# VERIFICATION TEST PLAN

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

Portland State
UNIVERSITY

**Project Name**: Design and Verification of AHB to APB Bridge using UVM

**Members**:
Maithreyi Venkatesan - 900119348
Raghavendra Davarapalli - 955118446
Neil Austin Tauro - 956372423
Suraj Vijay Shetty - 967611331

## Table of Contents

## Introduction

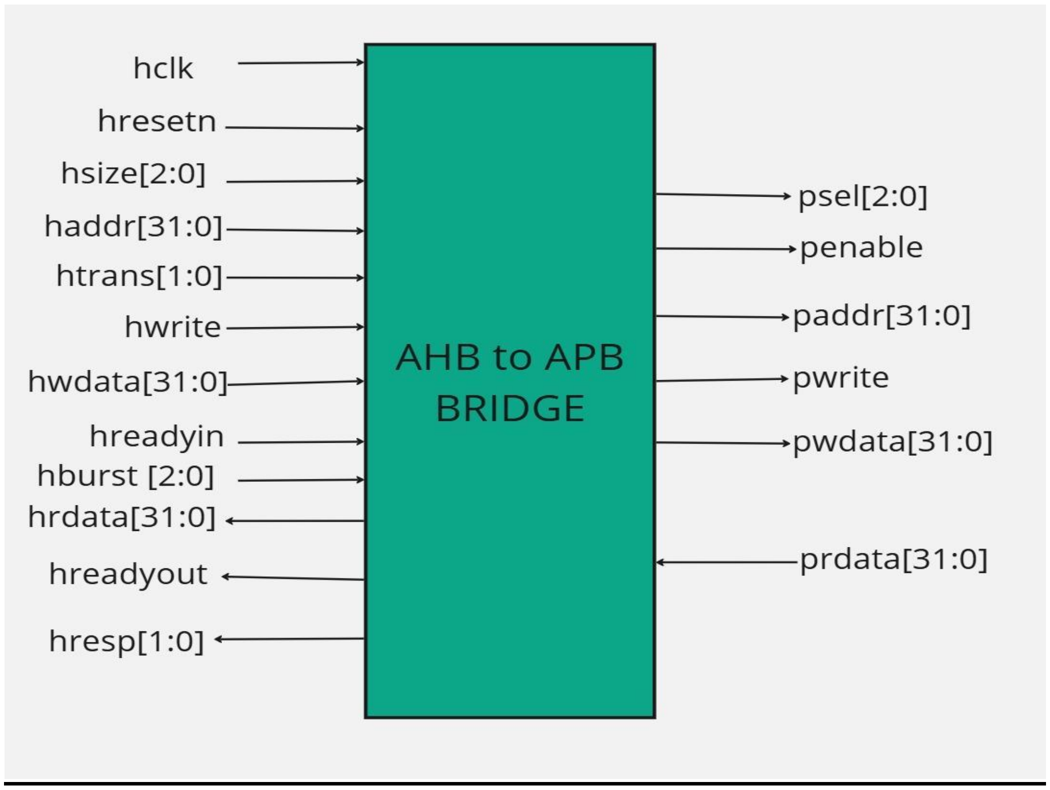
The goal of this verification plan is to specify the strategy and verification methodology that we incorporated to verify the functionality of the (Advanced High-Performance Bus) to APB (Advanced Peripheral Bus) bridge. The AHB to APB Bridge is a critical component in the AMBA (Advanced Microcontroller Bus Architecture) system, enabling seamless communication between high-performance master devices (on the AHB bus) and lower-bandwidth peripheral devices (on the APB bus).

### a) Objective of the verification

The dummy (AHB Master) module includes two tasks, one dedicated to handling read operations and the other focused on managing write operations. These tasks are implemented to simplify the process of interacting with the design under test (DUT) by encapsulating the logic for reading and writing in reusable procedural blocks.

In addition to the tasks, a self-checking testbench has been developed to validate the functionality of the design. By combining the read and write tasks with the self-checking mechanism, the testbench ensures efficient and reliable verification of the design's behavior under different scenarios.

### b) Block Diagram



## c) Specifications for the design

- The AHB to APB Bridge is the only synthesizable module in our project and the signals are all present exactly as in the block diagram. Since the AHB is pipelined and the APB is non-pipelined, we need to convert the AHB signals to APB signals by inserting wait states. This protocol will be according to the Finite State Machine (FSM) employed in the design specification.
- We are only verifying the single read/write and a simple incrementing by 4 burst transfer and this will be the scope of this project.

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

| Signals | Type | Direction | Description |
|---|---|---|---|
| HCLK | Bus clock | Input | This clock times all bus transfers. |
| 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. |
| 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 enable 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). |

## Verification Requirements

### a) 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.

### b) Controllability and Observability

Controllability is typically high at this level since individual modules can be directly stimulated with specific input values to exercise all possible scenarios. Observability is also high because the outputs of each module (e.g., hreadyout, hrdata, prdata) can be monitored without interference from other components. Debugging is straightforward as there are fewer dependencies.

## Required Tools

The verification environment for the AHB-APB bridge design will be built using several key tools and technologies:

### a) Simulation Tool and Language*:*

QuestaSim serves as our primary simulation platform. It's particularly well-suited for our needs as it offers comprehensive verification capabilities and supports SystemVerilog, which is our chosen language. The tool provides robust debugging features that will be essential for our verification process.

For testbench development, we've selected SystemVerilog as our primary language. This choice leverages SystemVerilog's advanced features that extend beyond basic Verilog, including object-oriented programming concepts like classes, as well as randomization and functional coverage capabilities. These features will enable us to create thorough and sophisticated test scenarios.

### b) Version Control System:

To ensure proper project management and collaboration, we'll implement version control using Git. This will help us track changes throughout the development process and maintain a clear history of our design and testbench modifications.

c) Coverage Analysis Tool:

Coverage analysis will be handled through QuestaSim's built-in capabilities. This will allow us to measure both code coverage and functional coverage, providing insights into the thoroughness of our verification efforts.

d) Documentation Tool:

For project documentation, we'll use a combination of Microsoft Word and Google Docs to maintain detailed records of our verification plan, test cases, and results.

This comprehensive toolset will enable us to effectively verify that the AHB-APB bridge meets all design specifications and functions correctly.

## Risks and Dependencies

**Risks:** While pushing the code to GitHub, there is a high chance of merge conflicts to the main branch. We also have a very tight schedule. To mitigate these risks, proper branch management practices should be followed, including frequent merging, resolving conflicts promptly, and ensuring clear communication within the team. Understanding the complexity of the AHB to APB design takes time due to the intricacies of the bus protocols and signal mapping. It's important to thoroughly review the documentation and collaborate with the team to address any challenges as they arise.

**Dependencies:** Our project relies on the availability of the QuestaSim simulator, the completion of the RTL design, and access to the ARM documentation for the AHB-APB bridge. We will need to collaborate closely with the design team to ensure the design is completed on time and that any design changes are communicated to us promptly.

## Functions

While our design includes distinct modules (AHB Master, AHB Slave Interface, APB FSM Controller, and APB Interface), our verification strategy will be driven by the AMBA specification document rather than module boundaries. We'll extract the required bridge functionalities from this specification and verify them against our RTL implementation. Each functionality will undergo thorough testing across multiple scenarios, encompassing normal operating conditions, boundary cases, and error situations. This specification-driven approach ensures we verify the design's actual requirements rather than just testing individual modules in isolation.

<u>**Functions to be verified**</u>**:**

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. 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
4. 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

<u>**Functions not verified**</u>**:**

1. Burst transfers:
    - Current implementation limited to single transfers 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.

2. 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

<u>**Critical Functions:**</u>

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

## Testing Methods

In the initial stages, we will employ black box verification, focusing on the system's inputs and outputs to ensure it behaves as expected for given inputs. This approach, complemented by assertions for sanity checks and finite state machine transition verification, is well-suited for preliminary testing. As we progress, we will transition to grey box verification, leveraging our knowledge of the system's internal workings. This hybrid method is particularly beneficial for our AHB-APB bridge design, as it provides deeper insights into the interactions between components like the AHB Master, AHB Slave Interface, APB FSM Controller, and APB Interface, and their contributions to the system's overall functionality.

At this point, we are implementing directed test cases using the AHB Master module that sanity checks our design using single read/write and burst (INC by 4) operations. Moving forward, we will be moving onto constrained random testing with a class based testbench and finally moving onto a full fledged UVM based testbench.

## Checking Methodology

Our verification strategy incorporates two key approaches:

1. Comprehensive Output Verification: We'll validate all system outputs to ensure correct behavior under various input conditions. This includes:
    - Monitoring FSM state transitions
    - Verifying accuracy of read data operations
    - Confirming proper APB bus control signal timing and values.

