VERIFICATION TEST PLAN

ECE-593: Fundamentals of Pre-Silicon Validation
Maseeh College of Engineering and Computer Science
Winter, 2026

Project Name: Design and Verification of AHB to APB Bridge

Members:

Amey Kulkarni - 933959421

Annapoorna Hamsagaru Manjunath – 984436960

Date: 03/31/2026

Table of Contents

1 Introduction:

The purpose of this verification plan is to outline the approach and verification methodology used to validate the functionality of the Advanced High-Performance Bus (AHB) to Advanced Peripheral Bus (APB) bridge. The AHB-to-APB bridge plays a vital role within the AMBA (Advanced Microcontroller Bus Architecture) framework by facilitating reliable data transfer between high-performance AHB master components and low-speed APB peripheral devices.

Our verification plan is designed to cover the different aspects of the bridge's functionality. We will be using Synopsys VCS to run the simulation and SystemVerilog (SV) and SystemVerilog Assertions (SVA) to write the testbench in the first checkpoint and later on convert the SV testbench to UVM testbench for the final submission.

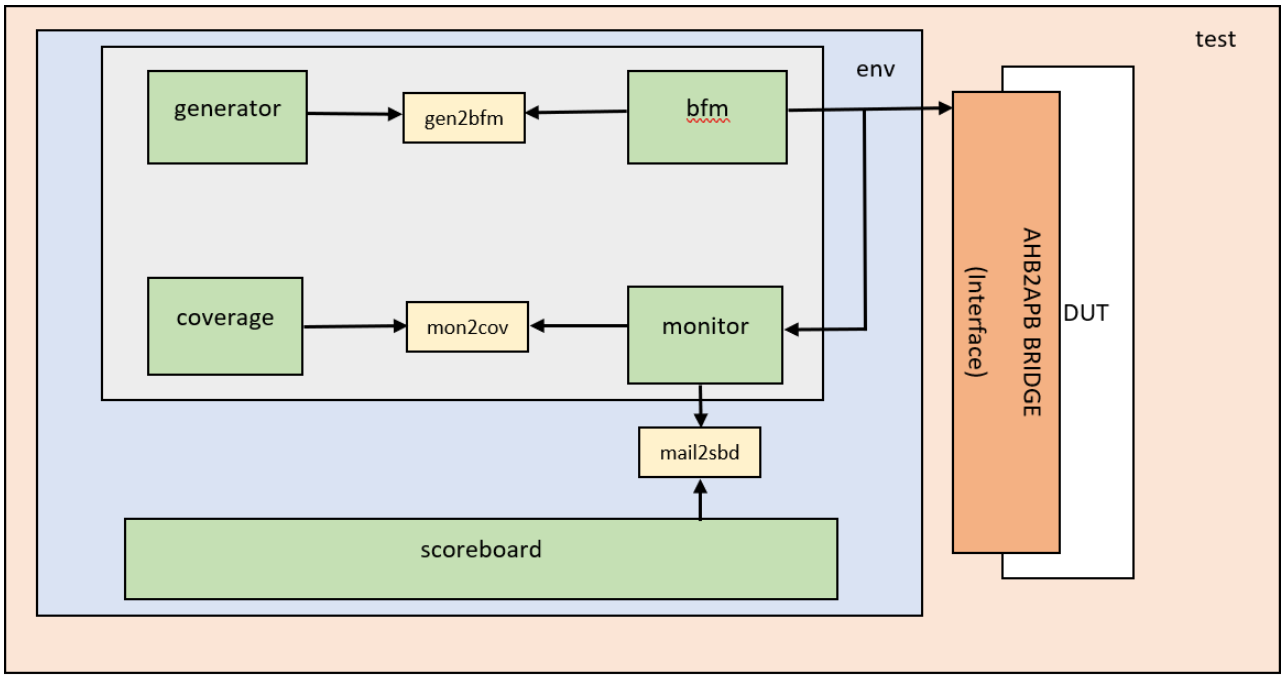
1.1 Objective of the verification plan

The dummy AHB Master module contains two separate tasks, one responsible for performing read transactions and the other for executing write transactions. These tasks abstract the read and write functionality into reusable procedural blocks, making interactions with the design under test (DUT) simpler and more structured.

