



**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 3 SV VERIFICATION IP**

**MOHAMED GHONIM**

**GAYATRI VEMURI**

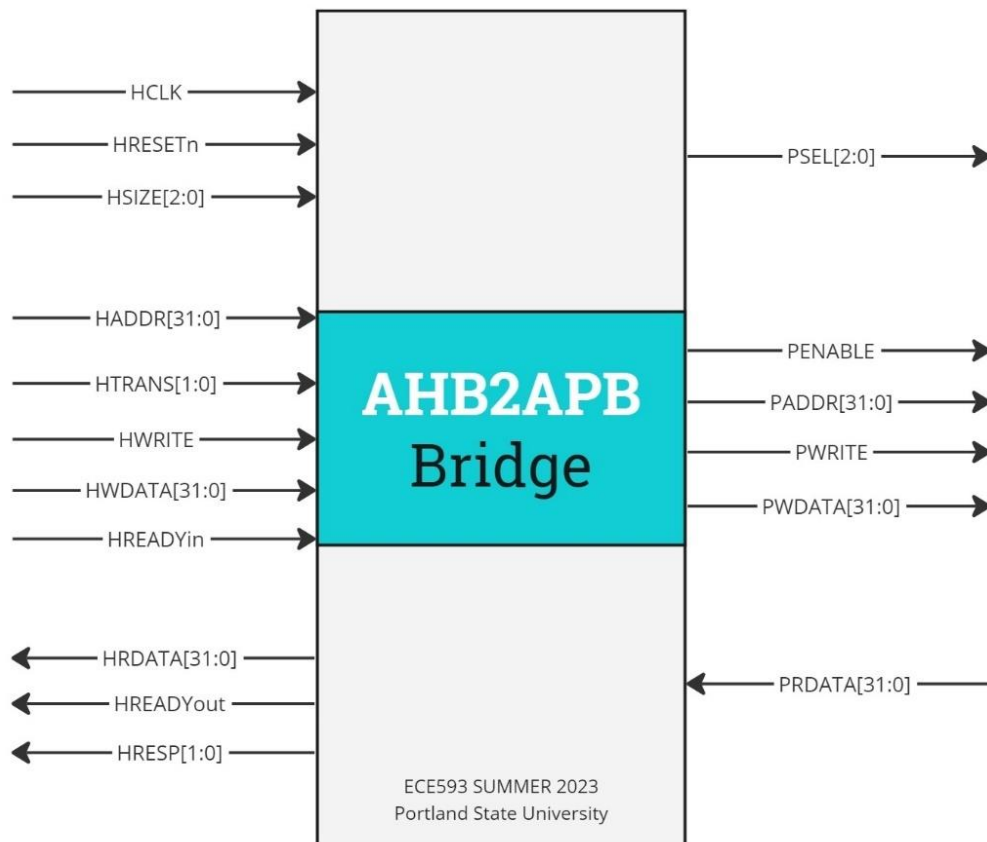
**08/03/2023**

## Table of Contents

Introduction .....	2
Verification IP Structure .....	3
Verification Methodology: .....	4
Verification Components: .....	5
Top class: .....	5
Interface (BFM):.....	6
Transactions:.....	6
Coverage:.....	7
Assertions:.....	9
AHB to ABP FSM assertions .....	9
Environment: .....	10
Generator: .....	11
Driver:.....	11
Monitor: .....	11
Scoreboard: .....	12
Test: .....	13
Results:.....	15
Debugging .....	17
Summary .....	18
Conclusion.....	19