2. Multi-Level Design Verification: Our verification focuses on both functional and structural aspects:
    - Design Context: Verifying the bridge operates correctly within its intended use case
    - Microarchitecture: Ensuring internal components work together properly
    - Component Interaction: Validating proper communication between the bridge's internal blocks

- ○ Protocol Compliance: Confirming adherence to both AHB and APB specifications

**Testcase Scenarios:**

Initially, simple test cases have been used for the standard conventional testbench and Class based testbench.

| Test Name/Number | Test Description/Features |
|---|---|
| Single write halfword non sequential | Performs a single non-sequential write transaction by randomizing and setting the necessary signals before sending the transaction to the driver. |
| Single read halfword non sequential | Performs a single non-sequential read transaction by randomizing and setting the necessary signals before sending the transaction to the driver. |
| Single write non sequential error | Performs a single non-sequential write transaction with an error response (hresp=1), randomizes and sets the necessary signals, then sends the transaction to the driver after sampling coverage. |
| Single read byte okay | Performs a single read transaction for a byte-sized transfer (Hsize = 3'b000), sets the necessary signals, samples coverage, and sends the transaction to the driver. |
| Single write halfword seq okay | Performs a single sequential write transaction by setting the necessary signals (Hwrite = 1, Htrans = 2'b11), samples coverage, and sends the transaction to the driver. |

| | |
|---|---|
| Single read word okay | Performs a single read transaction for a word-sized transfer (Hsize = 3'b010), sets the necessary signals, samples coverage, and sends the transaction to the driver. |
| Single write byte sequential error | Performs a sequential byte-sized write transaction (Hsize=3'b000) with an error response (hresp=1), samples coverage, and sends the transaction to the driver. |
| Single read byte nonsequential reset | Performs a single non-sequential byte-sized read transaction (Hsize=3'b000) with a reset condition (hreset=1), samples coverage, and sends the transaction to the driver. |
| Single write halfword nonsequential reset | Performs a single non-sequential halfword write transaction (Hsize=3'b001) with a reset condition (hreset=1), then sends the transaction to the driver. |
| Single read word nonsequential reset | Performs a single non-sequential word-sized read transaction (Hsize=3'b010) with a reset condition (hreset=1), then sends the transaction to the driver. |
| Single read byte sequential reset | Performs a sequential byte-sized read transaction (Hsize=3'b000) with a reset condition (hreset=1), then sends the transaction to the driver. |
| Single write halfword sequential reset | Performs a sequential byte read transaction with a reset condition (hreset=1) and sends it to the driver. |

| | |
|---|---|
| Single read word sequential reset | Performs a sequential word-sized read transaction (Hsize=3'b010) with a reset condition (hreset=1), randomizes the transaction, and sends it to the driver. |
| Single write byte sequential error reset | Performs a sequential byte-sized write transaction (Hsize=3'b000) with an error response (hresp=1) and a reset condition (hreset=1), randomizes the transaction, and sends it to the driver. |
| Single write byte idle error | Performs a byte-sized write transaction (Hsize =3'b000) with an idle transaction (Htrans= 2'b00) and an error response (hresp=1), samples coverage, and sends it to the driver. |

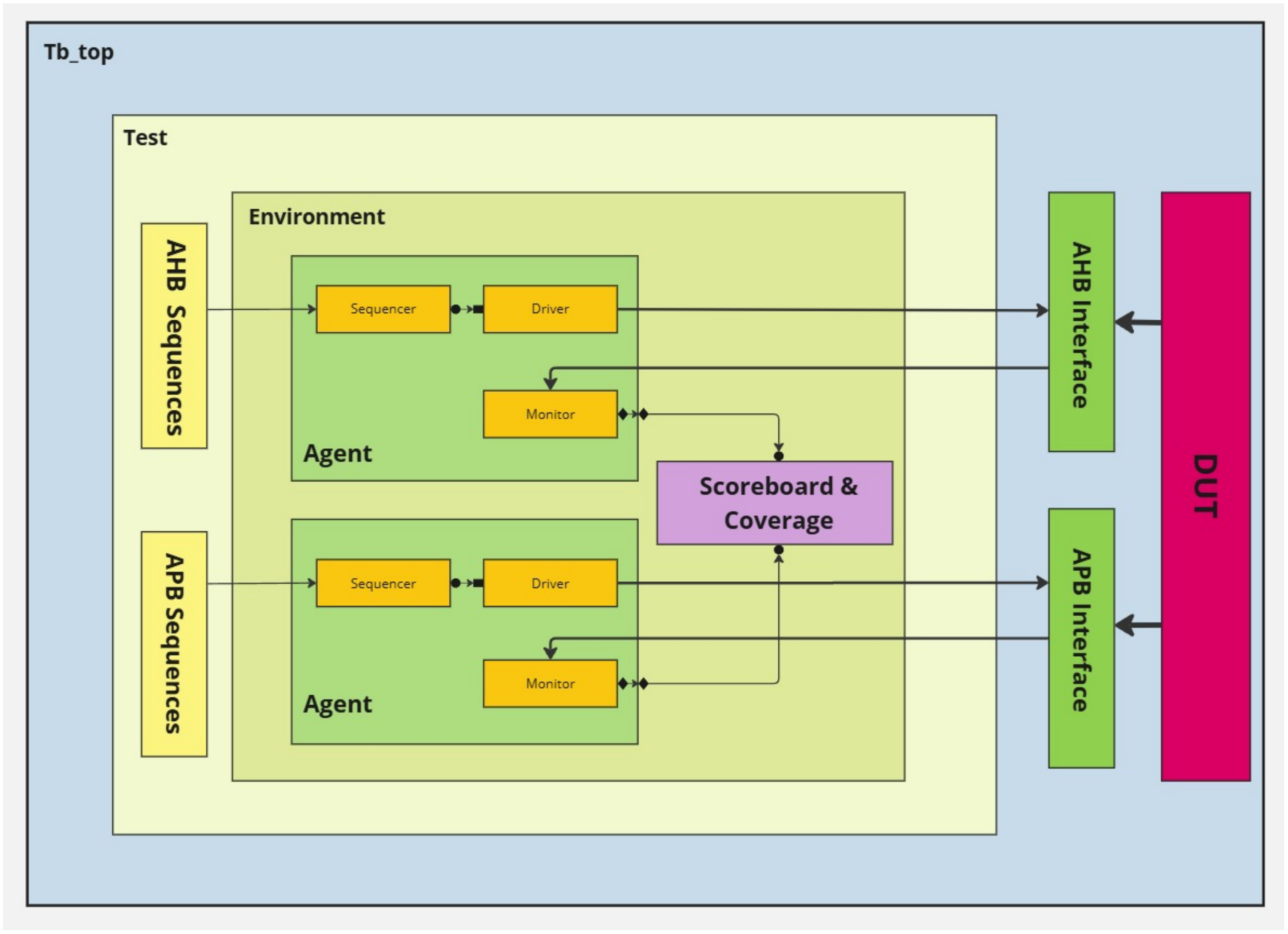
Finally, for the UVM testbench, we have modified our test cases.

| **Test Name/Number** | **Test Description/Features** |
|---|---|
| APB sequence | Generates a series of randomized APB transactions, initializes apb_transaction objects, randomizes them, samples coverage, and sends them to the driver for execution. |
| APB Random sequence | Generates APB transactions with controlled PSLVERR values, creating sequences with PSLVERR = 0 for most iterations, PSLVERR = 1 for a subset, and a final transaction with PSLVERR = 0 before sending them to the driver. |

| | |
|---|---|
| APB Single Write sequence | Performs a series of APB write transactions with PSLVERR set to 0 for three iterations, samples coverage, and sends the transactions to the driver for execution. |
| APB Single Read sequence | Performs a series of APB read transactions with PSLVERR set to 0 for three iterations, samples coverage, and sends the transactions to the driver for execution. |
| AHB Sequence | Generates a series of randomized AHB transactions, initializes ahb_transaction objects, randomizes them, samples coverage, and sends them to the driver for execution. |
| AHB Random sequence | Generates AHB transactions with specific HTRANS values, including IDLE (00), NONSEQ (10), and alternating values, samples coverage, and sends the transactions to the driver for execution. |
| AHB Single Write sequence | Generates a series of AHB write transactions with varying constraints on reset, selection, and transfer type, samples coverage, and sends the transactions to the driver for execution. |
| AHB Single Read sequence | Generates a series of AHB read transactions with varying constraints on reset, AHB selection, and transfer type. The sequence ensures different states, such as IDLE and non-sequential transfer, are covered and sends the transactions to the driver for execution, allowing for thorough verification of the read operations. |

|  |  |
|---|---|
| Combined AHB sequence | Generates a series of AHB write and read transactions with varying constraints on reset, selection, and transfer type. The sequence first sends multiple write transactions followed by read transactions, with a small delay between the two types, ensuring comprehensive coverage of different AHB operations and states. |

**Verification IP structure**



## Verification Methodology

The testbench follows a layered architecture and consists of the following components:

**Tb_top (Testbench Top):** The highest-level testbench module that integrates all components.

**Test:** Defines the verification scenario, applies stimulus, and collects results.

**AHB and APB Sequences:** Generates stimulus (transactions) for AHB and APB protocols.

## Verification Components

### Agents (AHB and APB):

**Sequencer**: It generates stimulus transactions and sends them to the driver in a controlled manner. It ensures that the transactions are executed in the correct order and manages their flow during simulation.

**Driver**: It receives transactions from the sequencer and converts them into pin-level signals that interact with the DUT. It mimics the behavior of actual hardware components, ensuring accurate communication with the DUT.