Along with these tasks, a self-checking testbench has been implemented to verify the correctness of the design. The integration of the read and write tasks with the self-checking mechanism enables automated and dependable validation of the design's behavior across a variety of test scenarios.

1.2 Top Level block diagram

SV based ENV-



1.3 Specifications for the design

- The AHB-to-APB bridge design incorporates a simple clock domain crossing (CDC) mechanism between the APB controller and the APB bus to ensure safe and reliable signal transfer across differing clock domains. This CDC logic is implemented to maintain data integrity and proper synchronization when control and data signals traverse from the controller domain to the APB peripheral domain.
- The AHB-to-APB bridge constitutes the sole synthesizable component of this project, with all interface signals implemented in strict accordance with the specified block diagram. As the AHB protocol supports pipelined transfers whereas the APB protocol operates in a non-pipelined manner, the bridge performs protocol translation by introducing appropriate wait states to align the timing requirements of the two buses. This translation mechanism is controlled by a Finite State Machine (FSM) as defined in the design specification.
- The scope of this verification effort is confined to validating single read and single write transactions, as well as a basic burst transfer mode with address incrementation by four bytes.

2 Verification Requirements

2.1 Verification Levels

The verification process will be carried out at two levels: block (module) level and top level. At the block level, individual components/modules—such as the AHB Master, APB FSM Controller, and APB Slave—will be verified independently. At the top level, the integrated AHB-APB Bridge will be verified as a complete system.

2.1.1 Interfaces and specifications with signals

a. The design has clean interfaces between the AHB bus, APB bus, and the bridge. Each module has clearly defined inputs and outputs, and the data flow between modules is well-structured.

b. The specification of the design is clear, with each module's functionality and interaction with other modules well-defined. The FSM in the APB_FSM_Controller is particularly well-specified, with clear states and transitions.

The signals that are present in the modules are provided in the table below-

Signals	Type	Direction	Description
HCLK	AHB Bus Clock	Input	This clock times all bus transfers on AHB side.
HRESETn	Reset	Input	The bus reset signal is active LOW, and is used to reset the system and the bus.
HSIZE [2:0]	Data size	Input	Specifies the size of the data transfer.
HADDR [31:0]	Address bus	Input	The 32-bit system address bus.
HTRANS [1:0]	Transfer type	Input	This indicates the type of the current transfer, which can be NONSEQUENTIAL, SEQUENTIAL, IDLE or BUSY.
HWRITE	Transfer direction	Input	When HIGH this signal indicates a write transfer, and when LOW, a read transfer.
HWDATA [31:0]	Write data bus	Input	The write data bus is used to transfer data from the master to the bus slaves during write operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation.
HBURST [2:0]	Burst signals	Input	The hburst is a 3 bit signal.
HREADYIN/ HREADYout	Transfer done	Input/Output	When HIGH the HREADY signal indicates that a transfer has finished on the bus. This signal may be driven LOW to extend a transfer
HRDATA [31:0]	Read data bus	Output	The read data bus is used to transfer data from bus slaves to the bus master during read

			operations. A minimum data bus width of 32 bits is recommended. However, this may easily be extended to allow for higher bandwidth operation
HRESP [1:0]	Transfer response	Output	The transfer response gives further details about the status of a transfer. This module will consistently generate the OKAY response.
DATA READ [31:0]	Data read from hrdata for AHB Master	Output	The 32-bit data bus for read operations on the AHB Master side.
PClk	APB Bus Clock	Input	This clock times all bus transfers on APB side.
PSEL [2:0]	Peripheral slave select	Output	There is one of these signals for each APB peripheral present in the system. The signal indicates that the slave device is selected, and that a data transfer is required. It has the same timing as the peripheral address bus. It becomes HIGH at the same time as PADDR, but will be set LOW at the end of the transfer
PENABLE	Peripheral enable	Output	This signal is used to time all accesses on the peripheral bus. PENABLE goes HIGH on the second clock rising edge of the transfer, and LOW on the third (last) rising clock edge of the transfer.
PADDR [31:0]	Peripheral address bus	Output	This is the APB address bus, which may be up to 32 bits