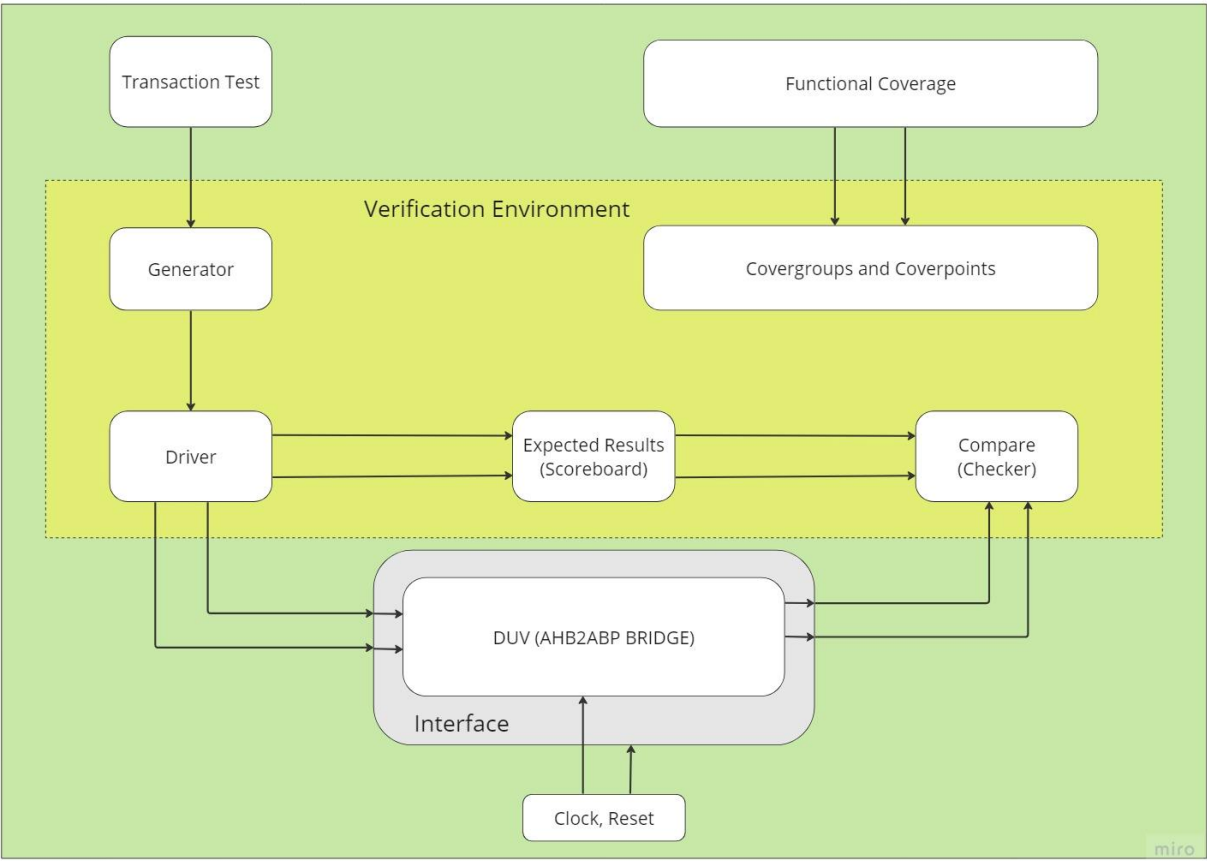
## Introduction

This report documents the design, development, and usage of a verification environment using SystemVerilog for the AHB-APB Bridge verification IP (VIP). This VIP facilitates a robust and comprehensive verification strategy ensuring high-quality validation of an AHB-APB Bridge design. The verification environment, constructed using a rigorous methodology, encompasses several classes and components which coherently simulate, monitor, and analyze various scenarios in the target design. These components are assembled based on an object-oriented programming (OOP) paradigm and include a transaction model, generator, driver, monitor, scoreboard, and coverage collector, each serving a unique purpose in the overall verification flow. Furthermore, the verification environment is founded on the concept of a testbench architecture, encapsulating all the vital aspects of a typical verification suite including generation of stimuli, application of the stimuli to the design under verification (DUV), monitoring of outputs, checking results for correctness, and measuring the effectiveness of the tests through functional coverage.



# Verification IP Structure

Top



## Verification Methodology:

---

In the AHB APB Bridge Verification project, we've employed a modular, bottom-up verification methodology that aligns with the best practices of system verification.