**Monitor**: It passively observes the signals on the interface, capturing DUT responses without driving any signals. It forwards the captured data to the scoreboard and coverage components for validation and analysis.

### Environment:

A collection of agents, scoreboard, and coverage.

**Scoreboard**: Compares expected vs. actual DUT outputs.

**Coverage**: Collects functional coverage to track tested scenarios.

**AHB-APB Verification Environment (ahb_apb_env):** The ahb_apb_env class serves as the central verification environment for the AHB-APB Bridge, integrating all key verification components. It facilitates the setup, connection, and execution of the verification process by incorporating agents for both AHB and APB protocols, along with a scoreboard for result validation.

**Test and Top_module:**

**AHB & APB Interfaces:** Connect the testbench to the DUT for communication.

**DUT (Design Under Test):** The actual hardware design being verified.

**Key Components:**

1. **Configuration Handle (env_config_h)**
   - Manages the environment's configuration, defining its structure and behavior.
   - Determines which agents and components are active during a simulation.

2. **AHB and APB Agents (**ahb_agent_h, apb_agent_h**)**

   - Drive and monitor transactions on the AHB and APB interfaces, respectively.
   - Each agent includes:
     - A driver for generating transactions.
     - A monitor for observing bus interactions.

3. **Scoreboard (sb_h)**

   - Validates the DUT's behavior by comparing observed transactions with expected results.
   - Receives data from monitors to ensure correct protocol execution.

This environment plays a crucial role in verifying the functionality and correctness of the AHB-APB Bridge.

We employed a bottom-up testing approach, first verifying individual modules before integrating and testing larger subsystems. Starting with the transaction generator, we ensured its basic functionality using simple test scenarios before progressing to more complex cases. We verified the transaction generator, which creates and drives transactions into the DUT. Initial tests covered basic functionality with simple scenarios. Next, we integrated the monitor to observe DUT transactions and the scoreboard to validate results.

We began with basic write and read tests, gradually increasing complexity and coverage. As more test cases were introduced, we thoroughly assessed the DUT's robustness against various scenarios and edge cases. This structured approach ensured comprehensive validation of the

AHB APB Bridge, confirming its compliance with specifications and functional correctness. By methodically verifying its reliability, we ensured a production-ready design.

**Key Methods:**

1. **Constructor (new)**
    - Initializes the environment component by setting its name and parent.
2. **Build Phase (build_phase)**
    - Creates the necessary environment components based on configuration settings.
    - Determines which agents and the scoreboard should be instantiated.
    - For example, if the AHB agent is enabled in the configuration, it is instantiated accordingly, and the same applies to the APB agent and the scoreboard.
3. **Connect Phase (connect_phase)**
    - Establishes connections between various components.
    - If both the AHB agent and the scoreboard are enabled, the AHB monitor's output port is linked to the scoreboard's AHB analysis port, ensuring transactions are sent for validation.
    - Similarly, if the APB agent and scoreboard are enabled, the APB monitor's output port is connected to the scoreboard's APB analysis port for transaction checking.

## AHB-APB Scoreboard (ahb_apb_scoreboard)

The ahb_apb_scoreboard class plays a crucial role in the verification process by monitoring, predicting, and validating data transactions between the AHB and APB interfaces.

**Core Components:**

1. **FIFOs (ahb_fifo and apb_fifo)**
    - Serve as temporary storage for transactions captured by the AHB and APB monitors.
    - Ensure that transactions are processed in the order they are received.

2. **Packet Variables**
    - Hold incoming transactions (ahb_data_pkt and apb_data_pkt).
    - Store temporary data (ahb_temp_data).
    - Maintain predicted values (ahb_predicted_pkt and apb_predicted_pkt) for validation.

3. **Counters**

    - Keep track of transaction statistics, including total processed packets and correctly verified transactions.
    - Important counters include ahb_pkt_count, apb_pkt_count, and verified_data_count.

4. **Covergroup (cov_group)**
    - Collects functional coverage data to assess which aspects of the design have been exercised.
    - Tracks key operations such as reset behavior, read and write operations, and transaction types to ensure comprehensive verification.

**Key Methods:**

1. **Constructor (new)**
    - Initializes the FIFOs, sets up packet-related variables, and configures the coverage group.
2. **Run Phase (run_phase)**
    - Actively monitors packets from both AHB and APB interfaces.
    - Logs each transaction and predicts expected data based on sampled packets.
    - Collects functional coverage data throughout this phase.
3. **Data Prediction (predict_data)**
    - Determines the expected output for AHB and APB transactions.
    - Predictions are made based on the type of operation, such as an AHB read or write.
4. **PSELx Configuration (configure_pselx)**
    - Sets the PSELx signal according to the AHB address.
    - Directs data to the appropriate slave on the APB side.
5. **Data Validation (check_apb_data and check_ahb_data)**
    - Compares observed data against predicted values.
    - Increments verified_data_count for each successful match.
6. **Report Phase (report_phase)**
    - Summarizes the verification results at the end of the simulation.
    - Provides details on the total packets processed, verified transactions, and coverage metrics.

**Single Write Test Class (ahb_apb_single_write_test)**

The ahb_apb_single_write_test class extends ahb_apb_base_test and focuses on validating the DUT’s behavior during **single write operations** on the AHB and APB interfaces.

**Core Components:**

1. **Single Write Sequences (ahb_seq_h and apb_seq_h)**
    - These sequence handles generate single write transactions on the AHB and APB interfaces.

**Primary Methods:**

1. **Constructor (new)**
    - Initializes the test class by setting its name and parent.
2. **Build Phase (build_phase)**
    - Calls the base test's build_phase using super.build_phase(phase).
    - Instantiates single write sequences for AHB and APB interfaces.
3. **Run Phase (run_phase)**
    - Raises an objection to keep the simulation running.
    - Executes the **single write sequences** concurrently on their respective sequencers using fork...join.
    - Drops the objection once execution completes.
    - Sets a drain time to allow any remaining transactions to be processed before simulation ends.

**<u>Single Read Test Class (ahb_apb_single_read_test)</u>**

The ahb_apb_single_read_test class, derived from ahb_apb_base_test, is designed to validate the DUT's behavior during **individual read operations** on the AHB and APB interfaces.

**Core Components:**

1. **Single Read Sequences (ahb_seq_h and apb_seq_h):** These sequence handles generate single read transactions on the AHB and APB interfaces.

**Primary Methods:**

1. **Constructor (new)**
    - Initializes the test class by assigning its name and parent.
2. **Build Phase (build_phase)**
    - Calls the base test's build_phase using super.build_phase(phase).
    - Creates instances of the single read sequences for AHB and APB interfaces.
3. **Run Phase (run_phase)**
    - Raises an objection to keep the simulation active.

- Launches the **single read sequences** concurrently on their respective sequencers using fork...join.
- Drops the objection once execution is complete.
- Assigns a **drain time** to allow pending transactions to be processed before the simulation concludes.

## <u>Base Test Class (ahb_apb_base_test)</u>

The ahb_apb_base_test class serves as the foundation for all test scenarios, establishing the essential environment, configurations, and interfaces required for the AHB-APB testbench.

**Core Components:**

1. **Environment Configuration (env_config_h)**
   - Defines the overall testbench setup, including agent statuses, interfaces, and the scoreboard.

2. **Main Testbench Environment (env_h)**
   - Represents the primary testbench environment where the verification process takes place.

**Primary Methods:**

1. **Constructor (new)**
   - Initializes the test class by assigning a name and parent.

2. **Build Phase (build_phase)**
   Creates the environment configuration object.
   - Retrieves virtual interfaces (ahb_vif and apb_vif) for AHB and APB from the configuration database.
   - Activates the AHB and APB agents for the test.
   - Instantiates the main testbench environment (env_h) to facilitate verification.

## <u>Random Test Class (ahb_apb_random_test)</u>

The ahb_apb_random_test class extends the base test and introduces **randomized traffic** on both AHB and APB interfaces. This test enhances verification by assessing the DUT's behavior under **unpredictable** conditions, ensuring robustness.

**Core Components:**

1. **Random Sequences (ahb_rand_seq_h and apb_rand_seq_h):** These handles correspond to sequences that generate random transactions for the AHB and APB interfaces.

**Primary Methods:**

1. **Constructor (new)**
   - Initializes the test instance by setting its name and parent.
2. **Build Phase (build_phase)**
   - Calls super.build_phase(phase) to inherit the base test's build setup.
   - Creates instances of random sequences for both AHB and APB interfaces.

3. **Run Phase (run_phase)**
   Raises an objection to keep the simulation active.
   - Launches random sequences on their respective sequencers concurrently using fork...join.
   - Drops the objection once execution is complete.
   - Assigns a drain time to ensure all pending transactions are processed before ending the simulation.

## Coverage Requirements

Our verification strategy focuses on comprehensive testing of the design through multiple coverage metrics. We will use QuestaSim's 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.