			wide and is used by individual peripherals for decoding register accesses to that peripheral. The address becomes valid after the first rising edge of the clock at the start of the transfer. If there is a following APB transfer, then the address will change to the new value, otherwise it will hold its current value until the start of the next APB transfer.
PWRITE	Peripheral transfer direction	Output	This signal indicates a write to a peripheral when HIGH, and a read from a peripheral when LOW. It has the same timing as the peripheral address bus.
PWDATA [31:0]	Peripheral write data bus	Output	The peripheral write data bus is continuously driven by this module, changing during write cycles (when PWRITE is HIGH).
PRDATA [31:0]	Peripheral read data bus	Input	The peripheral read data bus is driven by the selected peripheral bus slave during read cycles (when PWRITE is LOW).

3 Required Tools

3.1 Software and hardware toolsets needed:

Synopsys VCS is used as the primary simulation platform for this project. It is well suited to the verification requirements due to its robust support for SystemVerilog and its capability to efficiently handle large and complex designs. In addition, waveform analysis and debugging are performed using Synopsys DVE and Verdi, which provide powerful visualization and debugging features essential for effective verification and issue resolution.

SystemVerilog has been selected as the primary language for testbench development. This choice takes advantage of SystemVerilog's advanced capabilities beyond traditional Verilog, including support for object-oriented programming constructs, constrained randomization, and functional coverage. These features facilitate the creation of comprehensive and sophisticated test scenarios, enabling thorough validation of the design functionality.

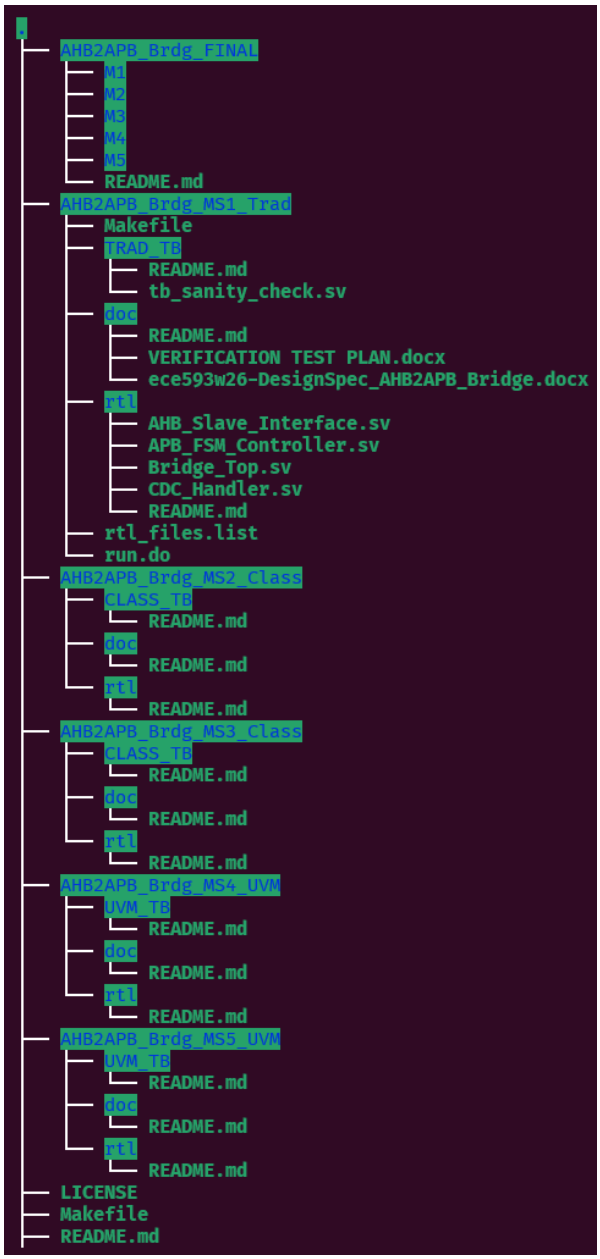
Read the "README.md" file in "AHB2APB_Brdg_MS1_Trad" directory to see how to run this design.