Our approach started with the smallest and simplest components of the system, verifying the function of individual modules before integrating them and testing the larger subsystems. This is known as "bottom-up" testing. This technique allowed us to isolate problems early, minimize their impact, and fix them quickly before moving to more complex scenarios.

We started by verifying the generator (transaction generator), which is the component responsible for creating and driving transactions into the design under test (DUT). The generator was tested with a limited number of scenarios first to ensure the basic functionality.

After the generator was tested and validated, we gradually increased the complexity of the test scenarios, adding more variables and conditions. This helped in exploring the operational boundaries and different state spaces of the AHB APB Bridge.

After that, we integrated the monitor and the scoreboard into our test environment. The monitor, a passive component, was used to listen to and record transactions occurring in the DUT. This recorded data was then sent to the scoreboard, which is essentially the "truth model" that we compared our test outcomes against.

We started with a simple check of the write and read operations and then gradually increased the complexity and the range of our tests. As more test cases were added, the robustness of the DUT against different operational scenarios and edge cases was extensively checked.

This methodical, step-by-step approach ensured a comprehensive coverage of all potential scenarios that the AHB APB Bridge might encounter in a real-world setting. This robust methodology was paramount in confirming that our design meets the desired specifications and is free from any functional errors. It allowed us to gain confidence in the functionality and reliability of the DUT, ensuring a robust and reliable design ready for production.

## Verification Components:

---

### Top class:

The top-level module, **ahb\_apb\_top**, acts as the backbone of this AHB to APB bridge verification process. It unifies all the classes and modules that make up the testbench, and links them with the Device Under Verification (DUT), which is the AHB to APB bridge module named **Bridge\_Top**.

Upon initiation, the **ahb\_apb\_top** module generates a clock signal 'clk' that oscillates with a half-period of 5 ns, thus creating a clock cycle of 10 ns, simulating a real-time clocking system. This clock signal is essential for driving the synchronous behavior of the DUT. Alongside the clock, a reset signal 'reset' is initialized at 0, then set to 1 after 10 time units, essentially creating an asynchronous reset mechanism. These signals are connected to the DUT and the interface **bfm**.

The top module includes ten crucial files which each define a different aspect of the verification environment. These files cover a wide range of classes from the transaction model (**transactions.sv**), to the coverage collector (**coverage.sv**), and to the actual test (**test.sv**).

The DUT is instantiated as 'dut' and connected to the verification environment using the bus functional model (BFM) interface named 'bfm'. This BFM interface is the conduit for signals moving between the DUT and the verification environment.

The Test class is instantiated as 'test\_h', initializing the test sequence of the verification process. The 'run' method of this Test class is then called, signaling the beginning of the verification process. During this process, transactions are generated, dispatched, and monitored while the DUT's responses are observed and validated.

Once the verification process has been completed, the simulation is halted after 100000 time units using the '\$stop' system task. It is important to note that the 'run' method includes coverage sampling to ensure all crucial aspects of the DUT's function have been tested.

In essence, the **ahb\_apb\_top** module ties the entire verification environment together, orchestrating the interactions between the DUT and the verification classes, and controlling the sequence of the verification process.

### Interface (BFM):

The `ahb_apb_bfm_if` interface encapsulates the signal set for the AHB to APB bridge protocol in this SystemVerilog testbench environment, facilitating an organized approach to managing these signals. This interface is critical as it helps establish communication channels between various verification components such as the driver, monitor, and the DUT.

This module specifically defines two clocking blocks - '`drv_cb`' and '`mon_cb`', for the driver and monitor respectively. Clocking blocks are vital in a verification environment as they enable precise control over the timing of signal driving and sampling, contributing to the accuracy of the verification process.

The '`drv_cb`' clocking block is synchronized with the clock and is utilized by the driver to drive signals towards the DUT. Conversely, the '`mon_cb`' clocking block is used by the monitor to sample signals from the DUT, ensuring that both the driver and monitor function in harmony with the clock.