**Functional Coverage:**

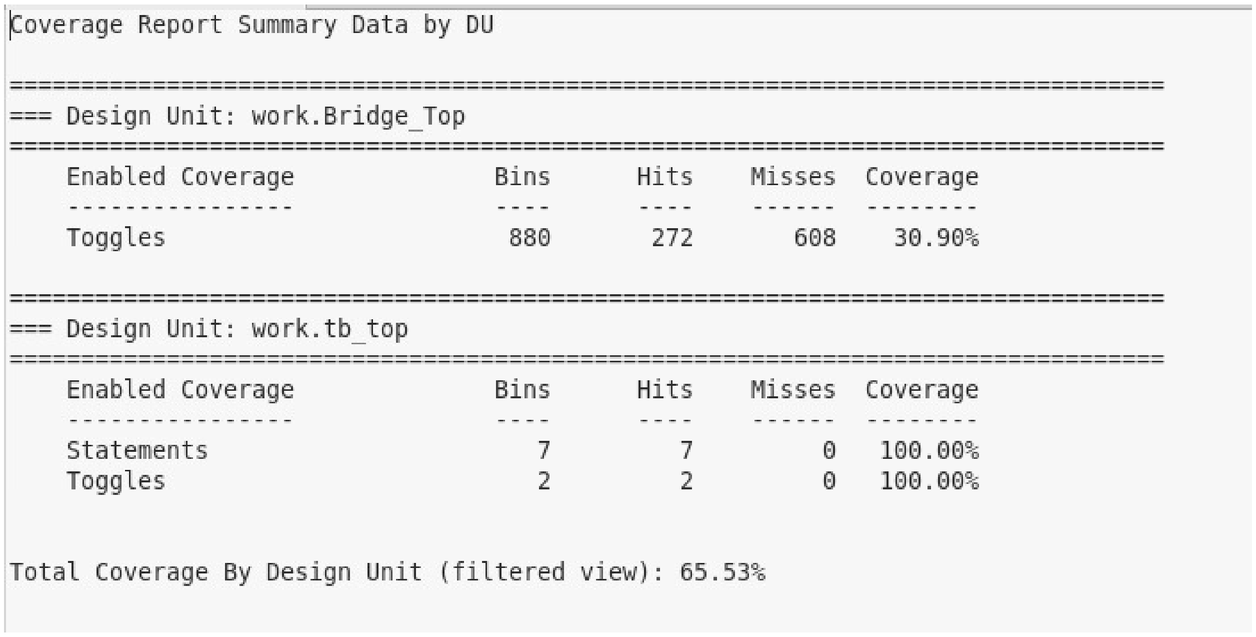
In our UVM-based verification environment, functional coverage is used to evaluate and ensure that all potential scenarios, states, and transitions of the design are tested. The covergroup construct in SystemVerilog is used to define specific coverage points and bins that represent different values or value ranges.

**Coverpoints:** These are specific points in the design or verification environment where coverage is measured. In our cov_group, we define multiple coverpoints such as reset, bus_write, bus_read, H_Data and trans_type.

- The reset coverpoint monitors the coverage of the reset signal (HRESETn), specifically focusing on the scenario where the reset signal is 0.
- The bus_write and bus_read coverpoints track the HWRITE signal, capturing scenarios where write and read operations occur, respectively.
- The trans_type coverpoint tracks the different transaction types (HTRANS) in the AHB protocol, including idle, non-sequential, and sequential transactions.
- The H_Data coverpoint tracks the range of values taken by the HWDATA signal.

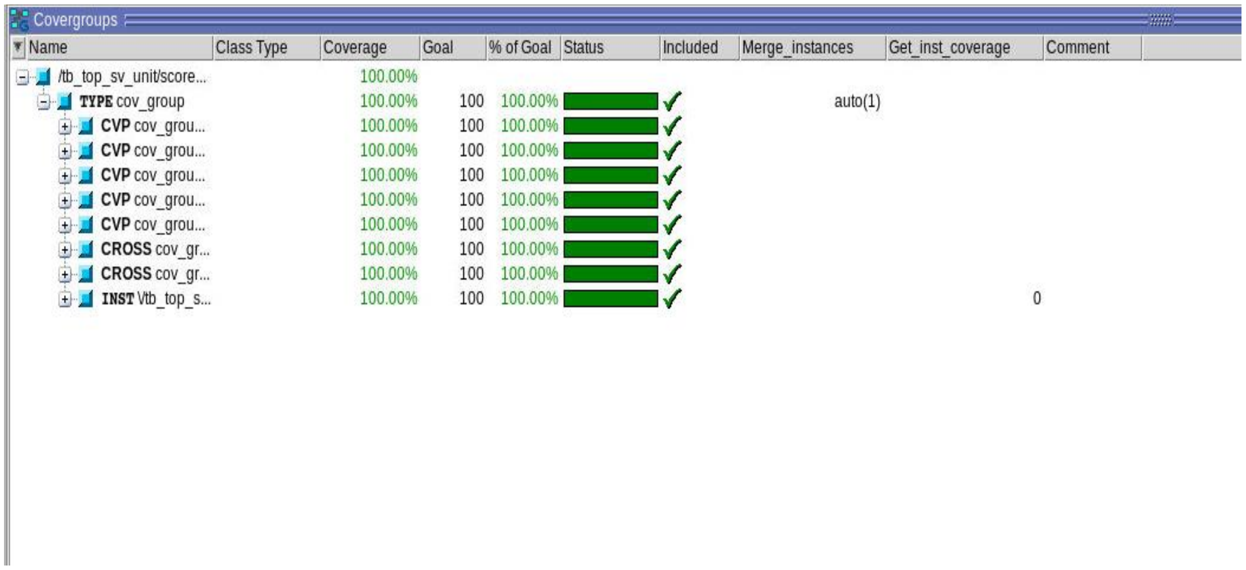
Bins: Each coverpoint uses bins to partition the value space into subsets. Each bin captures a specific value or a range of values associated with the coverpoint. For example, the write_val bin for the bus_write coverpoint captures the scenario where HWRITE is 1, indicating a write operation.

Cross Coverage: This feature allows tracking combinations of multiple coverpoints. In our covergroup, WRITE_COVERAGE tracks combinations of write operations (bus_write) with different transaction types (trans_type). Similarly, READ_COVERAGE tracks combinations of read operations with various transaction types. H_Data tracks the range of data values taken by HWDATA.



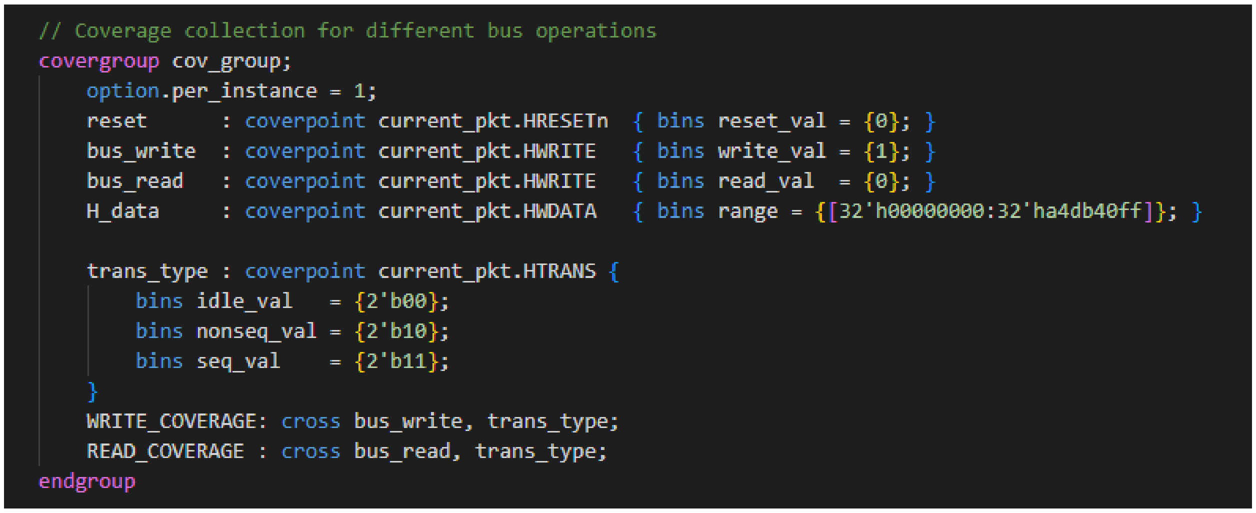
```
Coverage Report Summary Data by DU

=====================================================================================
=== Design Unit: work.Bridge_Top
=====================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Toggles                        880       272       608    30.90%

=====================================================================================
=== Design Unit: work.tb_top
=====================================================================================
    Enabled Coverage              Bins      Hits    Misses  Coverage
    ----------------              ----      ----    ------  --------
    Statements                       7         7         0   100.00%
    Toggles                          2         2         0   100.00%


Total Coverage By Design Unit (filtered view): 65.53%
```

Achieved a **65.53% code coverage**, a significant improvement from the initial **21% coverage**. Increasing coverage was challenging due to **limited toggle activity**, which restricted the ability to exercise certain logic paths effectively. Additionally, **timing constraints** posed challenges in reaching deeper states of the design, as some test scenarios were restricted by timing violations. Other factors, such as **unreachable conditions in the RTL and constraints in the testbench stimulus**, further limited coverage improvement.


