



Maseeh College of Engineering
and Computer Science

PORTLAND STATE UNIVERSITY

**ECE 593: SUMMER 2023 - CLASS PROJECT
FUNDAMENTALS OF PRE-SILICON VALIDATION**

VERIFICATION OF AN AHB2APB BRIDGE

CHECKPOINT 4 UVM-BASED VERIFICATION IP

MOHAMED GHONIM

08/19/2023

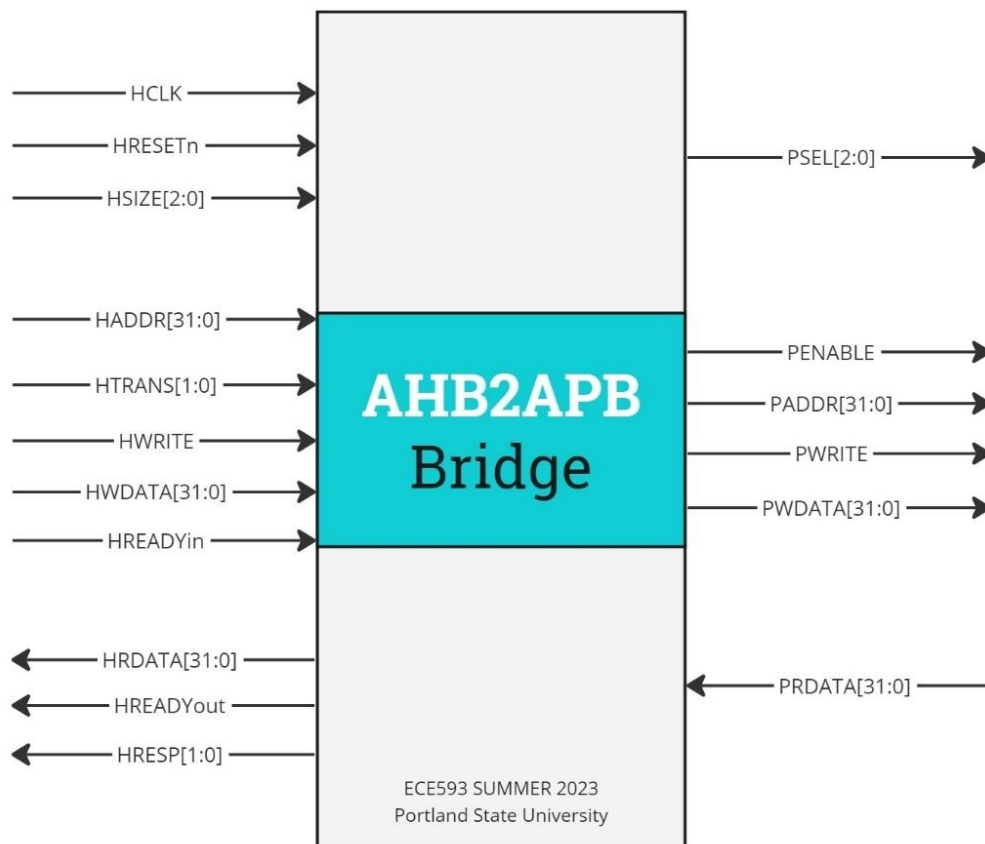
Table of Contents

Introduction	3
Verification IP Structure	4
Verification Methodology:	5
1. UVM Framework:	5
2. Testbench Architecture:	5
3. Stimulus Generation:	5
4. Configuration Database:	5
Verification Components:	6
Top Module:	6
AHB-APB Verification Environment (ahb_apb_env).....	6
Key Components:	6
Key Methods:	7
AHB-APB Scoreboard (ahb_apb_scoreboard).....	7
Core Components:	7
Primary Methods:	8
Base Test Class (ahb_apb_base_test).....	8
Core Components:	8
Primary Methods:	8
Random Test Class (ahb_apb_random_test)	9
Core Components:	9
Primary Methods:	9
Single Write Test Class (ahb_apb_single_write_test)	9
Primary Methods:	10
Single Read Test Class (ahb_apb_single_read_test)	10
Core Components:	10
Primary Methods:	10
Burst Write Test Class (ahb_apb_burst_write_test).....	11
Core Components:	11
Primary Methods:	11
Burst Read Test Class (ahb_apb_burst_read_test).....	11
Core Components:	11

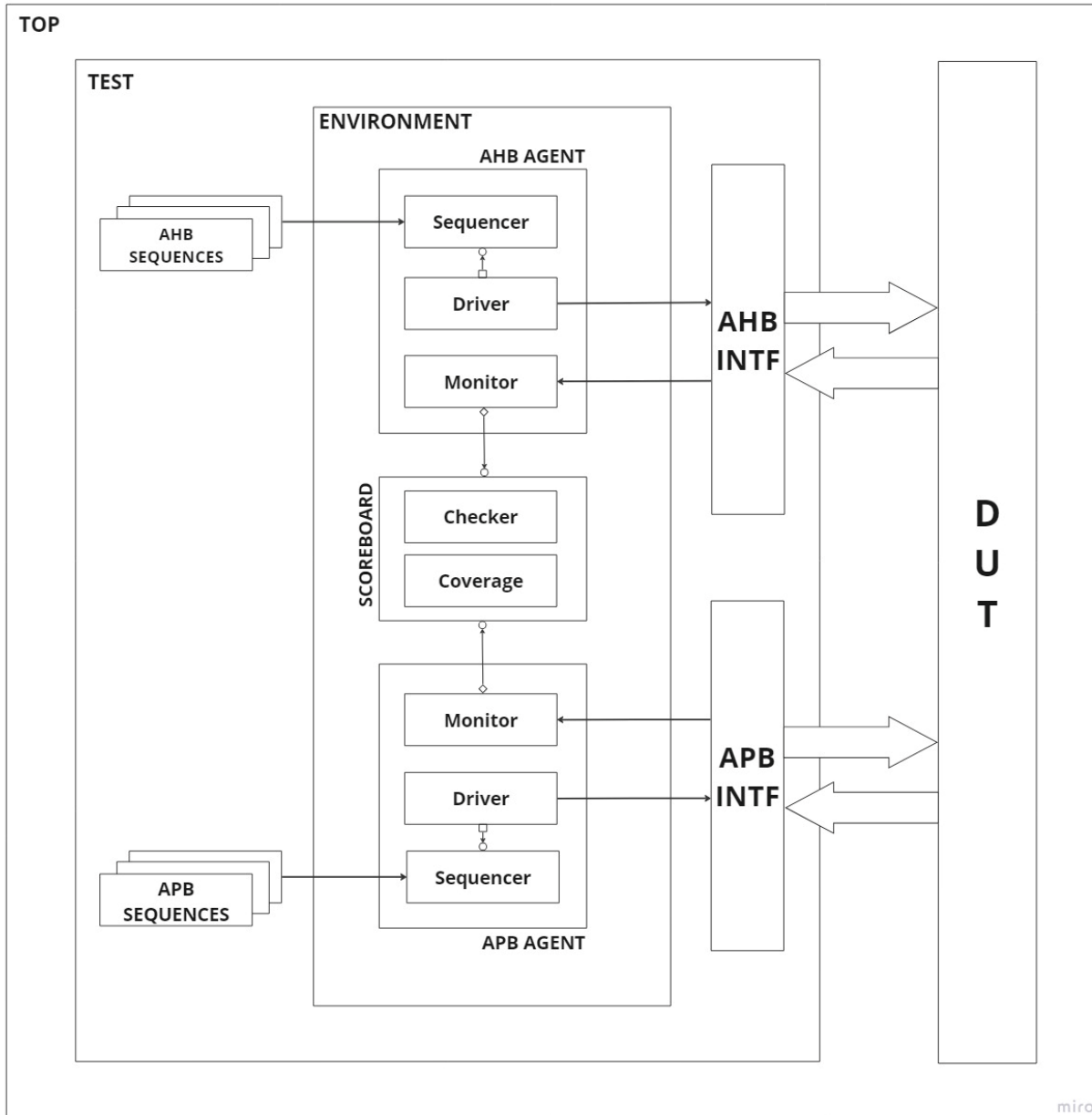
Primary Methods:	12
UVM Imports and Macros:	12
Global Settings:.....	12
Sequence Items and Configuration:	12
AHB Components:	12
APB Components:.....	13
Assertions:	13
Assertions in AHB and APB Sequences	13
AHB to APB FSM assertions	16
Coverage	17
Running the Verification Environment.....	19
1. The Makefile:.....	19
2. Flexible Test Configuration:	19
Results and Discussion:	20
Debugging the Transaction Print Method:	20
Enhancing Transaction Visibility:	21
Scoreboard Insights:	21
Debugging	34
The Journey from SystemVerilog to UVM:	34
Challenges Encountered:	35
Contrasting the Old and New Designs:	35
Summary	36
Conclusion.....	36

