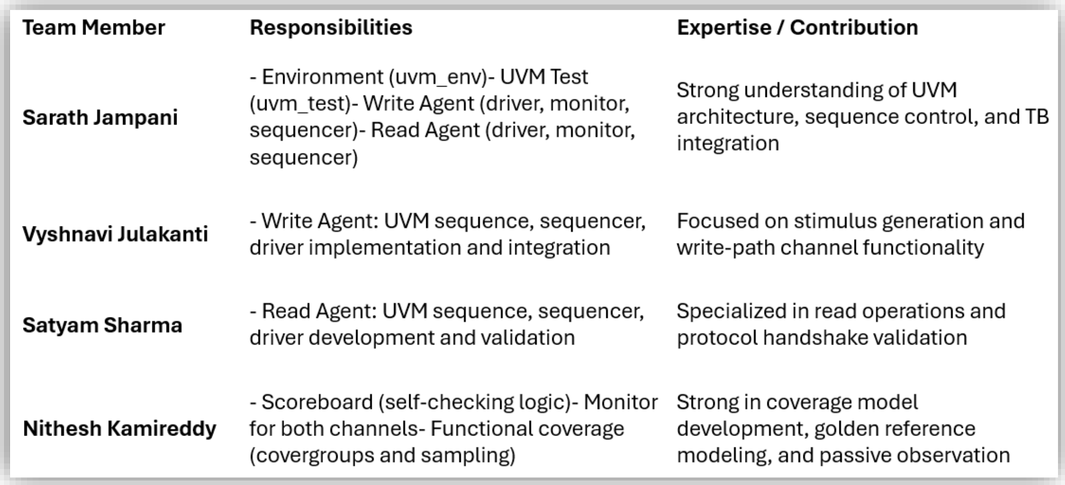
Assertion Name	Expression	Purpose
assert_aw_handshake	`AWVALID	-> ##1 AWREADY`
assert_w_handshake	`WVALID	-> ##1 WREADY`
assert_ar_handshake	`ARVALID	-> ##1 ARREADY`
assert_r_handshake	`RVALID	-> ##1 RREADY`

Impact on Verification Quality:

- These assertions will help enforce **protocol correctness**, especially under random and interleaved traffic. While not directly contributing to coverage metrics, they serve as critical checks to prevent handshake mismatches or timing bugs that might otherwise go undetected.

9 Resources requirements

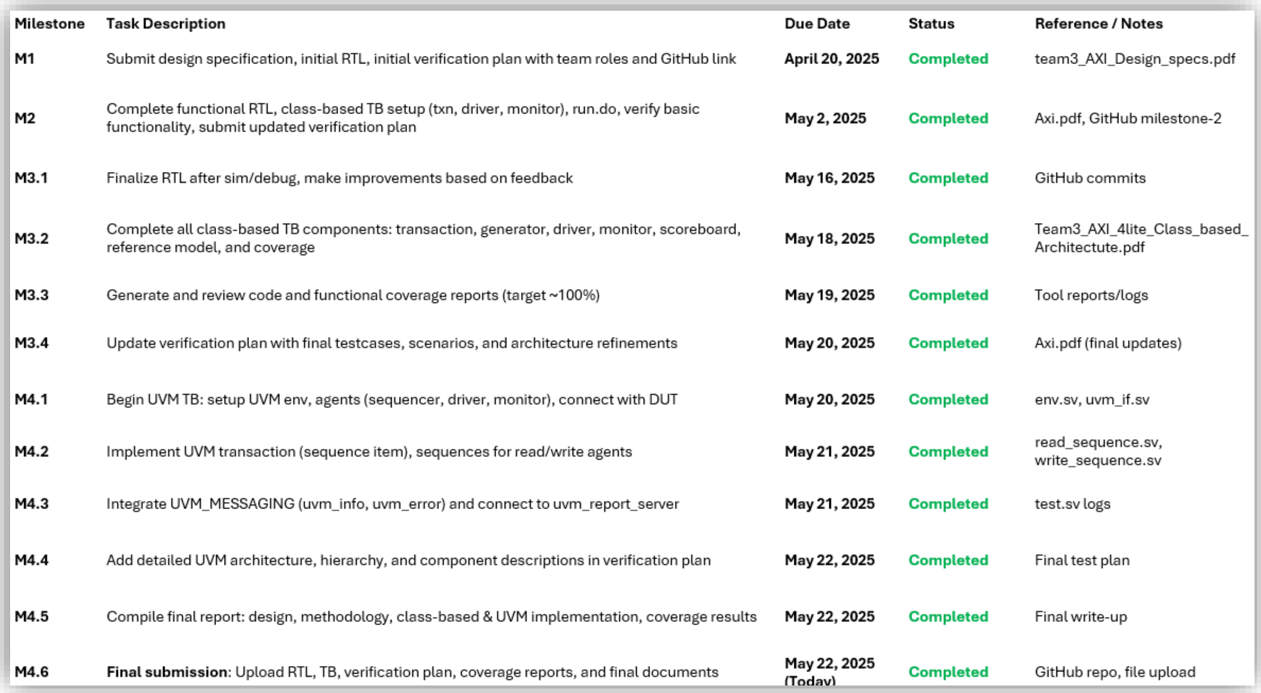
9.1 Team members and who is doing what and expertise.