Covergroups

| Name | Class Type | Coverage | Goal | % of Goal | Status | Included | Merge_instances | Get_inst_coverage | Comment |
|---|---|---|---|---|---|---|---|---|---|
| /tb_top_sv_unit/score... | | 100.00% | | | | | | | |
| TYPE cov_group | | 100.00% | 100 | 100.00% | 🟩 | ✓ | auto(1) | | |
| CVP cov_grou... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | | |
| CVP cov_grou... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | | |
| CVP cov_grou... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | | |
| CVP cov_grou... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | | |
| CVP cov_grou... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | | |
| CROSS cov_gr... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | | |
| CROSS cov_gr... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | | |
| INST \/tb_top_s... | | 100.00% | 100 | 100.00% | 🟩 | ✓ | | 0 | |

Achieved 100% functional coverage with the following coverpoints and bins.



```
// Coverage collection for different bus operations
covergroup cov_group;
    option.per_instance = 1;
    reset      : coverpoint current_pkt.HRESETn  { bins reset_val = {0}; }
    bus_write  : coverpoint current_pkt.HWRITE   { bins write_val = {1}; }
    bus_read   : coverpoint current_pkt.HWRITE   { bins read_val  = {0}; }
    H_data     : coverpoint current_pkt.HWDATA   { bins range = {[32'h00000000:32'ha4db40ff]}; }

    trans_type : coverpoint current_pkt.HTRANS {
        bins idle_val    = {2'b00};
        bins nonseq_val = {2'b10};
        bins seq_val     = {2'b11};
    }
    WRITE_COVERAGE: cross bus_write, trans_type;
    READ_COVERAGE : cross bus_read, trans_type;
endgroup
```

Additionally we have also added assertions coverage and achieved 100% coverage for assertions as well and we used assertions to catch functional bugs in the RTL.

**Assertions:**

The assertions cover the following key functional areas:

1. AHB Protocol Compliance:
    - Ensures HTRANS carries valid transaction types (IDLE, NONSEQ, SEQ).
    - Ensures HRESP remains OKAY during normal operation.
2. Data Transfer Mechanisms:
    - Validates address pipelining by ensuring Haddr1 retains the previous address value correctly.
    - Checks the correct generation of the valid signal based on address ranges and HTRANS types.
3. Reset Behavior:
    - Ensures that Penable is de-asserted when Hresetn is active low.
    - Verifies that reset transitions properly restore the bridge to a known state.
4. Basic Transaction Validations:
    - Asserts that a basic write transaction (Hwrite asserted with NONSEQ transfer) completes successfully with Hreadyout.

    ○ Asserts that a basic read transaction (non-Hwrite with NONSEQ transfer) completes successfully with Hreadyout.

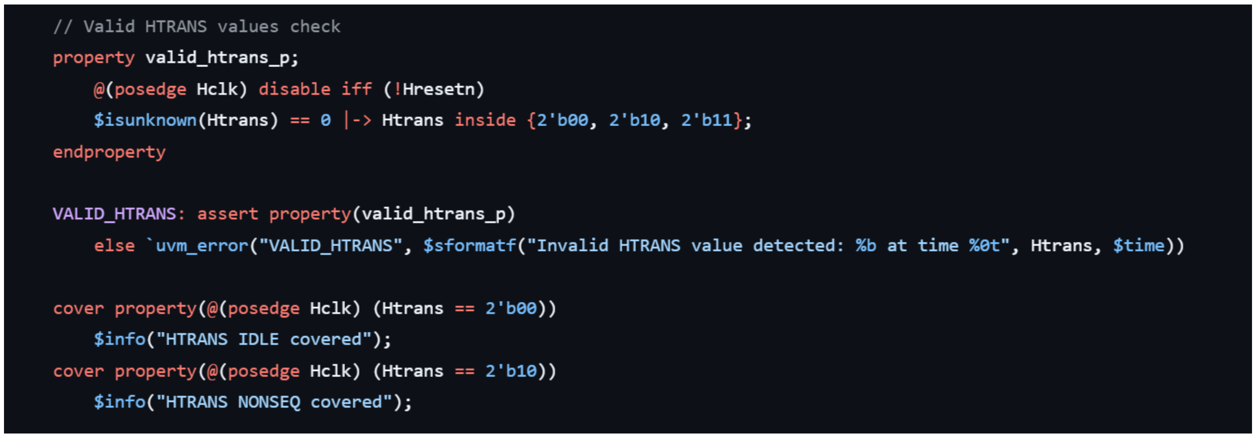
**Assertion Coverage Goals:**

The verification plan includes **functional coverage properties** to ensure all major scenarios are exercised during simulation:

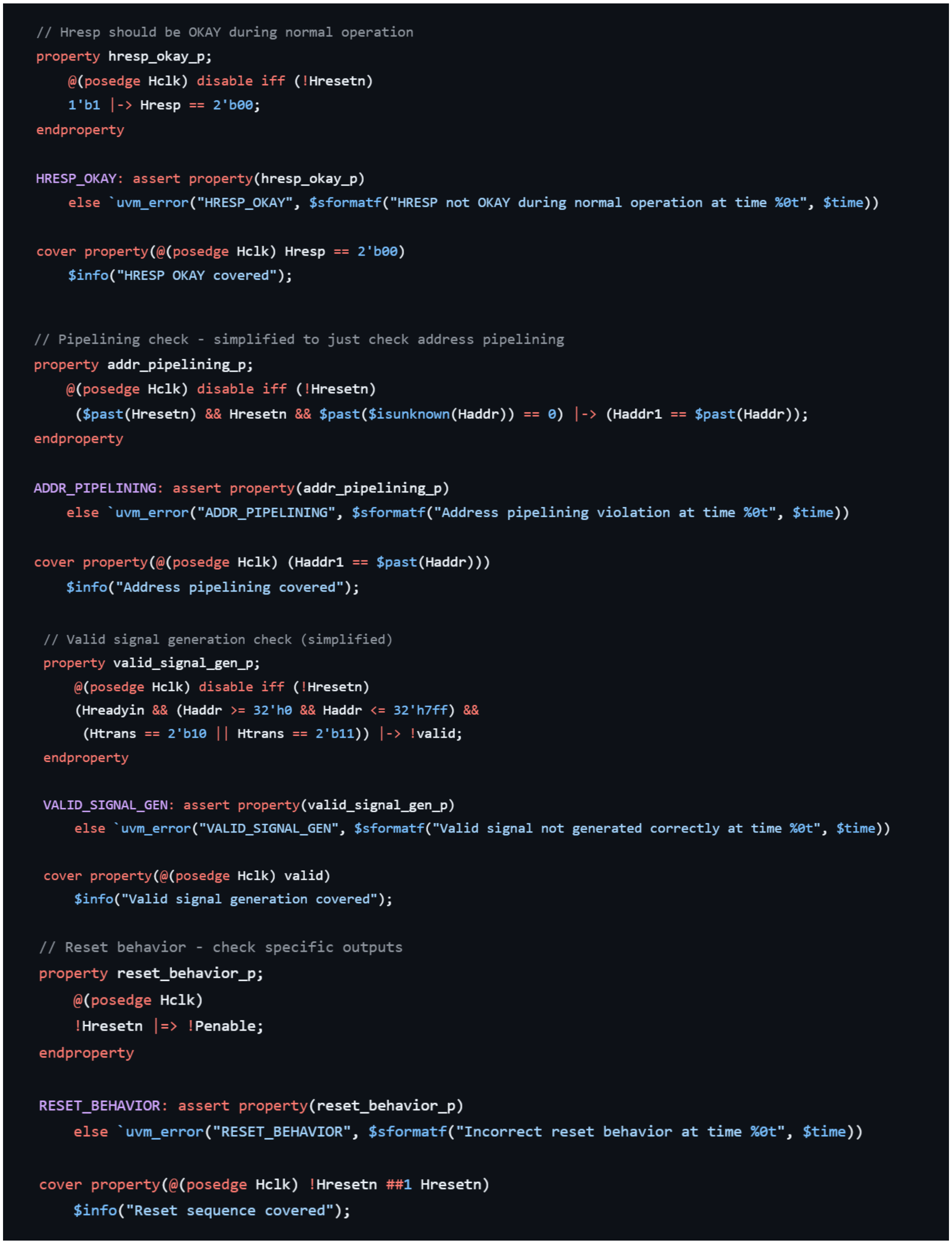
- Coverage points for HTRANS values (IDLE, NONSEQ, SEQ).
- Ensuring HRESP remains OKAY under normal conditions.
- Address pipelining validation.
- Checking that reset conditions are met and properly restored.
- Basic read/write transaction completion.