Introduction

This report covers our next step in verifying the AHB-APB Bridge: moving from a SystemVerilog-based setup to one using the Universal Verification Methodology (UVM). UVM is a widely accepted standard in the world of verification, designed to handle the increasing complexities of modern digital designs. In our previous phase, Checkpoint 3, we built a verification environment using traditional SystemVerilog techniques. With Checkpoint 4, we're taking things up a notch. We're using UVM to create a structured environment with specialized components like agents, sequencers, and a more advanced scoreboard. This new setup not only makes the verification process smoother but also gives us the flexibility to reuse and adjust components for different test scenarios. All of this ensures that our AHB-APB Bridge design is thoroughly and efficiently tested, aligning with the best practices in the industry.



AHB to APB Bridge UVM-Based Verification IP



Verification Methodology:

1. UVM Framework:

UVM is not just a methodology; it's a framework that has been co-developed by the semiconductor industry's leading experts. It provides a blueprint for creating highly reusable and modular verification environments. With its layered approach, components in UVM can be easily replaced, updated, or reused across various projects, leading to significant time savings and consistency across verification projects.

2. Testbench Architecture:

Our UVM-based testbench is structured meticulously to mirror the inherent modularity and hierarchy of UVM. At its core, it includes:

Agents: These are modular blocks that can operate in active (driving stimulus) or passive (only monitoring) modes, enhancing reusability across different test scenarios.

Sequencers: Responsible for generating and randomizing the stimulus based on specific test scenarios or sequences.

Drivers: They take the sequences from sequencers and drive them onto the DUT interfaces.

Monitors: Continuously observe the DUT's interfaces, capturing its responses and behavior.

Scoreboard: A critical component that compares the DUT's responses with expected results, flagging any discrepancies.

UVM Tests: These encapsulate specific test scenarios, leveraging the above components to stimulate and verify the DUT for that particular scenario.

3. Stimulus Generation:

Generating varied and random stimulus is at the heart of coverage-driven verification. Our sequences are designed to produce both targeted (for specific test scenarios) and randomized stimulus to ensure the DUT is tested exhaustively. The global `N_TX` allows testers to easily scale the number of transactions based on their testing needs.

4. 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.

Verification Components:

Top Module:

The top module, **tb_top**, serves as the foundation for the UVM-based verification of the AHB-APB Bridge. Acting as the primary orchestrator, it integrates the various UVM components, sequences, and configurations, ensuring a seamless interaction with the Device Under Verification (DUT) - the AHB-APB Bridge, here named **Bridge_Top**.

Upon activation, **tb_top** commences by generating a clock signal, **clk**, toggling every 5 ns, mirroring a real-world 10 ns clock cycle. This clock is pivotal for steering the synchronous operations within the DUT. The module also configures the virtual interfaces, ensuring the UVM environment can communicate with the DUT through the AHB and APB interfaces, named **AHB_INF** and **APB_INF**, respectively.

The module imports a series of files, each encapsulating a distinct facet of the UVM verification environment. From sequence items like **ahb_sequence_item** to more intricate components like the **ahb_apb_scoreboard**, each import plays a pivotal role in the intricate dance of the verification process.

The actual DUT, **Bridge_Top**, is then instantiated and connected to the UVM environment via the AHB and APB interfaces, which act as the communication bridges, simulating real-world bus transactions.

To initiate the verification process, the **run_test** method is called, triggering the desired UVM test sequence. This method streamlines the entire verification process from generating transactions, interacting with the DUT, monitoring its outputs, and verifying its behavior against expectations.

AHB-APB Verification Environment (**ahb_apb_env**)

The **ahb_apb_env** class represents the core verification environment for the AHB-APB Bridge. This environment encapsulates all the essential verification components, including agents for both the AHB and APB protocols and a scoreboard for results validation. The environment is crucial as it helps in setting up, connecting, and orchestrating the verification process.

Key Components:

1. **Configuration Handle (**env_config_h**):** This handle provides access to the environment's configuration, which dictates the behavior and structure of the environment. The configuration settings determine which agents or components are active during a simulation run.
2. **AHB and APB Agents (**ahb_agent_h**, **apb_agent_h**):** These agents are responsible for driving and monitoring transactions on the AHB and APB interfaces, respectively.

Each agent consists of a driver to send transactions and a monitor to observe the interactions on the bus.

3. **Scoreboard (sb_h):** The scoreboard is tasked with checking and validating the behavior of the DUT. It receives transactions from the monitors and checks if the observed outputs match the expected results.

Key Methods:

1. **Constructor (new):** The constructor initializes the environment component, setting its name and parent.
2. **Build Phase (build_phase):** In this phase, the environment components are instantiated based on the configuration settings. The environment checks the settings to determine which agents and the scoreboard should be created. For instance, if the AHB agent is enabled in the configuration, the AHB agent is instantiated. The same logic applies to the APB agent and the scoreboard.
3. **Connect Phase (connect_phase):** This phase is responsible for connecting the various components together. If both the AHB agent and the scoreboard are enabled, the AHB monitor's output port gets connected to the scoreboard's AHB analysis port, allowing the observed AHB transactions to be sent to the scoreboard for checking. Similarly, the APB monitor's output port is connected to the scoreboard's APB analysis port if both the APB agent and the scoreboard are enabled.

AHB-APB Scoreboard (ahb_apb_scoreboard)

The **ahb_apb_scoreboard** class is instrumental in the verification flow as it serves the primary function of observing, predicting, and validating the data flow between the AHB and APB interfaces.

Core Components:

1. **FIFOs (ahb_fifo and apb_fifo):** These First-In-First-Out structures store the transactions collected from the AHB and APB monitors. They act as buffers, ensuring that the transactions are processed in the order they arrive.
2. **Packet Variables:**
These are used to hold incoming packets (**ahb_data_pkt** and **apb_data_pkt**), temporary data (**ahb_temp_data**), and the predicted packets (**ahb_predicted_pkt** and **apb_predicted_pkt**).
3. **Counters:** Variables like **ahb_pkt_count**, **apb_pkt_count**, and **verified_data_count** track the total number of packets processed and the number of correctly verified transactions.
4. **Covergroup (cov_group):** This captures the functional coverage metrics, giving insight into which parts of the design have been exercised and which have not. The

coverage metrics include checking the reset, read, write operations, and transaction types.

Primary Methods:

1. **Constructor (new):** Initializes the FIFOs, sets up the variables for packets, and initializes the coverage group.
2. **Run Phase (run_phase):** This is the active phase where the scoreboard continuously samples packets from both the AHB and APB monitors. Each sampled packet is logged, and the expected data is predicted based on the sampled data. Functional coverage is also sampled in this phase.
3. **Predict Data (predict_data):** This method predicts the expected data for the AHB and APB based on the current transactions. The predictions are based on the type of transaction, be it an AHB read or write operation.
4. **Configure PSELx (configure_pselx):** This method sets the PSELx signal based on the AHB address, determining which slave the data is directed to on the APB side.
5. **Check Data Methods (check_apb_data and check_ahb_data):** These methods validate the observed data against the predicted values. If the observed data matches the prediction, the **verified_data_count** is incremented.
6. **Report Phase (report_phase):** At the end of the simulation, this function displays a summary of the scoreboard's observations, including the total number of packets processed, the number of verified transactions, and a coverage report.