In addition to the clocking blocks, two modports '`master`' and '`slave`' are also defined. These provide structured access to the interface for the driver and monitor respectively. The '`master`' modport represents the driver's perspective, which drives the signals, while the '`slave`' modport symbolizes the monitor's view, which samples the signals.

The interface includes a comprehensive set of signals corresponding to the AHB-APB Bridge protocol, including write signals, read signals, address signals, transfer type encoding, and response signals for both AHB and APB. Each signal is defined as either logic or wire based on their usage within the protocol.

In essence, the `ahb_apb_bfm_if` interface serves as a crucial interconnection point in the verification process, managing the signals and ensuring the synchronization and structured access to these signals for the driver and monitor components.

### Transactions:

The **Transaction** class is a SystemVerilog class that represents a transaction in the AHB-APB bridge. It carries all the necessary information such as the address, data, transaction type, and other necessary properties that are specific to the AHB and APB protocols.

A transaction type is defined as an enumerated type **trans\_type\_e** that can either be an AHB\_READ or AHB\_WRITE. All of the transaction attributes like `Haddr`, `Hwdata`, `Hwrite`, `Htrans`, `Hsize`, `Hburst`, `Paddr`, `Pwdata`, `Pwrite`, `Pselx`, `hresp`, and `hreset` are defined as constrained-random variables (**randc**). This helps in creating randomized but unique transactions during simulation.

Several constraints are set on these attributes to ensure valid transaction generation, like `Haddr`, `Hsize`, and `Hburst`. For instance, the constraint on `Haddr` ensures that the upper 20



bits of Haddr are zero. Constraints on Hsize and Hburst limit their possible values to 0, 1, and 2.

A covergroup **cov\_cg** is defined to measure coverage. It uses coverpoints on Hwrite, Htrans, Hsize, and Hburst, defining bins to classify different possibilities of these parameters. Cross coverage is also defined to measure the intersection of multiple coverpoints, providing a comprehensive understanding of how different signal combinations are covered.

The class constructor (**new()**) initializes the covergroup. A method **update\_trans\_type()** is defined to update the transaction type based on the Hwrite signal, and the covergroup samples are updated each time this method is called.

A method **print\_transaction()** is provided to print the details of the transaction, which is useful for debugging and logging the activity during the simulation.

This **Transaction** class provides an abstract representation of an AHB-APB transaction, enabling complex test scenarios with random yet constrained values for each transaction attribute, and facilitates comprehensive coverage measurement.

#### Coverage:

Coverage is an essential aspect of the verification process. It is used to measure the effectiveness of the tests and to ascertain that all aspects of the design have been thoroughly checked. In this particular **Transaction** class, a covergroup **cov\_cg** is defined for this purpose. It has coverpoints for signals like Hwrite, Htrans, Hsize, and Hburst, with each coverpoint having specific bins to account for different possible values. This arrangement provides a detailed and categorized insight into how many and which specific scenarios were hit during the simulations. The class also defines cross-coverage among these coverpoints to examine the intersection of different scenarios. The coverage data is updated by calling **cov\_cg.sample()** in the **update\_trans\_type** method. This comprehensive coverage methodology ensures the transaction class, and thereby, the entire AHB APB Bridge design is thoroughly verified, improving the reliability of the final design.

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/top_sv_unit/Transacti...		100.00%				
TYPE cov_cg		100.00%	100	100.00%		✓
CVP cov_cg::H...		100.00%	100	100.00%		✓
CVP cov_cg::H...		100.00%	100	100.00%		✓
CVP cov_cg::H...		100.00%	100	100.00%		✓
CVP cov_cg::H...		100.00%	100	100.00%		✓
CROSS cov_cg::C...		100.00%	100	100.00%		✓
CROSS cov_cg::C...		100.00%	100	100.00%		✓
CROSS cov_cg::C...		100.00%	100	100.00%		✓