**Pass/Fail Criteria:**

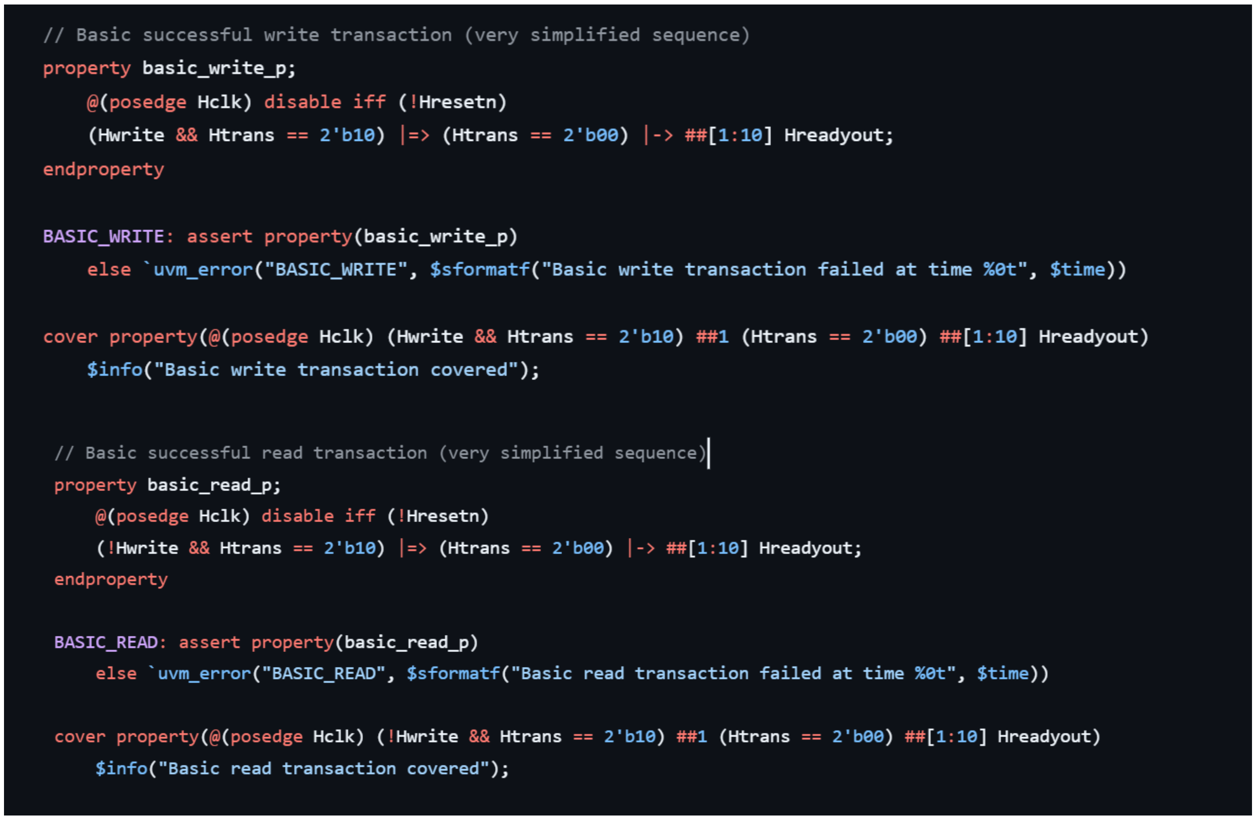
- A simulation run with **no assertion failures** indicates correct bridge functionality.
- Functional coverage should achieve **100% assertion coverage** for all defined properties.
- UVM error logs (uvm_error) should not report any assertion violations during test execution.
- Coverage metrics should be analyzed to identify potential gaps in the verification plan.



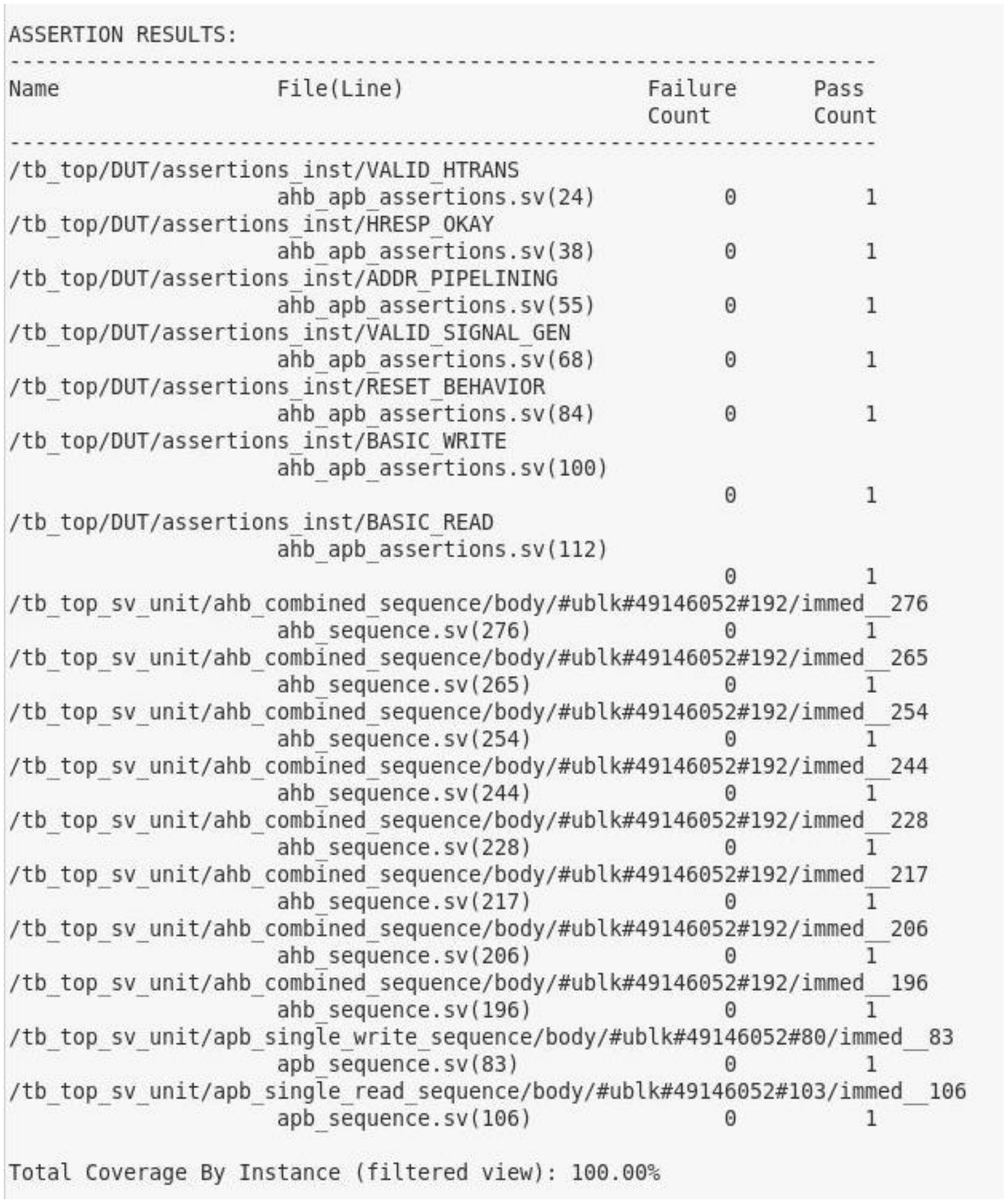
```
// Valid HTRANS values check
property valid_htrans_p;
    @(posedge Hclk) disable iff (!Hresetn)
    $isunknown(Htrans) == 0 |-> Htrans inside {2'b00, 2'b10, 2'b11};
endproperty

VALID_HTRANS: assert property(valid_htrans_p)
    else `uvm_error("VALID_HTRANS", $sformatf("Invalid HTRANS value detected: %b at time %0t", Htrans, $time))

cover property(@(posedge Hclk) (Htrans == 2'b00))
    $info("HTRANS IDLE covered");
cover property(@(posedge Hclk) (Htrans == 2'b10))
    $info("HTRANS NONSEQ covered");
```



```
// Hresp should be OKAY during normal operation
property hresp_okay_p;
    @(posedge Hclk) disable iff (!Hresetn)
    1'b1 |-> Hresp == 2'b00;
endproperty

HRESP_OKAY: assert property(hresp_okay_p)
    else `uvm_error("HRESP_OKAY", $sformatf("HRESP not OKAY during normal operation at time %0t", $time))

cover property(@(posedge Hclk) Hresp == 2'b00)
    $info("HRESP OKAY covered");


// Pipelining check - simplified to just check address pipelining
property addr_pipelining_p;
    @(posedge Hclk) disable iff (!Hresetn)
     ($past(Hresetn) && Hresetn && $past($isunknown(Haddr)) == 0) |-> (Haddr1 == $past(Haddr));
endproperty

ADDR_PIPELINING: assert property(addr_pipelining_p)
    else `uvm_error("ADDR_PIPELINING", $sformatf("Address pipelining violation at time %0t", $time))

cover property(@(posedge Hclk) (Haddr1 == $past(Haddr)))
    $info("Address pipelining covered");

 // Valid signal generation check (simplified)
 property valid_signal_gen_p;
     @(posedge Hclk) disable iff (!Hresetn)
     (Hreadyin && (Haddr >= 32'h0 && Haddr <= 32'h7ff) &&
      (Htrans == 2'b10 || Htrans == 2'b11)) |-> !valid;
 endproperty

 VALID_SIGNAL_GEN: assert property(valid_signal_gen_p)
     else `uvm_error("VALID_SIGNAL_GEN", $sformatf("Valid signal not generated correctly at time %0t", $time))

 cover property(@(posedge Hclk) valid)
     $info("Valid signal generation covered");

 // Reset behavior - check specific outputs
 property reset_behavior_p;
     @(posedge Hclk)
     !Hresetn |=> !Penable;
 endproperty

 RESET_BEHAVIOR: assert property(reset_behavior_p)
     else `uvm_error("RESET_BEHAVIOR", $sformatf("Incorrect reset behavior at time %0t", $time))

 cover property(@(posedge Hclk) !Hresetn ##1 Hresetn)
     $info("Reset sequence covered");
```