Base Test Class (ahb_apb_base_test)

The **ahb_apb_base_test** class establishes the foundation for subsequent test scenarios. This class sets up the necessary environment, configurations, and interfaces for the AHB-APB testbench.

Core Components:

1. **Environment Configuration (env_config_h):** This handle is associated with the main environment configuration which defines the setup for the testbench, including the status of agents, interfaces, and the scoreboard.
2. **Main Testbench Environment (env_h):** This handle is associated with the main testbench environment where the actual verification takes place.

Primary Methods:

1. **Constructor (new):** Sets up the name and parent for the test class instance.
2. **Build Phase (build_phase):**
 - Creates the environment configuration object.

- Retrieves the virtual interfaces (**ahb_vif** and **apb_vif**) for AHB and APB from the configuration database.
- Sets the AHB and APB agents to active mode.
- Instantiates the main testbench environment (**env_h**).

Random Test Class (**ahb_apb_random_test**)

The **ahb_apb_random_test** class is derived from the base test class and aims to introduce randomized traffic on both AHB and APB interfaces. It helps in verifying how the DUT behaves under unpredictable scenarios, adding robustness to the verification process.

Core Components:

1. **Random Sequences (**ahb_rand_seq_h** and **apb_rand_seq_h**):** These handles are linked to sequences that generate randomized traffic on the AHB and APB interfaces respectively.

Primary Methods:

1. **Constructor (**new**):** Defines the name and parent for the random test class instance.
2. **Build Phase (**build_phase**):**
 - Inherits the build phase of the base test by calling **super.build_phase(phase)**.
 - Instantiates the random sequences for AHB and APB interfaces.
3. **Run Phase (**run_phase**):**
 - Raises an objection to keep the simulation running.
 - Executes the random sequences on their respective sequencers in parallel using the **fork...join** construct.
 - Drops the objection once the sequences have finished running.
 - Sets a drain time to allow any remaining transactions to be processed before the simulation ends.

Single Write Test Class (**ahb_apb_single_write_test**)

The **ahb_apb_single_write_test** class, derived from the foundational **ahb_apb_base_test**, is dedicated to verifying the behavior of the DUT during single write operations on the AHB and APB interfaces.

Core Components:

1. **Single Write Sequences (**ahb_seq_h** and **apb_seq_h**):** These handles link to sequences that generate single write traffic on the AHB and APB interfaces, respectively.

Primary Methods:

1. **Constructor (new):** Establishes the name and parent for the single write test class instance.
2. **Build Phase (build_phase):**
 - Inherits the build phase of the base test by invoking **super.build_phase(phase)**.
 - Creates instances of the single write sequences for both AHB and APB interfaces.
3. **Run Phase (run_phase):**
 - Raises an objection to ensure the simulation remains active.
 - Launches the single write sequences on their respective sequencers concurrently using the **fork...join** construct.
 - Drops the objection once the sequences have concluded.
 - Assigns a drain time to allow the processing of any residual transactions before the simulation terminates.

Single Read Test Class (ahb_apb_single_read_test)

The **ahb_apb_single_read_test** class, which extends the **ahb_apb_base_test**, focuses on verifying 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 handles correspond to sequences that produce single read traffic on the AHB and APB interfaces, respectively.

Primary Methods:

1. **Constructor (new):** Sets up the name and parent for the single read test class instance.
2. **Build Phase (build_phase):**
 - Inherits the base test's build phase using **super.build_phase(phase)**.
 - Instantiates the single read sequences for both AHB and APB interfaces.
3. **Run Phase (run_phase):**
 - Raises an objection to keep the simulation running.
 - Initiates the single read sequences on their respective sequencers simultaneously using the **fork...join** construct.

- Drops the objection post completion of the sequences.
- Sets a drain time to ensure any lingering transactions are fully processed prior to the end of the simulation.

Burst Write Test Class (`ahb_apb_burst_write_test`)

The **`ahb_apb_burst_write_test`** class, a derivative of **`ahb_apb_base_test`**, is specifically designed to verify the behavior of the DUT during burst write operations on the AHB and APB interfaces.

Core Components:

1. **Burst Write Sequences (`ahb_seq_h` and `apb_seq_h`):** These handles point to sequences that generate burst write traffic on the AHB and APB interfaces, respectively.

Primary Methods:

1. **Constructor (`new`):** Defines the name and parent for the burst write test class instance.
2. **Build Phase (`build_phase`):**
 - Inherits the build phase from the base test using **`super.build_phase(phase)`**.
 - Creates instances of the burst write sequences for both AHB and APB interfaces.
3. **Run Phase (`run_phase`):**
 - Raises an objection to ensure the simulation doesn't prematurely conclude.
 - Initiates the burst write sequences on their respective sequencers in parallel using the **`fork...join`** construct.
 - Drops the objection after the sequences are finished.
 - Assigns a drain time to handle any remaining transactions before concluding the simulation.

Burst Read Test Class (`ahb_apb_burst_read_test`)

The **`ahb_apb_burst_read_test`** class, an extension of **`ahb_apb_base_test`**, focuses on verifying the behavior of the DUT during burst read operations on the AHB and APB interfaces.

Core Components:

1. **Burst Read Sequences (`ahb_seq_h` and `apb_seq_h`):** These handles correspond to sequences that spawn burst read traffic on the AHB and APB interfaces, respectively.

Primary Methods:

1. **Constructor (new):** Establishes the name and parent for the burst read test class instance.
2. **Build Phase (build_phase):**
 - Inherits the base test's build phase by invoking **super.build_phase(phase)**.
 - Instantiates the burst read sequences for both AHB and APB interfaces.
3. **Run Phase (run_phase):**
 - Raises an objection to keep the simulation active.
 - Starts the burst read sequences on their respective sequencers simultaneously using the **fork...join** structure.
 - Drops the objection post the sequences' completion.
 - Sets a drain time to process any residual transactions before the simulation concludes.

Without being as thorough or detailed as the components above, here is a brief description of the remaining components and pieces of our UVM-Based VIP.

UVM Imports and Macros:

- The Universal Verification Methodology (UVM) package is imported to utilize its standardized features.
- The UVM macros are included to aid in UVM component registration and message reporting.

Global Settings:

- **N_TX:** Defines the total number of transactions to be generated. This value can be adjusted as needed.
- A provision is made to generate a random number of transactions for burst sequences.

Sequence Items and Configuration:

- **ahb_sequence_item.sv** and **apb_sequence_item.sv:** Define the data items or transactions that flow through the AHB and APB interfaces respectively.
- **ahb_apb_env_config.sv:** Describes the configuration settings for the environment, like which agents are active and the type of tests to run.

AHB Components:

- Components related to the AHB protocol are included:
 - **ahb_sequencer.sv:** Manages the order of executing sequences for AHB.

- **ahb_driver.sv**: Drives signals onto the AHB interface based on the sequence item.
- **ahb_monitor.sv**: Observes the AHB interface and reports transactions.
- **ahb_agent.sv**: A container that encapsulates the AHB sequencer, driver, and monitor.

APB Components:

- Components related to the APB protocol are included:
 - **apb_sequencer.sv**: Manages the order of executing sequences for APB.
 - **apb_driver.sv**: Drives signals onto the APB interface based on the sequence item.
 - **apb_monitor.sv**: Observes the APB interface and reports transactions.
 - **apb_agent.sv**: A container that encapsulates the APB sequencer, driver, and monitor.