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

We will use following computer resources –

1. MCES servers: auto.ece.pdx.edu – For VCS and Verdi license
2. Our laptops – For RTL and Testbench development

Directory structure is given by our Professor Venkatesh Patil to follow using a Makefile which will set our paths.



4 Risks and Dependencies

4.1 Risks:

Gaining a thorough understanding of the AHB-to-APB bridge design requires time due to the complexity of the bus protocols and signal mappings involved. Delays in the verification process due to unforeseen complexities or difficulties, and the learning curve associated with using advanced verification methodologies and tools. We also have a very tight schedule. To mitigate these risks, we will use a systematic and thorough verification approach, start our work early, and seek help when needed. When integrating code into the GitHub main branch, there is a significant risk of encountering merge conflicts, particularly under a constrained project timeline. To reduce these risks, disciplined branch management practices must be adopted, such as performing regular merges, addressing conflicts in a timely manner, and maintaining clear and consistent communication among team members. Careful review of the design documentation and close collaboration within the team are essential to effectively resolve challenges as they emerge.

4.2 Dependencies:

Our project relies on the availability of the Synopsys VCS simulator, the completion of the RTL design, and access to the ARM documentation for the AHB-APB bridge.

5 Functions to be Verified.

5.1 Functions from specification and implementation

5.1.1 Functions that will be verified.

1. Single read:
   - AHB master initiates a single read transaction.
   - Bridge converts it to APB protocol
2. Single write:
   - AHB master initiates a single write transaction.
   - Bridge converts the write command to APB protocol
3. Burst read (INC by 4):
   - Sequential read operations with address incrementing by 4
   - Proper handling of multiple data phases ○ Correct address sequencing
   - Maintaining protocol timing requirements across multiple transfers
4. Burst write (INC by 4):
   - Sequential write operations with address incrementing by 4
   - Consistent data transfer across multiple write operations
   - Proper address incrementing
   - Protocol timing maintenance across burst sequence
5. Reset:
   - Synchronous active low reset behavior verification
   - All state machines return to idle state
   - All control signals return to default values

- Proper resumption of operation after reset

6. Address Decoding:
   - Correct slave selection based on address range
   - Proper address translation between AHB and APB domains ○ Valid/invalid address range handling ○ Address alignment checking

5.1.2 Functions that will not be verified.

Error Handling:

- Basic error handling excluded from initial design to focus on core protocol conversion functionality
- Features like slave error responses, timeout conditions, and invalid address handling are currently not implemented
- Error handling mechanisms can be integrated in future versions

Burst transfers beyond INC by 4:

- Current implementation limited to INCR4 bursts due to timing constraints and design complexity considerations
- Other burst types (WRAP4, WRAP8, WRAP16, INCR8, INCR16, undefined length bursts) are not implemented
- Future scope exists to expand burst support once core functionality is thoroughly verified.

5.1.3 Critical functions

Reset, Data transfer and addressing are critical functions that are mandatory to be correct for the basic functionality of the AHB to APB Bridge.

6 Tests and Methods

6.1.1 Testing methods to be used

We will employ black-box verification initially, focusing on the system's inputs and outputs to ensure correct behavior for specified input scenarios. As development progresses, grey-box verification will be applied to leverage knowledge of internal design components for deeper insights into functionality.

6.1.2 PROs and CONs for using the DUV.

Black-box Verification

- **Pros:** Simple to implement, validates system behavior without requiring internal knowledge, effective for early sanity checks.
- **Cons:** Limited visibility into internal interactions, may miss corner-case internal errors.

Grey-box Verification

- **Pros:** Provides insight into internal interactions, enables more targeted testing of component interactions.
- **Cons:** Requires detailed knowledge of internal design, more complex to implement.

This hybrid approach suits the AHB-APB bridge, allowing verification of both overall behavior and internal component interactions.