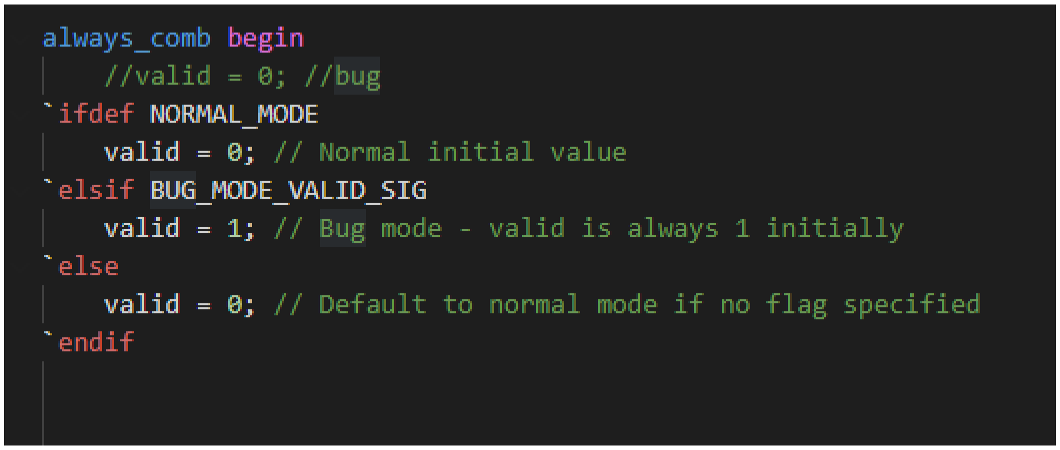
```
// Basic successful write transaction (very simplified sequence)
property basic_write_p;
    @(posedge Hclk) disable iff (!Hresetn)
    (Hwrite && Htrans == 2'b10) |=> (Htrans == 2'b00) |-> ##[1:10] Hreadyout;
endproperty

BASIC_WRITE: assert property(basic_write_p)
    else `uvm_error("BASIC_WRITE", $sformatf("Basic write transaction failed at time %0t", $time))

cover property(@(posedge Hclk) (Hwrite && Htrans == 2'b10) ##1 (Htrans == 2'b00) ##[1:10] Hreadyout)
    $info("Basic write transaction covered");


 // Basic successful read transaction (very simplified sequence)
 property basic_read_p;
     @(posedge Hclk) disable iff (!Hresetn)
     (!Hwrite && Htrans == 2'b10) |=> (Htrans == 2'b00) |-> ##[1:10] Hreadyout;
 endproperty

 BASIC_READ: assert property(basic_read_p)
     else `uvm_error("BASIC_READ", $sformatf("Basic read transaction failed at time %0t", $time))

 cover property(@(posedge Hclk) (!Hwrite && Htrans == 2'b10) ##1 (Htrans == 2'b00) ##[1:10] Hreadyout)
     $info("Basic read transaction covered");
```



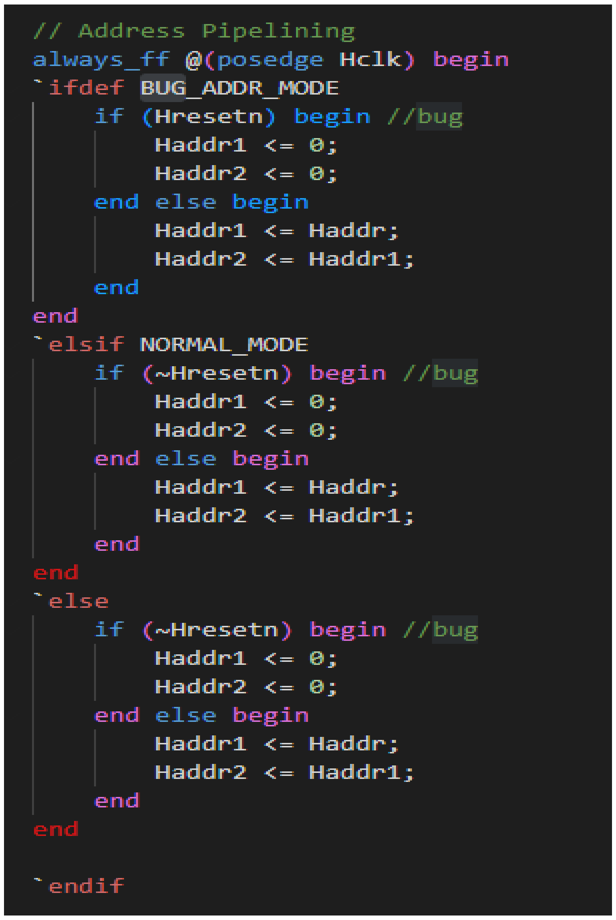
```
ASSERTION RESULTS:
-------------------------------------------------------------------
Name                 File(Line)                  Failure      Pass
                                                 Count        Count
-------------------------------------------------------------------
/tb_top/DUT/assertions_inst/VALID_HTRANS
                     ahb_apb_assertions.sv(24)         0            1
/tb_top/DUT/assertions_inst/HRESP_OKAY
                     ahb_apb_assertions.sv(38)         0            1
/tb_top/DUT/assertions_inst/ADDR_PIPELINING
                     ahb_apb_assertions.sv(55)         0            1
/tb_top/DUT/assertions_inst/VALID_SIGNAL_GEN
                     ahb_apb_assertions.sv(68)         0            1
/tb_top/DUT/assertions_inst/RESET_BEHAVIOR
                     ahb_apb_assertions.sv(84)         0            1
/tb_top/DUT/assertions_inst/BASIC_WRITE
                     ahb_apb_assertions.sv(100)
                                                       0            1
/tb_top/DUT/assertions_inst/BASIC_READ
                     ahb_apb_assertions.sv(112)
                                                       0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__276
                     ahb_sequence.sv(276)              0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__265
                     ahb_sequence.sv(265)              0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__254
                     ahb_sequence.sv(254)              0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__244
                     ahb_sequence.sv(244)              0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__228
                     ahb_sequence.sv(228)              0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__217
                     ahb_sequence.sv(217)              0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__206
                     ahb_sequence.sv(206)              0            1
/tb_top_sv_unit/ahb_combined_sequence/body/#ublk#49146052#192/immed__196
                     ahb_sequence.sv(196)              0            1
/tb_top_sv_unit/apb_single_write_sequence/body/#ublk#49146052#80/immed__83
                     apb_sequence.sv(83)               0            1
/tb_top_sv_unit/apb_single_read_sequence/body/#ublk#49146052#103/immed__106
                     apb_sequence.sv(106)              0            1

Total Coverage By Instance (filtered view): 100.00%
```

## Bug Injection

We have injected bugs in our RTL, in the address pipelining logic and the valid signal generation logic.



```
always_comb begin
    //valid = 0; //bug
`ifdef NORMAL_MODE
    valid = 0; // Normal initial value
`elsif BUG_MODE_VALID_SIG
    valid = 1; // Bug mode - valid is always 1 initially
`else
    valid = 0; // Default to normal mode if no flag specified
`endif
```

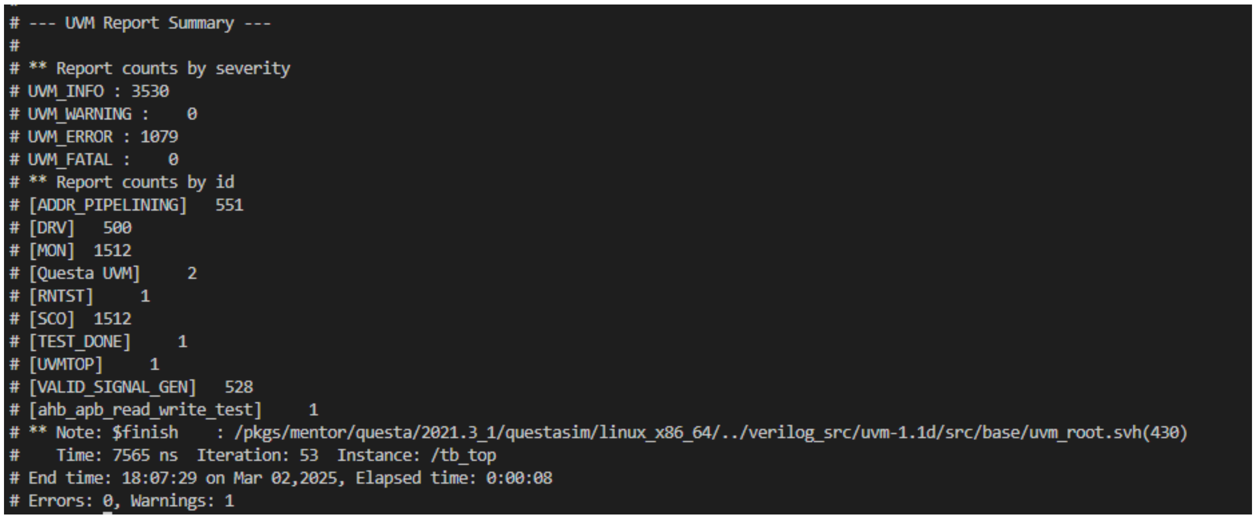


```
// Address Pipelining
always_ff @(posedge Hclk) begin
`ifdef BUG_ADDR_MODE
    if (Hresetn) begin //bug
        Haddr1 <= 0;
        Haddr2 <= 0;
    end else begin
        Haddr1 <= Haddr;
        Haddr2 <= Haddr1;
    end
end
`elsif NORMAL_MODE
    if (~Hresetn) begin //bug
        Haddr1 <= 0;
        Haddr2 <= 0;
    end else begin
        Haddr1 <= Haddr;
        Haddr2 <= Haddr1;
    end
end
`else
    if (~Hresetn) begin //bug
        Haddr1 <= 0;
        Haddr2 <= 0;
    end else begin
        Haddr1 <= Haddr;
        Haddr2 <= Haddr1;
    end
end

`endif
```