Assertions:

Assertions in AHB and APB Sequences

Assertions play a pivotal role in the verification process by enabling the validation of specific behaviors or characteristics of a design. In the context of the AHB and APB sequences, assertions are primarily employed to ensure that specific attributes of the sequence items are appropriately randomized based on the desired criteria.


```

74         req.HRESETn == 1'b0;
75         req.HWRITE == 1'b1;
76         req.HTRANS == 2'b00;
77     });
78     finish_item(req);
79
80     // Generate another write sequence with different constraints.
81     req = ahb_sequence_item::type_id::create("req");
82     start_item(req);
83     assert(req.randomize() with {
84         req.HRESETn == 1'b1;
85         req.HWRITE == 1'b1;
86         req.HSELAHB == 1'b1;
87         req.HTRANS == 2'b10;
88     });
89     finish_item(req);
90
91     // Generate a third write sequence with further constraints.
92     req = ahb_sequence_item::type_id::create("req");
93     start_item(req);
94     assert(req.randomize() with {
95         req.HRESETn == 1'b1;
96         req.HSELAHB == 1'b1;
97         req.HWRITE == 1'b1;
98         req.HTRANS == 2'b00;
99     });
100    finish_item(req);
101    endtask
102    endclass

```

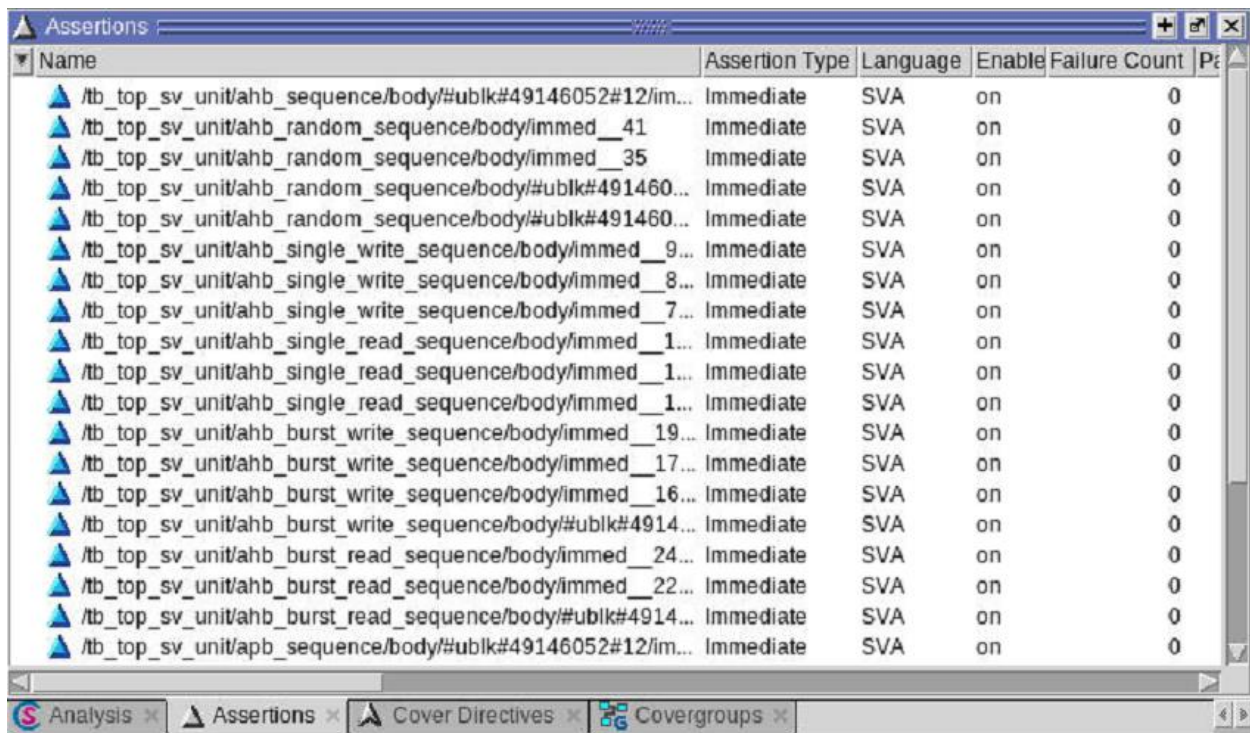
1. AHB Sequences:

- **ahb_sequence:** This basic sequence class generates random AHB transactions. The assertion ensures that the request object (**req**) is randomized correctly.
- **ahb_random_sequence:** This class is tailored to generate specific AHB transactions based on the **HTRANS** values. The assertions enforce the constraint on the **HTRANS** field for different parts of the sequence.
- **ahb_single_write_sequence:** Focused on generating specific write transactions, this sequence uses assertions to ensure that the AHB transactions meet the desired characteristics, such as asserting the reset condition or specifying the write operation.

- **ahb_single_read_sequence**: Similar to its write counterpart, this sequence aims to generate specific read transactions. The assertions ensure the correct transaction type and conditions.
- **ahb_burst_write_sequence and ahb_burst_read_sequence**: These sequences generate burst write and read AHB transactions, respectively. They utilize assertions to confirm the correct transaction attributes and sequences.

2. APB Sequences:

- **apb_sequence**: This foundational sequence class spawns random APB transactions. An assertion is present to validate the correct randomization of the request object.
- **apb_random_sequence**: Tailored to generate specific APB transactions based on the **PSLVERR** values, assertions are employed to enforce constraints on the **PSLVERR** field.
- **apb_single_write_sequence and apb_single_read_sequence**: These sequences are dedicated to generating specific single write and read transactions. Assertions ascertain that the transactions conform to the desired criteria.
- **apb_burst_write_sequence and apb_burst_read_sequence**: Designed for burst operations, these sequences generate burst write and read transactions, respectively. Assertions are used to ensure the correct behavior and attributes of the transactions.



Name	Assertion Type	Language	Enable	Failure Count	Pass/Fail
/tb_top_sv_unit/ahb_sequence/body/#ublk#49146052#12/im...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_random_sequence/body/immed_41	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_random_sequence/body/immed_35	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_random_sequence/body/#ublk#491460...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_random_sequence/body/#ublk#491460...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_single_write_sequence/body/immed_9...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_single_write_sequence/body/immed_8...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_single_write_sequence/body/immed_7...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_single_read_sequence/body/immed_1...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_single_read_sequence/body/immed_1...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_single_read_sequence/body/immed_1...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_burst_write_sequence/body/immed_19...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_burst_write_sequence/body/immed_17...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_burst_write_sequence/body/immed_16...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_burst_write_sequence/body/#ublk#4914...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_burst_read_sequence/body/immed_24...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_burst_read_sequence/body/immed_22...	Immediate	SVA	on	0	
/tb_top_sv_unit/ahb_burst_read_sequence/body/#ublk#4914...	Immediate	SVA	on	0	
/tb_top_sv_unit/apb_sequence/body/#ublk#49146052#12/im...	Immediate	SVA	on	0	

AHB to ABP FSM assertions

Those assertions were used in our Checkpoint 3, and commented out in our Checkpoint 4 since we already verified them in Checkpoint 3 and they slowed down our debugging efforts while working on checkpoint 4. They can easily be copy-pasted in the DUT to be incorporated into this UVM Verification architecture, however.

```
//For the transition from ST_IDLE to ST_WWAIT or ST_READ:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_IDLE && valid && Hwrite) |>= NEXT_STATE == ST_WWAIT);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_IDLE && valid && ~Hwrite) |>= NEXT_STATE == ST_READ);

//For the transition from ST_WWAIT to ST_WRITE or ST_WRITEP:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WWAIT && ~valid) |>= NEXT_STATE == ST_WRITE);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WWAIT && valid) |>= NEXT_STATE == ST_WRITEP);

//For the transition from ST_READ to ST_RENABLE:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_READ) |>= NEXT_STATE == ST_RENABLE);

//For the transition from ST_WRITE to ST_WENABLE or ST_WENABLEP:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WRITE && ~valid) |>= NEXT_STATE == ST_WENABLE);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WRITE && valid) |>= NEXT_STATE == ST_WENABLEP);

//For the transition from ST_WRITEP to ST_WENABLEP:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WRITEP) |>= NEXT_STATE == ST_WENABLEP);

//For the transition from ST_RENABLE to ST_IDLE, ST_WWAIT, or ST_READ:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_RENABLE && ~valid) |>= NEXT_STATE == ST_IDLE);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_RENABLE && valid && Hwrite) |>= NEXT_STATE == ST_WWAIT);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_RENABLE && valid && ~Hwrite) |>= NEXT_STATE == ST_READ);

//For the transition from ST_WENABLE to ST_IDLE, ST_WWAIT, or ST_READ:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WENABLE && ~valid) |>= NEXT_STATE == ST_IDLE);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WENABLE && valid && Hwrite) |>= NEXT_STATE == ST_WWAIT);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WENABLE && valid && ~Hwrite) |>= NEXT_STATE == ST_READ);

//For the transition from ST_WENABLEP to ST_WRITE, ST_WRITEP, or ST_READ:
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WENABLEP && ~valid && Hwritereg) |>= NEXT_STATE == ST_WRITE);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WENABLEP && valid && Hwritereg) |>= NEXT_STATE == ST_WRITEP);
assert property (@(posedge Hclk) (PRESENT_STATE == ST_WENABLEP && ~Hwritereg) |>= NEXT_STATE == ST_READ);
```