Team Member	Responsibilities	Expertise / Contribution
Sarath Jampani	- Environment (uvm_env)- UVM Test (uvm_test)- Write Agent (driver, monitor, sequencer)- Read Agent (driver, monitor, sequencer)	Strong understanding of UVM architecture, sequence control, and TB integration
Vyshnavi Julakanti	- Write Agent: UVM sequence, sequencer, driver implementation and integration	Focused on stimulus generation and write-path channel functionality
Satyam Sharma	- Read Agent: UVM sequence, sequencer, driver development and validation	Specialized in read operations and protocol handshake validation
Nithesh Kamireddy	- Scoreboard (self-checking logic)- Monitor for both channels- Functional coverage (covergroups and sampling)	Strong in coverage model development, golden reference modeling, and passive observation

10 Schedule

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



Milestone	Task Description	Due Date	Status	Reference / Notes
M1	Submit design specification, initial RTL, initial verification plan with team roles and GitHub link	**April 20, 2025**	Completed	team3_AXI_Design_specs.pdf
M2	Complete functional RTL, class-based TB setup (txn, driver, monitor), run.do, verify basic functionality, submit updated verification plan	**May 2, 2025**	Completed	Axi.pdf, GitHub milestone-2
M3.1	Finalize RTL after sim/debug, make improvements based on feedback	**May 16, 2025**	Completed	GitHub commits
M3.2	Complete all class-based TB components: transaction, generator, driver, monitor, scoreboard, reference model, and coverage	**May 18, 2025**	Completed	Team3_AXI_4lite_Class_based_ Architectute.pdf
M3.3	Generate and review code and functional coverage reports (target ~100%)	**May 19, 2025**	Completed	Tool reports/logs
M3.4	Update verification plan with final testcases, scenarios, and architecture refinements	**May 20, 2025**	Completed	Axi.pdf (final updates)
M4.1	Begin UVM TB: setup UVM env, agents (sequencer, driver, monitor), connect with DUT	**May 20, 2025**	Completed	env.sv, uvm_if.sv
M4.2	Implement UVM transaction (sequence item), sequences for read/write agents	**May 21, 2025**	Completed	read_sequence.sv, write_sequence.sv
M4.3	Integrate UVM_MESSAGING (uvm_info, uvm_error) and connect to uvm_report_server	**May 21, 2025**	Completed	test.sv logs
M4.4	Add detailed UVM architecture, hierarchy, and component descriptions in verification plan	**May 22, 2025**	Completed	Final test plan
M4.5	Compile final report: design, methodology, class-based & UVM implementation, coverage results	**May 22, 2025**	Completed	Final write-up
M4.6	**Final submission**: Upload RTL, TB, verification plan, coverage reports, and final documents	**May 22, 2025 (Today)**	Completed	GitHub repo, file upload

11 References Uses / Citations/Acknowledgements

- **ARM AMBA AXI4-Lite Specification**
  https://developer.arm.com/documentation/ihi0022/latest/
- **SystemVerilog IEEE 1800-2017 Standard**
- **UVM Class Reference Manual (UVM 1.2)**
- **ECE-593 Course Materials**
  Portland State University, Spring 2025 — Instructor: Prof. Venkatesh Patil
- Melikyan, V., Harutyunyan, S., Kirakosyan, A., & Kaplanyan, T. (2021).
  UVM Verification IP for AXI.
  In **2021 IEEE East-West Design & Test Symposium (EWDTS)**, Batumi, Georgia, pp. 1–4.
  DOI: 10.1109/EWDTS52692.2021.9580997
  Keywords: AXI, UVM, Verification, Functional Coverage, SystemVerilog, IP Design
- Sangani, H., & Mehta, U. (2022).
  UVM-Based Verification of Read and Write Transactions in AXI4-Lite Protocol.
  In **2022 IEEE Region 10 Symposium (TENSYMP)**, Mumbai, India, pp. 1–5.
  DOI: 10.1109/TENSYMP54529.2022.9864552
  Keywords: AXI4-Lite, UVM, Read/Write Protocol, SoC Verification, Simulation, SystemVerilog

GitHub Repository:
https://github.com/nitheshkamireddy7/team_3_AXI4_lite_Verfication

Contributors Acknowledged:
All team members equally contributed to implementation, integration, and documentation across various verification phases.