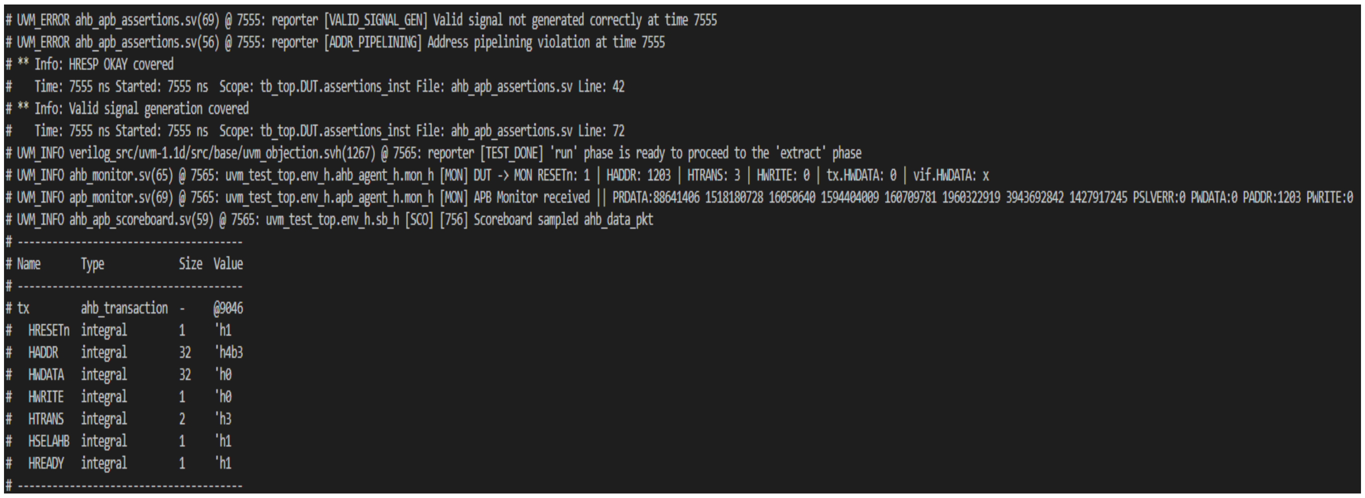
These injected bugs are clearly marked with comment lines (//bug) for easy identification. The **functional assertions** written in the testbench will monitor these logic errors and trigger the uvm_error reporting macro upon detection. This ensures that any deviations from expected behavior are **immediately flagged** during simulation, allowing us to debug and validate the design efficiently.

We utilized conditional compilation with `ifdef..`endif to inject bugs and have three separate run.do files for this namely, run_normalMode.do, run_bugData.do and run_bugValidSig.do.

Output with functional bugs,



```
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO : 3530
# UVM_WARNING :    0
# UVM_ERROR : 1079
# UVM_FATAL :    0
# ** Report counts by id
# [ADDR_PIPELINING]   551
# [DRV]   500
# [MON]  1512
# [Questa UVM]     2
# [RNTST]     1
# [SCO]  1512
# [TEST_DONE]     1
# [UVMTOP]     1
# [VALID_SIGNAL_GEN]   528
# [ahb_apb_read_write_test]     1
# ** Note: $finish    : /pkgs/mentor/questa/2021.3_1/questasim/linux_x86_64/../verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)
#    Time: 7565 ns  Iteration: 53  Instance: /tb_top
# End time: 18:07:29 on Mar 02,2025, Elapsed time: 0:00:08
# Errors: 0, Warnings: 1
```



```
# UVM_ERROR ahb_apb_assertions.sv(69) @ 7555: reporter [VALID_SIGNAL_GEN] Valid signal not generated correctly at time 7555
# UVM_ERROR ahb_apb_assertions.sv(56) @ 7555: reporter [ADDR_PIPELINING] Address pipelining violation at time 7555
# ** Info: HRESP OKAY covered
#    Time: 7555 ns Started: 7555 ns  Scope: tb_top.DUT.assertions_inst File: ahb_apb_assertions.sv Line: 42
# ** Info: Valid signal generation covered
#    Time: 7555 ns Started: 7555 ns  Scope: tb_top.DUT.assertions_inst File: ahb_apb_assertions.sv Line: 72
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 7565: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ahb_monitor.sv(65) @ 7565: uvm_test_top.env_h.ahb_agent_h.mon_h [MON] DUT -> MON RESETn: 1 | HADDR: 1203 | HTRANS: 3 | HWRITE: 0 | tx.HWDATA: 0 | vif.HWDATA: x
# UVM_INFO apb_monitor.sv(69) @ 7565: uvm_test_top.env_h.apb_agent_h.mon_h [MON] APB Monitor received || PRDATA:88641406 1518180728 16050640 1594404009 160709781 1960322919 3943692842 1427917245 PSLVERR:0 PWDATA:0 PADDR:1203 PWRITE:0
# UVM_INFO ahb_apb_scoreboard.sv(59) @ 7565: uvm_test_top.env_h.sb_h [SCO] [756] Scoreboard sampled ahb_data_pkt
# --------------------------------------
# Name       Type            Size  Value
# --------------------------------------
# tx         ahb_transaction -     @9046
#   HRESETn  integral        1     'h1
#   HADDR    integral        32    'h4b3
#   HWDATA   integral        32    'h0
#   HWRITE   integral        1     'h0
#   HTRANS   integral        2     'h3
#   HSELAHB  integral        1     'h1
#   HREADY   integral        1     'h1
# --------------------------------------
```

## Resource Requirements

a) Project Members
   - Neil Tauro
     - M1 - Design Specification and first half of Bridge controller
     - M2 and M3 - Scoreboard and Design specification
     - M4 - Scoreboard and functional coverage
     - M5 - Presentation slides.

   - Suraj Shetty
     - M1- Design Specification and APB slave interface
     - M2 and M3 - Monitor and Design Specification
     - M4 - Transaction, interface and monitor classes.
     - M5 - Assertions, functional coverage and modifications to test sequences.

   - Maithreyi Venkatesan
     - M1 -Verification plan and Testbench development
     - M2 and M3 - Driver, functional coverage, test class and Verification plan.
     - M4 - APB sequence, driver and monitor components.
     - M5 - Presentation slides and bug injection.

   - Raghavendra Davarapalli
     - M1 - Verification plan, Interface for modules and AHB Master module
     - M2 and M3 - Transaction, Generator, Environment classes and Verification plan.
     - M4 - AHB sequence, driver, monitor, environment and test components.
     - M5 - Presentation slides and bug injection.

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

c) Simulator license
   - QuestaSim provided by PSU remote systems.

**<u>Schedule</u>**

| **Milestones** | **Date** | **Objectives** |
|---|---|---|
| Milestone 1 | 01/31/2025 | 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 | 02/18/2025 | Class based testbench with all components and interfaces. Should verify ~50 random bursts of data. |
| Milestone 3 | 02/18/2025 | Finalize changes in RTL. code and functional coverage. Update verification plan document. |
| Milestone 4 | 02/26/2025 | UVM testbench along with updated verification plan. UVM reporting macros to log reports/data to the console. |
| Final Deliverables | 03/04/2025 | Complete UVM architecture, environment and testbench. Bug injection. Finalized documents and presentations. |

## References and Citations

- Utilized ClaudeAI for adding comments into the code and paraphrasing the document.
- RTL Design code referenced from "https://github.com/prajwalgekkouga/AHB-to-APB-Bridge"
- ARM Documentation "https://developer.arm.com/documentation/ihi0033/latest/"
- UVM testbench scoreboard model referenced from "https://github.com/Ghonimo/Pre_Silicon-AHB-to_APB-Verification/blob/main/Checkpoint%204_UVM_Based%20Testbench/AHB_APB_UVM_VIP/ahb_apb_scoreboard.sv"