6.1.3 Testbench Architecture; Components used

We use a SystemVerilog-based testbench with the following components:

- Drivers / Bus Functional Models (BFM): Generate read/write transactions for the AHB Master.
- Monitors: Capture AHB and APB signals for verification and coverage analysis.
- Scoreboards: Compare expected vs. observed results for read/write operations.
- Checkers / Assertions: Validate protocol compliance, FSM transitions, and signal timing.
- Coverage Collectors: Track functional coverage for bursts, increments, and corner cases.

6.1.4 Verification Strategy:

We are employing dynamic simulation, enabling detailed functional testing of stimulus-response behavior over time. This approach is ideal for catching timing-related errors and verifying complex transaction sequences in the bridge design.

6.1.5 Driving methodology

Test generation begins with constrained-random tests to explore a broad set of scenarios. Based on functional coverage analysis, directed tests are added to target untested paths or corner cases.

6.1.6 Checking methodology

Verification uses multi-level checks:

- From Specification we'll Ensure DUT meets AHB and APB protocol requirements.
- From Implementation we'll check **s**anity checks of FSM states and signal interactions.
- From Context we will **v**alidate bridge operation in its intended system environment.
- From Architecture we are going to **c**onfirm correct communication and coordination between internal modules.

6.1.7 Testcase Scenarios (Matrix)

6.1.7.1 Basic Tests

Test Name / Number	Test Description/ Features	Status
1.1.1 Single Write Halfword Nonseq	Check basic write operation with 16bits	Done
1.1.2 Single Read Halfword Nonseq	Check basic read operation with 16bits	Done
1.1.3 Read Increment Burst Halfword Nonseq	Check read operation with increment burst type with 16bits	Done
1.1.4 Write increment word Nonseq	Check write operation with increment burst type with 16bits	Done
1.1.5 Read Nonseq Wrap	Check read operation with wrap as burst type with 8 bits	Done
1.1.6 Write Nonseq Wrap	Check write operation with wrap as burst type with 8 bits	Done

6.1.7.2 Complex Tests

Test Name / Number	Test Description/ Features
1.2.1 Back-to-back write and read	Performs write then read
1.2.2 Burst Read	Performs an incrementing burst read of four transfers by setting hburst to INCR4, hsize to word (32-bit), and updating haddr sequentially while capturing hrdata from APB slave.
1.2.3 Burst Write	Performs an incrementing burst write of four transfers by setting hburst to INCR4, hsize to word (32-bit), and updating haddr sequentially while writing hwdata at each cycle.
1.2.4 Pipeline write	Stress tests the pipeline write function by giving next address (Haddr) and data (HWrite) in same cycle
1.2.5 Pipeline read	Stress tests the pipeline read function by giving next address (Haddr) and data (HWrite) in same cycle
1.2.6 CDC test	Performs test by keeping PClk 2x slower than HClk frequency

6.1.7.3 Regression Tests

Test Name / Number	Test Description/Features
1.3.1 Single Write	Check basic write operation
1.3.2 Single Write	Check basic read operation
1.3.3 Back-to-back write and read	Performs write then read
1.3.4 Burst Read	Performs an incrementing burst read of four transfers by setting hburst to INCR4, hsize to word (32-bit), and updating haddr sequentially while capturing hrdata from APB slave.
1.3.5 Burst Write	Performs an incrementing burst write of four transfers by setting hburst to INCR4, hsize to word (32-bit), and updating haddr sequentially while writing hwdata at each cycle.

6.1.7.4 Any special or corner cases testcases

Test Name / Number	Test Description
1.4.1 Hsize> than supported	Try to input Hsize > than supported in AMBA AHB and the transfers should not pass.
1.4.2	

7 Coverage Requirements

Our verification strategy focuses on comprehensive testing of the design through multiple coverage metrics. We will use Synopsys VCS code coverage tools to verify that all code lines have been exercised during testing.

Beyond basic code coverage, we'll implement functional coverage to ensure we've tested all relevant bus operation scenarios, including edge cases and critical corner conditions. This dual approach of code coverage and functional coverage will help ensure that our verification effort is thorough and that the design behaves correctly across all possible usage scenarios.