top_sv_unit/Transacti...	100.00%				
TYPE cov_cg	100.00%	100	100.00%	<div></div>	✓
CVP cov_cg::H...	100.00%	100	100.00%	<div></div>	✓
bin read	801	1	100.00%	<div></div>	✓
bin write	900	1	100.00%	<div></div>	✓
CVP cov_cg::H...	100.00%	100	100.00%	<div></div>	✓
bin non_seq	1000	1	100.00%	<div></div>	✓
bin idle	101	1	100.00%	<div></div>	✓
bin seq	600	1	100.00%	<div></div>	✓
CVP cov_cg::H...	100.00%	100	100.00%	<div></div>	✓
bin size_byt...	651	1	100.00%	<div></div>	✓
bin size_half...	600	1	100.00%	<div></div>	✓
bin size_wor...	450	1	100.00%	<div></div>	✓
CVP cov_cg::H...	100.00%	100	100.00%	<div></div>	✓
bin single	601	1	100.00%	<div></div>	✓
bin incr	200	1	100.00%	<div></div>	✓
bin wrap4	300	1	100.00%	<div></div>	✓
bin incr4	300	1	100.00%	<div></div>	✓

CROSS cov_c...	100.00%	100	100.00%	<div></div>	✓
bin <write,se...	300	1	100.00%	<div></div>	✓
bin <write,no...	500	1	100.00%	<div></div>	✓
bin <write,idl...	100	1	100.00%	<div></div>	✓
bin <read,se...	300	1	100.00%	<div></div>	✓
bin <read,no...	500	1	100.00%	<div></div>	✓
bin <read,idl...	1	1	100.00%	<div></div>	✓
CROSS cov_c...	100.00%	100	100.00%	<div></div>	✓
bin <write,si...	250	1	100.00%	<div></div>	✓
bin <write,si...	250	1	100.00%	<div></div>	✓
bin <write,si...	400	1	100.00%	<div></div>	✓
bin <read,siz...	200	1	100.00%	<div></div>	✓
bin <read,siz...	350	1	100.00%	<div></div>	✓
bin <read,siz...	251	1	100.00%	<div></div>	✓

CROSS cov_c...	100.00%	100	100.00%	<div></div>	✓
bin <write,wr...	100	1	100.00%	<div></div>	✓
bin <write,si...	350	1	100.00%	<div></div>	✓
bin <write,in...	100	1	100.00%	<div></div>	✓
bin <write,in...	200	1	100.00%	<div></div>	✓
bin <read,wr...	200	1	100.00%	<div></div>	✓
bin <read,si...	251	1	100.00%	<div></div>	✓
bin <read,in...	100	1	100.00%	<div></div>	✓
bin <read,in...	100	1	100.00%	<div></div>	✓

## Assertions:

### Scoreboard Assertions

Assertions							
Name	Assertion Type	Language	Enable	Failure Count	Pass Count	Active	
 /top_sv_unit/ahb_apb_scoreboard/data_write/...	Immediate	SVA	on	0	1		
 /top_sv_unit/ahb_apb_scoreboard/data_read/...	Immediate	SVA	on	0	1		