Assertions					
Name	Assertion Type	Language	Enable	Failure Count	
▲ /ahb_apb_top/dut/APBControl/assert_16	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_15	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_14	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_13	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_12	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_11	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_10	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_9	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_8	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_7	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_6	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_5	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_4	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_3	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_2	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_1	Concurrent	SVA	on	0	
▲ /ahb_apb_top/dut/APBControl/assert_0	Concurrent	SVA	on	0	
▲ /top_sv_unit/ahb_apb_scoreboard/data_write/...	Immediate	SVA	on	0	
▲ /top_sv_unit/ahb_apb_scoreboard/data_read/...	Immediate	SVA	on	0	

Coverage

Coverage is an indispensable metric in the realm of verification, acting as a yardstick to measure the extent to which the design has been tested. It provides insights into areas of the design that have been sufficiently verified and those that may require additional attention. With growing design complexities, ensuring comprehensive coverage becomes paramount to guaranteeing that every aspect of the design behaves as expected.

Functional Coverage:

In the context of our UVM-based verification environment, functional coverage is used to measure and ensure that all possible scenarios, states, and transitions of the design have been exercised. The **covergroup** construct in SystemVerilog is employed to define specific coverage points and bins, which represent different values or ranges of values.

From the provided code snippet:

- **Coverpoints:** These are specific points in the design or the verification environment where we want to measure coverage. In our **cov_group**, we have defined multiple coverpoints like **reset**, **bus_write**, **bus_read**, and **trans_type**.
 - **reset** coverpoint captures the coverage of the reset signal (**HRESETn**). Specifically, it is interested in the scenario where the reset signal is '0'.
 - **bus_write** and **bus_read** coverpoints focus on the **HWRITE** signal, capturing scenarios where write and read operations occur respectively.

- **trans_type** coverpoint captures the different transaction types (**HTRANS**) of the AHB protocol, including idle, non-sequential, and sequential transactions.
- **Bins**: For every coverpoint, bins are used to partition the value space into subsets. Each bin captures a specific value or a range of values for the associated coverpoint. For instance, for the **bus_write** coverpoint, the **write_val** bin captures the scenario where **HWRITE** is '1', indicating a write operation.
- **Cross Coverage**: This allows for tracking combinations of multiple coverpoints. In our covergroup, **WRITE_COVERAGE** captures the combinations of write operations (**bus_write**) with different transaction types (**trans_type**). Similarly, **READ_COVERAGE** captures the combinations of read operations with various transaction types.

```
// Covergroup to capture coverage metrics.
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}; }

    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
```

Name	Class Type	Coverage	Goal	% of Goal	Status	Included	Merge_instal
/tb_top_sv_unit/ahb_...		88.88%					
TYPE cov_group		88.88%	100	88.88%	<div><div></div></div>	✓	
CVP cov_grou...		100.00%	100	100.00%	<div><div></div></div>	✓	
CVP cov_grou...		100.00%	100	100.00%	<div><div></div></div>	✓	
CVP cov_grou...		100.00%	100	100.00%	<div><div></div></div>	✓	
CVP cov_grou...		100.00%	100	100.00%	<div><div></div></div>	✓	
CROSS cov_gr...		100.00%	100	100.00%	<div><div></div></div>	✓	
CROSS cov_gr...		33.33%	100	33.33%	<div><div></div></div>	✓	
bin<read_v...		3	1	100.00%	<div><div></div></div>	✓	
bin<read_v...		0	1	0.00%	<div><div></div></div>	✓	
bin<read_v...		0	1	0.00%	<div><div></div></div>	✓	
INST Vtb_top_s...		88.88%	100	88.88%	<div><div></div></div>	✓	

The coverage above is not 100% because it depends on the test we're running. We have 4 different tests, and when they're all ran, the coverage can hit 100%.

Running the Verification Environment

1. The Makefile:

The provided Makefile is a tool to automate the compilation, simulation, and coverage report generation for the verification environment. Here's a breakdown of its functionality:

.PHONY: This line declares certain targets (like `all`, `compile`, etc.) as "phony", which means they don't represent files but rather are just names for a sequence of commands.

all: This is the default target. When you simply use the command `make`, it performs the operations in the sequence: `clean`, `compile`, `run_coverage`, and `report`.

compile: This target uses the `vlog` command to compile the source files `ahb_intf.sv`, `apb_intf.sv`, `DUT.sv`, and `tb_top.sv`.

run_coverage: Initiates the simulation with coverage enabled. The `-cvgperinstance` option calculates coverage for each instance of the design. The coverage save `-onexit covfile.ucdb` command ensures that coverage data is saved upon exit.

run: This target starts the simulation. The various flags and options provided set the timescale, access permissions, test name, verbosity, and coverage specifics.

clean: Removes the work directory (which holds compiled files) and the transcript file to ensure a clean environment for the next simulation run.

report: Generates a coverage report using the `vsim` tool. The report provides a detailed coverage analysis and is saved to the file `ahb_apb_bridge_report.txt`.

2. Flexible Test Configuration:

The design verification process includes multiple tests. In the provided code snippet, there's an initialization block in the UVM testbench that sets up the UVM environment and selects which test to run. While there are four tests in the architecture (`ahb_apb_single_write_test`, `ahb_apb_single_read_test`, `ahb_apb_burst_write_test`, and `ahb_apb_burst_read_test`), only one is uncommented and will be executed in the current configuration.

The beauty of UVM is its flexibility. If you want to run a different test, you can simply uncomment the desired test and comment out the others. This avoids the need to recompile the entire architecture. Alternatively, you can use UVM command-line syntax to specify or add new tests during simulation. The line

+UVM_TESTNAME=ahb_apb_bridge_burst_read_test in the Makefile's run target is an example of this. By changing the test name after +UVM_TESTNAME=, you can dynamically select a different test without altering the testbench code.

```
##### ECE 593 Pre-Si Validation Summer 2023 #####
.PHONY: all compile run_coverage run clean report
# Default target
all: clean compile run_coverage report

compile:
    vlog ahb_intf.sv apb_intf.sv DUT.sv tb_top.sv

run_coverage:
    vsim -cvgperinstance -c Bridge_Top tb_top -do "coverage save -onexit covfile.ucdb; run -all; exit"

run:
    vsim -c top_tb -do "vsim -Q -timescale 1ns/1ns -access +rw
+UVM_TESTNAME=ahb_apb_bridge_burst_read_test +UVM_VERBOSITY=UVM_NONE -coverage all -covfile covfile.ccf -
covdut ahb2apb -uvmnocdnsextra work.top; run -all; quit"

clean:
    rm -rf work
    rm -rf transcript

report:
    vsim -cvgperinstance -viewcov covfile.ucdb -do "coverage report -file ahb_apb_bridge_report.txt -
byfile -detail -noannotate -option -cvg"
```

Results and Discussion:

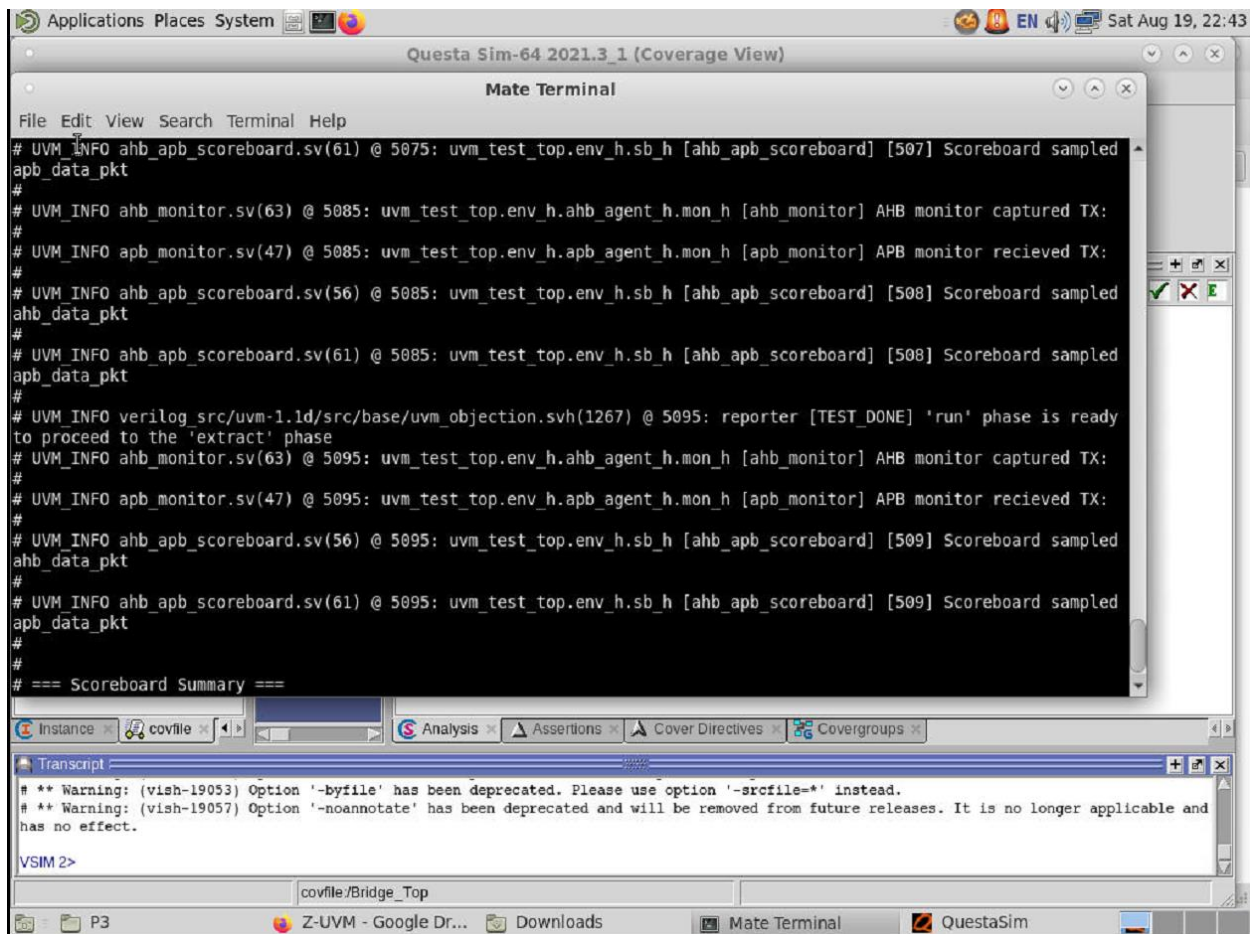
Debugging the Transaction Print Method:

One of the primary challenges faced during the verification process was the visualization and tracking of transaction information. We initially attempted to use the **print()** method to display transaction details. However, this approach led to several errors. As an alternative, we tried the **convert2string** method. Despite it being error-free, it didn't provide the desired output. Instead of giving the transaction's details, it only returned a generic string message.

Upon a more in-depth investigation using the UVM manual and various online resources, it became evident that two potential solutions could address the issue:

1. Customizing the settings of the **print()** method.
2. Utilizing the **sprint()** method.

After some experimentation, the **sprint()** method emerged as the more effective solution. It not only avoided the errors associated with **print()** but also provided a comprehensive string representation of the transaction. This representation can be easily displayed, logged, or further processed for analysis.



Enhancing Transaction Visibility:

With the implementation of **sprint()**, combined with the **uvm_info** macro across the testbench components, we achieved a significant enhancement in transaction visibility. This combination allowed us to:

- Display which packet/transaction was generated.
- Monitor the packet's output from the Device Under Test (DUT).
- Track the packet's progression through the verification environment.

This approach provided a dynamic way to observe and analyze transaction behavior in real-time. It also ensured that any irregularities or unexpected behaviors could be promptly identified and addressed.

Scoreboard Insights:

The scoreboard, an integral component of our verification environment, benefited significantly from the improved transaction visibility. With the data relayed using **sprint()** and **uvm_info**, the scoreboard could better monitor and validate the flow of data between the AHB and APB interfaces.

Key functionalities of the scoreboard include:

- **Monitoring Packets:** The scoreboard utilizes FIFOs (First-In-First-Out buffers) to hold and process packets from both the AHB and APB monitors.
- **Counters:** The scoreboard employs counters like **ahb_pkt_count** and **apb_pkt_count** to keep track of the number of processed packets. Additionally, **verified_data_count** provides insights into how many of these packets have been validated successfully.
- **Predictive Analysis:** A crucial feature of the scoreboard is its ability to predict expected data based on the packets sampled from the monitors. It then compares these predictions with the actual data to ascertain the correctness of the DUT's behavior.
- **Coverage and Reporting:** The scoreboard also plays a pivotal role in coverage analysis. Through the defined covergroup, **cov_group**, it captures essential coverage metrics, ensuring that the design is thoroughly tested. At the simulation's conclusion, a comprehensive report detailing packet counts, verification statistics, and coverage metrics is displayed, providing a holistic view of the verification process.

In summary, the incorporation of **sprint()** and **uvm_info** into the verification process has immensely improved the clarity and efficiency of our verification efforts. The enhanced visibility, combined with the scoreboard's robust capabilities, ensures that our design is rigorously tested and validated, paving the way for a reliable and robust final product.

```
#
# === Scoreboard Summary ===
# AHB Packets: 509
# APB Packets: 509
# Verified Transactions:      504
# Unverified Transactions:    5
# === Coverage Report ===
# RESET: 100.000000%
# WRITE: 100.000000%
# READ: 100.000000%
# === End of Summary ===
#
```

```

Mate Terminal
File Edit View Search Terminal Help
# PREADY integral 1 255
# -----
# UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1267) @ 5095: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# UVM_INFO ahb_monitor.sv(63) @ 5095: uvm_test_top.env_h.ahb_agent_h.mon_h [ahb_monitor] AHB monitor captured TX:
# -----
# Name      Type      Size  Value
# -----
# mon2sb    ahb_sequence_item -    @9090
# RESETn    integral    1    1
# HADDR     integral    32    'h49e
# HTRANS    integral    2    'd0
# HWRITE    integral    1    1
# HWDATA    integral    32    'hab4357e8
# HSELAHB   integral    1    1
# HREADY    integral    1    1
# -----
# UVM_INFO apb_monitor.sv(47) @ 5095: uvm_test_top.env_h.apb_agent_h.mon_h [apb_monitor] APB monitor recieved TX:
# -----
# Name      Type      Size  Value
# -----
# mon2sb    apb_sequence_item -    @9094
# PSLVERR   integral    1    'd0
# PREADY    integral    1    255
# -----

```

Here's a snippet of the terminal output:

```

# UVM_INFO ahb_driver.sv(92) @ 25: uvm_test_top.env_h.ahb_agent_h.drv_h [ahb_driver]
AHB driver Delivered Tx:

```

```

# -----
# Name      Type      Size  Value
# -----
# req       ahb_sequence_item -    @952
# begin_time time      64    0
# depth     int       32    'd2
# parent sequence (name) string    9    ahb_seq_h
# parent sequence (full name) string    52
uvm_test_top.env_h.ahb_agent_h.sequencer_h.ahb_seq_h
# sequencer string    42    uvm_test_top.env_h.ahb_agent_h.sequencer_h
# RESETn    integral    1    'd0
# HADDR     integral    32    'h2b5
# HTRANS    integral    2    'd0