8 UVM Architecture

UVM stands for Universal Verification Methodology. It is a standardized approach to verify complex digital designs using System Verilog. UVM provides a set of libraries, classes, and guidelines that help verification engineers to create reusable, scalable, and portable testbenches.

A basic testbench consists of different components like agent, drivers, monitors, environment, etc. System Verilog only provides a way by which we create these components by providing construct like class, enums, etc.

UVM also defines a standard verification architecture that consists of several components, such as test, environment, agent, driver, monitor, scoreboard, etc. These components communicate with each other through transactions and interfaces. UVM also provides a set of utilities, such as configuration database, factory, messaging service, register abstraction layer, etc. that simplify the verification tasks.

By using UVM, verification engineers can create modular, reusable, and configurable testbenches that can be easily adapted to different design specifications and verification goals. UVM also enables verification reuse across different levels of abstraction, such as block-level, subsystem-level, and system-level.

As it's core it has the following components-

- Agents: An agent encapsulates a sequencer, driver and monitor into a single entity by instantiating and connecting the components together via TLM interfaces. Since UVM is all about configurability, an agent can also have configuration options like the type of UVM agent (active/passive), knobs to turn on features such as functional coverage, and other similar parameters. These are modular blocks that can operate in active (driving stimulus) or passive (only monitoring) modes, enhancing reusability across different test scenarios.
- Sequencers: a uvm_sequencer is a UVM component responsible for managing the flow of transactions generated by UVM sequences. Sequences are like the "drivers" of stimulus, defining what actions should occur (e.g., a read, a write, a complex protocol step), and uvm_sequencer determines when and in what order these actions are delivered to the actual hardware interface through a uvm_driver. It's Responsible for generating and randomizing the stimulus based on specific test scenarios or sequences.
- Drivers: UVM driver is an active entity that has knowledge on how to drive signals to a particular interface of the design. All driver classes should be extended from uvm_driver, either directly or indirectly. Transaction level objects are obtained from the sequencer and the UVM driver drives them to the design via an interface handle.
- Monitors: A UVM monitor is responsible for capturing signal activity from the design interface and transaction level data objects that can be sent to other components. Continuously observe the DUT's interfaces, capturing its responses and behavior. A UVM monitor is derived from uvm_monitor base class and should have the following functions:
    - Collect bus or signal information through a virtual interface
    - Collected data can be used for protocol checking and coverage