```

```

# HWRITE          integral      1  1
# HWDATA          integral     32  'h1b1351e1
# HSELAHB         integral      1  1
# HREADY          integral      1  1
# -----
#
# UVM_INFO ahb_sequence_item.sv(49) @ 25:
# uvm_test_top.env_h.ahb_agent_h.sequencer_h@@ahb_seq_h.req [AHB_SEQUENCE_ITEM]
# Transaction [2]: -----
# -----
# Name              Type          Size Value
# -----
# req               ahb_sequence_item - @1038
# begin_time        time          64  25
# depth             int           32  'd2
# parent sequence (name) string      9  ahb_seq_h
# parent sequence (full name) string  52
# uvm_test_top.env_h.ahb_agent_h.sequencer_h.ahb_seq_h
# sequencer         string        42  uvm_test_top.env_h.ahb_agent_h.sequencer_h
# RESETn            integral       1  1
# HADDR             integral     32  'h403
# HTRANS            integral       2  2
# HWRITE            integral       1  1
# HWDATA            integral     32  'ha9ceb36a
# HSELAHB           integral       1  1
# HREADY            integral       1  'd0
# -----
#
# UVM_INFO ahb_driver.sv(92) @ 35: uvm_test_top.env_h.ahb_agent_h.drv_h [ahb_driver]
# AHB driver Delivered Tx:

```

```

# -----
# Name          Type          Size Value
# -----
# req           ahb_sequence_item - @1038
# begin_time    time          64 25
# depth         int           32 'd2
# parent sequence (name) string      9 ahb_seq_h
# parent sequence (full name) string  52
uvmm_test_top.env_h.ahb_agent_h.sequencer_h.ahb_seq_h
# sequencer     string        42 uvmm_test_top.env_h.ahb_agent_h.sequencer_h
# RESETn        integral      1 1
# HADDR         integral      32 'h403
# HTRANS        integral      2 2
# HWRITE        integral      1 1
# HWDATA        integral      32 'ha9ceb36a
# HSELAHB       integral      1 1
# HREADY        integral      1 1
# -----
#
# UVM_INFO ahb_monitor.sv(63) @ 35: uvmm_test_top.env_h.ahb_agent_h.mon_h
[ahb_monitor] AHB monitor captured TX:
# -----
# Name  Type          Size Value
# -----
# mon2sb ahb_sequence_item - @1042
# RESETn integral      1 'd0
# HADDR  integral      32 'h0
# HTRANS integral      2 'd0
# HWRITE integral      1 'd0

```



```

# HWDATA integral      32 'h0
# HSELAHB integral     1  'd0
# HREADY integral      1  1
# -----
#
# UVM_INFO apb_driver.sv(52) @ 35: uvm_test_top.env_h.apb_agent_h.drv_h [apb_driver]
APB driver delivered Tx:
# -----
# Name                Type      Size Value
# -----
# req                  apb_sequence_item -  @1033
# begin_time           time       64  25
# depth                int        32  'd2
# parent sequence (name) string     9  apb_seq_h
# parent sequence (full name) string  52
uvm_test_top.env_h.apb_agent_h.sequencer_h.apb_seq_h
# sequencer            string     42  uvm_test_top.env_h.apb_agent_h.sequencer_h
# PSLVERR              integral    1  'd0
# PREADY               integral    1  255
# -----
#
# UVM_INFO apb_monitor.sv(47) @ 35: uvm_test_top.env_h.apb_agent_h.mon_h
[apb_monitor] APB monitor recieved TX:
# -----
# Name    Type      Size Value
# -----
# mon2sb   apb_sequence_item -  @1046
# PSLVERR  integral    1  'd0
# PREADY   integral    1  255

```

```

# -----
#
# UVM_INFO ahb_apb_scoreboard.sv(56) @ 35: uvm_test_top.env_h.sb_h
[ahb_apb_scoreboard] [3] Scoreboard sampled ahb_data_pkt
# -----
# Name      Type      Size Value
# -----
# mon2sb    ahb_sequence_item - @1042
# RESETn    integral    1  'd0
# HADDR     integral    32  'h0
# HTRANS     integral    2  'd0
# HWRITE     integral    1  'd0
# HWDATA     integral    32  'h0
# HSELAHB    integral    1  'd0
# HREADY     integral    1  1
# -----
#
# UVM_INFO ahb_apb_scoreboard.sv(61) @ 35: uvm_test_top.env_h.sb_h
[ahb_apb_scoreboard] [3] Scoreboard sampled apb_data_pkt
# -----
# Name      Type      Size Value
# -----
# mon2sb    apb_sequence_item - @1046
# PSLVERR    integral    1  'd0
# PREADY     integral    1  255
# -----
#
# UVM_INFO ahb_sequence_item.sv(49) @ 35:
uvm_test_top.env_h.ahb_agent_h.sequencer_h@@ahb_seq_h.req [AHB_SEQUENCE_ITEM]

```

Transaction [3]: -----

# Name	Type	Size	Value
# req	ahb_sequence_item -	@1050	
# begin_time	time	64	35
# depth	int	32	'd2
# parent sequence (name)	string	9	ahb_seq_h
# parent sequence (full name)	string	52	uvm_test_top.env_h.ahb_agent_h.sequencer_h.ahb_seq_h
# sequencer	string	42	uvm_test_top.env_h.ahb_agent_h.sequencer_h
# RESETn	integral	1	1
# HADDR	integral	32	'h169
# HTRANS	integral	2	3
# HWRITE	integral	1	1
# HWDATA	integral	32	'hccc6033a
# HSELAHB	integral	1	1
# HREADY	integral	1	'd0

UVM_INFO ahb_driver.sv(92) @ 45: uvm_test_top.env_h.ahb_agent_h.drv_h [ahb_driver]
AHB driver Delivered Tx:

# Name	Type	Size	Value
# req	ahb_sequence_item -	@1050	
# begin_time	time	64	35
# depth	int	32	'd2
# parent sequence (name)	string	9	ahb_seq_h

```

# parent sequence (full name) string      52
uvm_test_top.env_h.ahb_agent_h.sequencer_h.ahb_seq_h

# sequencer      string      42  uvm_test_top.env_h.ahb_agent_h.sequencer_h
# RESETn         integral     1   1
# HADDR          integral     32  'h169
# HTRANS         integral     2   3
# HWRITE         integral     1   1
# HWDATA         integral     32  'hccc6033a
# HSELAHB        integral     1   1
# HREADY         integral     1   1

# -----
#

# UVM_INFO ahb_monitor.sv(63) @ 45: uvm_test_top.env_h.ahb_agent_h.mon_h
[ahb_monitor] AHB monitor captured TX:

# -----

# Name      Type      Size Value
# -----

# mon2sb    ahb_sequence_item -  @1058
# RESETn    integral     1  'd0
# HADDR     integral     32  'h2b5
# HTRANS    integral     2  'd0
# HWRITE    integral     1   1
# HWDATA    integral     32  'h0
# HSELAHB   integral     1   1
# HREADY    integral     1   1

# -----
#

# UVM_INFO apb_driver.sv(52) @ 45: uvm_test_top.env_h.apb_agent_h.drv_h [apb_driver]
APB driver delivered Tx:

```

```

# -----
# Name          Type          Size Value
# -----
# req           apb_sequence_item - @1054
# begin_time    time          64 35
# depth         int           32 'd2
# parent sequence (name) string      9 apb_seq_h
# parent sequence (full name) string  52
uvm_test_top.env_h.apb_agent_h.sequencer_h.apb_seq_h
# sequencer     string        42 uvm_test_top.env_h.apb_agent_h.sequencer_h
# PSLVERR       integral      1  'd0
# PREADY        integral      1  255
# -----
#
# UVM_INFO apb_monitor.sv(47) @ 45: uvm_test_top.env_h.apb_agent_h.mon_h
[apb_monitor] APB monitor recieved TX:
# -----
# Name    Type          Size Value
# -----
# mon2sb  apb_sequence_item - @1062
# PSLVERR integral      1  'd0
# PREADY  integral      1  255
# -----
#
# UVM_INFO ahb_apb_scoreboard.sv(56) @ 45: uvm_test_top.env_h.sb_h
[ahb_apb_scoreboard] [4] Scoreboard sampled ahb_data_pkt
# -----
# Name    Type          Size Value
# -----