- Collected data is exported via an analysis port

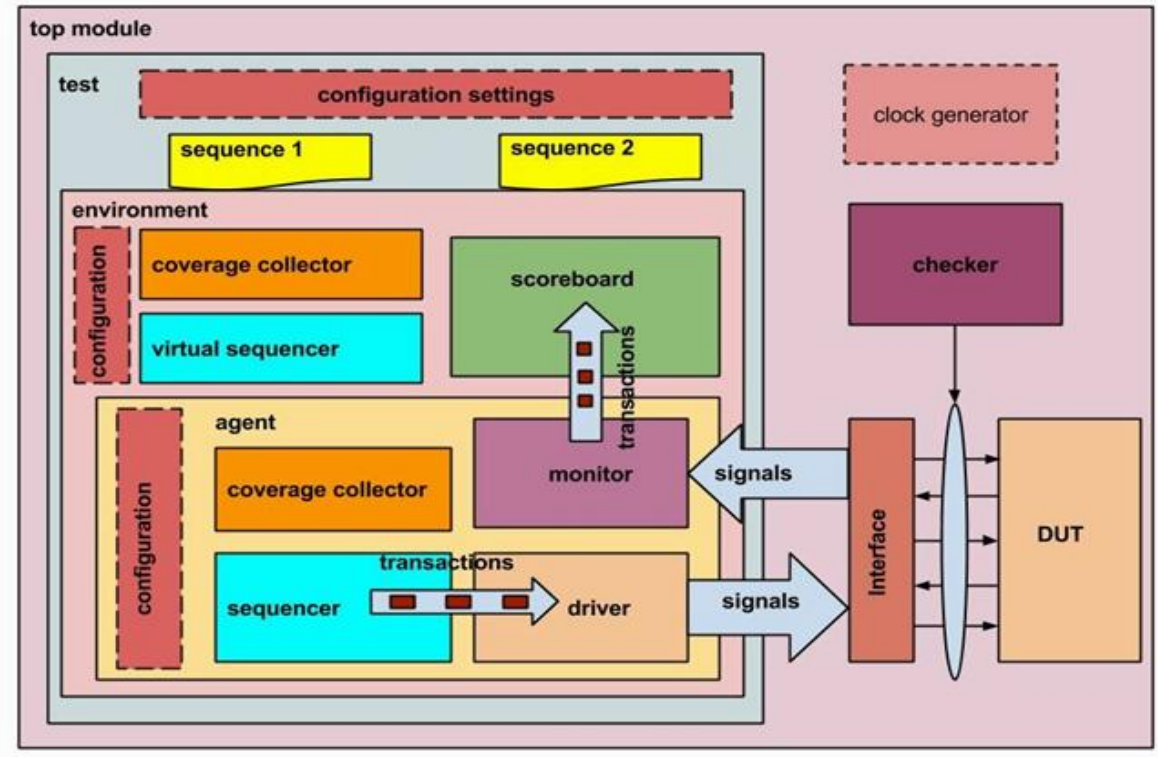
The UVM monitor functionality should be limited to basic monitoring that is always required.It may have knobs to enable/disable basic protocol checking and coverage collection. High level functional checking should be done outside the monitor, in a scoreboard.

- Scoreboard: A critical component that compares the DUT's responses with expected results, flagging any discrepancies. UVM scoreboard is a verification component that contains checkers and verifies the functionality of a design. It usually receives transaction level objects captured from the interfaces of a DUT via TLM Analysis Ports.
- UVM Tests: These encapsulate specific test scenarios, leveraging the above components to stimulate and verify the DUT for that scenario. A testcase is a pattern to check and verify specific features and functionalities of a design. A verification plan lists all the features and other functional items that needs to be verified, and the tests neeeded to cover each of them.

Custom classes can be created using uvm_object. uvm_object is the base class in UVM from which all other UVM components and objects are derived. uvm_object itself is derived from uvm_void void. It provides a set of common utility functions like print, copy, compare, and record that any class in a UVM testbench can use.

Configuration Database: The UVM configuration database is a powerful tool that allows for hierarchical configuration and customization of UVM components without direct connections. In our testbench, the uvm_config_db is pivotal in passing the virtual interfaces, and other configurations, to the UVM components, ensuring they have the necessary context to interact with the DUT and each other.

UVM based TB:



9 Resources requirements

a) Project Members

- Amey Kulkarni - Design Specification, first half of Bridge controller, AHB Master module and CDC logic.
- Annapoorna Hamsagaru Manjunath - Design Specification, APB slave interface, Interface for modules, Verification plan and Testbench development

b) Computing resources - Remote Systems from Portland State University.

c) Simulator license – Synopsys VCS provided by PSU remote systems.

10 Schedule

Milestones	**Date**	**Objectives**
Milestone 1	01/30/2026	Defining the design specification and verification plan, along with developing the APB slave interface, simple testbench, module interfaces, and the AHB master module for seamless communication along with the bridge controller.
Milestone 2	02/18/2026	Class based testbench with all components and interfaces. Should verify ~50 random bursts of data.
Milestone 3	02/18/2026	Finalize changes in RTL. code and functional coverage. Update verification plan document.
Milestone 4	02/27/2026	UVM testbench along with updated verification plan. UVM reporting macros to log reports/data to the console.
Milestone 5	03/14/2026	Complete UVM architecture, environment and testbench. Bug injection. Finalized documents, paper and presentations.

11 References Uses / Citations/Acknowledgements

- RTL Design code referenced from "https://github.com/prajwalgekkouga/AHB-to-APB-Bridge"
- ARM Documentation "https://developer.arm.com/documentation/ihi0033/latest/"