```

```

# mon2sb  ahb_sequence_item -  @1058
# RESETn  integral      1  'd0
# HADDR   integral     32  'h2b5
# HTRANS  integral      2  'd0
# HWRITE  integral      1  1
# HWDATA  integral     32  'h0
# HSELAHB integral      1  1
# HREADY  integral      1  1
# -----
#
# UVM_INFO ahb_apb_scoreboard.sv(61) @ 45: uvm_test_top.env_h.sb_h
[ahb_apb_scoreboard] [4] Scoreboard sampled apb_data_pkt
# -----
# Name      Type      Size Value
# -----
# mon2sb  apb_sequence_item -  @1062
# PSLVERR integral      1  'd0
# PREADY  integral      1  255
# -----
#
# UVM_INFO ahb_sequence_item.sv(49) @ 45:
uvm_test_top.env_h.ahb_agent_h.sequencer_h@@ahb_seq_h.req [AHB_SEQUENCE_ITEM]
Transaction [4]: -----
-----
# Name      Type      Size Value
# -----
# req      ahb_sequence_item -  @1066
# begin_time  time      64  45
# depth     int       32  'd2

```



```

# parent sequence (name)    string      9  ahb_seq_h
# parent sequence (full name) string     52
uvm_test_top.env_h.ahb_agent_h.sequencer_h.ahb_seq_h
# sequencer                 string     42  uvm_test_top.env_h.ahb_agent_h.sequencer_h
# RESETn                   integral    1  1
# HADDR                    integral    32  'h305
# HTRANS                   integral     2  3
# HWRITE                   integral     1  1
# HWDATA                   integral    32  'h581ddca6
# HSELAHB                  integral     1  1
# HREADY                   integral     1  'd0
# -----
#
# UVM_INFO ahb_monitor.sv(63) @ 55: uvm_test_top.env_h.ahb_agent_h.mon_h
[ahb_monitor] AHB monitor captured TX:
# -----
# Name      Type      Size Value
# -----
# mon2sb    ahb_sequence_item -  @1074
# RESETn    integral    1  1
# HADDR     integral    32  'h403
# HTRANS    integral     2  2
# HWRITE    integral     1  1
# HWDATA    integral    32  'h0
# HSELAHB   integral     1  1
# HREADY    integral     1  'd0
# -----
#

```

UVM_INFO apb_driver.sv(52) @ 55: uvm_test_top.env_h.apb_agent_h.drv_h [apb_driver]
APB driver delivered Tx:

```
# -----  
# Name          Type          Size Value  
# -----  
# req           apb_sequence_item - @1070  
# begin_time    time          64 45  
# depth         int           32 'd2  
# parent sequence (name) string      9 apb_seq_h  
# parent sequence (full name) string  52  
uvm_test_top.env_h.apb_agent_h.sequencer_h.apb_seq_h  
# sequencer     string        42 uvm_test_top.env_h.apb_agent_h.sequencer_h  
# PSLVERR       integral      1  'd0  
# PREADY        integral      1  255  
# -----  
#
```

UVM_INFO apb_monitor.sv(47) @ 55: uvm_test_top.env_h.apb_agent_h.mon_h
[apb_monitor] APB monitor recieved TX:

```
# -----  
# Name  Type          Size Value  
# -----  
# mon2sb apb_sequence_item - @1078  
# PSLVERR integral      1  'd0  
# PREADY  integral      1  255  
# -----  
#
```

UVM_INFO ahb_apb_scoreboard.sv(56) @ 55: uvm_test_top.env_h.sb_h
[ahb_apb_scoreboard] [5] Scoreboard sampled ahb_data_pkt

```
# -----
```

```

# Name      Type      Size Value
# -----
# mon2sb    ahb_sequence_item - @1074
# RESETn    integral    1    1
# HADDR     integral    32    'h403
# HTRANS     integral    2    2
# HWRITE     integral    1    1
# HWDATA     integral    32    'h0
# HSELAHB    integral    1    1
# HREADY     integral    1    'd0
# -----
#
# UVM_INFO ahb_apb_scoreboard.sv(61) @ 55: uvm_test_top.env_h.sb_h
[ahb_apb_scoreboard] [5] Scoreboard sampled apb_data_pkt
# -----
# Name      Type      Size Value
# -----
# mon2sb    apb_sequence_item - @1078
# PSLVERR    integral    1    'd0
# PREADY     integral    1    255
# -----

```

Debugging

The Journey from SystemVerilog to UVM:

Transitioning from a SystemVerilog testbench (from Checkpoint 3) to the Universal Verification Methodology (UVM) proved to be a particularly arduous task. My initial approach was to directly

translate the SV TB to UVM. However, the complexities arose when I tried to incorporate essential UVM features like analysis ports, configuration databases (**config_db**), and others into the existing code. Editing an already established codebase to fit the UVM paradigm was like trying to fit a square peg into a round hole.

Instead of continuing down this seemingly endless rabbit hole, I took a step back. The decision to start afresh was pivotal. I began by outlining the architecture of the testbench, abstracting the communication between its different components, and mapping out how each part would interact within the UVM framework.

Challenges Encountered:

The transition was not without its challenges. Apart from the aforementioned **convert2string** method anomaly, I encountered a plethora of issues, including:

- **Compilation Errors:** These were common and often resulted from syntax mistakes, undeclared variables, or incorrect module instantiations.
- **Simulation Errors:** These errors cropped up during the simulation runs, often indicating mismatches between expected and observed behaviors, or due to incorrect sequence operations.
- **UVM Usage Errors:** As a sophisticated methodology, UVM has its intricacies. Mistakes like not setting up the configuration database properly, incorrectly connecting TLM ports, or misusing UVM macros were frequent. For example, an analysis port might have been connected to a wrong export or the configuration data might not have been correctly retrieved from **config_db**.

Each error became a learning opportunity. Through rigorous debugging sessions, hours of scrutinizing the UVM manual, and leveraging online communities for insights, I was able to systematically address and rectify each issue.

Contrasting the Old and New Designs:

Attached are two designs: the incomplete testbench translated from Checkpoint 3, and the new functional design.

The initial design, though a direct translation, lacked the elegance, flexibility, and power of UVM. It was more rigid and didn't leverage UVM's capabilities fully.

On the other hand, the new design is a testament to UVM's robustness. It fully harnesses UVM's potent randomization techniques. Instead of relying on a deterministic sequence of tests, the new design adopts a constraint-random approach. By using UVM, we can generate a plethora of random transactions, ensuring a more comprehensive verification. This approach, combined with structured test scenarios, offers a more in-depth and randomized testing strategy than what was available in Checkpoint 3.

Summary

This report goes into the development and debugging of a UVM-based verification environment, focusing on two main protocols: AHB and APB. Various classes were designed to generate sequences and monitor transactions for both protocols, with the sequences ranging from basic random transactions to specific constrained ones. Throughout the development process, the power of assertions was harnessed to ensure the validity of transactions.

An in-depth discussion was presented on the Makefile used to run the verification environment, shedding light on its structure and functionality. This Makefile not only ensures a systematic flow from compilation to reporting but also offers flexibility to the user in terms of the tests they wish to run.

The results and discussion section brought to the forefront the challenges faced during the transition from a traditional SystemVerilog testbench to a UVM-based one. Various debugging methodologies and solutions were discussed, highlighting the iterative nature of the process and the learnings drawn from each challenge.

Conclusion

The journey of developing a UVM-based verification environment, as detailed in this report, underscores the complexity and depth of modern verification processes. While the transition from conventional verification methodologies to UVM can be daunting, the benefits, in terms of flexibility, reusability, and comprehensiveness, are undeniable.

This report serves as a testament to the iterative nature of design and debugging in the verification domain. Each challenge encountered was not a setback but rather an opportunity to delve deeper into the intricacies of UVM and refine the verification environment.

In the ever-evolving landscape of digital systems, where complexity and scale continue to grow, methodologies like UVM stand as pivotal tools ensuring that our digital systems function as intended. The experiences and insights drawn from this project will undoubtedly serve as invaluable assets in future endeavors in the domain of hardware verification.