### AHB to ABP FSM assertions

```
//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	

#### Environment:

In the context of this AHB APB Bridge Verification module, the **environment** class is central to the verification framework. It is responsible for the instantiation and management of all verification components such as the generator (**gen**), driver (**driv**), monitor (**moni**), and scoreboard (**sb**). Communication between these components is facilitated using SystemVerilog mailboxes (**gen2driv**, **driv2sb**, **mail2sb**, and **driv2cor**), which are established in this class.

The environment also contains the logic for executing specific test cases, each represented as a separate task. These test cases define different transaction scenarios, where a certain type of transaction (read or write) is performed with different parameters (e.g., single word, half word, byte, non-sequential, etc.). Each test case sets up the generation of the transaction, drives it to the design under test (DUT), monitors the operation and checks the result in the scoreboard, running these processes in parallel using `fork-join_none` constructs.

In essence, the environment serves as the manager of the verification process, orchestrating the generation, execution, and checking of various transaction scenarios for thorough testing and verification of the AHB APB bridge design under test.

### Generator:

The SystemVerilog **generator** class is a critical component of the AHB APB Bridge Verification project, responsible for generating and managing various transaction types. It consists of a transaction handle **tx** and a mailbox **gen2driv**, where **tx** defines the transaction to be generated and **gen2driv** serves as the conduit to send these transactions to the driver. The **generator** also uses a virtual interface **vif** to interact with the Device Under Test (DUT).

Different tasks within the **generator** class create unique transactions, each representing a distinct test case by assigning specific values to the fields of the **tx** handle. This transaction diversity ensures comprehensive functional coverage of the AHB APB Bridge. Moreover, each transaction is sampled for coverage using the **tx.cov\_cg.sample()** command, which records a snapshot of the transaction post-definition. This functional coverage sampling is a key aspect of the verification process, certifying that all transaction types have been thoroughly generated and tested.

### Driver:

The SystemVerilog **driver** class in the AHB APB Bridge Verification project is tasked with receiving transactions from the **generator** and driving these into the Design Under Verification (DUV). It utilizes a virtual interface **vif** for interacting with the DUV.

The **driver** class is equipped with a transaction handle **tx** and three mailboxes: **gen2driv**, **driv2sb**, and **driv2cor**. **gen2driv** is the mailbox for receiving transactions from the **generator**, **driv2sb** sends transactions to the scoreboard for further verification, and **driv2cor** facilitates the direct sending of transactions to the DUV.

The key routine of the **driver** class is the **drive** task. This task retrieves a transaction from the **generator** through **gen2driv.get(tx)**, subsequently sending it to both the scoreboard and the DUV via **driv2sb.put(tx)** and **driv2cor.put(tx)**.

Following this, the **drive** task uses the virtual interface to directly drive the transaction values to the DUV. Each signal within the DUV is assigned a corresponding value from the transaction. Then, the task awaits a clock edge before proceeding, thereby ensuring synchronization with the DUV's operation cycle. This precise and systematic flow of operations within the **driver** class ensures a thorough and efficient verification process.

### Monitor:

The **ahb\_apb\_monitor** class is a fundamental component of the verification process in a SystemVerilog testbench for the AHB APB Bridge Verification project. This class observes the interface, captures transactions on the bus, and forwards these to the scoreboard for verification against expected results. It acts as a passive and non-intrusive listener to the Design Under Test (DUT).



The **ahb\_apb\_monitor** class specifically observes the signals on the AHB APB bridge interface, forms transaction objects based on the observed transactions, and forwards these to the scoreboard. It operates in a continuous loop, always monitoring the interface for new transactions.

The monitor uses a clocking block **mon\_cb** to synchronously sample the interface signals with the clock. These sampled values are used to form the transaction object, which is then forwarded to the scoreboard via the **mail2sb** mailbox.

In the **watch** task, a transaction is created, and then the monitor waits for any transaction to start. Upon initiation of a transaction, the monitor samples various signals, stores them in the transaction, and then sends the transaction to the scoreboard via **mail2sb.put(tx)**. The monitor's duty is to keep a vigilant eye on the interface and accurately capture transactions, ensuring the efficacy of the verification process.

#### Scoreboard:

The **ahb\_apb\_scoreboard** class serves a pivotal role in the verification process of the AHB APB Bridge Verification project. This class validates the correctness of the Design Under Test (DUT) and includes a memory model that imitates the DUT's behavior.

The **ahb\_apb\_scoreboard** class receives transactions from both the driver and the monitor. This dual-source transaction intake allows the scoreboard to compare expected and actual responses, verifying the accuracy of the transactions. The scoreboard contains methods for handling both write (**data\_write**) and read (**data\_read**) operations.

In a write operation, the scoreboard verifies that the data is correctly written into the memory model. This is done by asserting that the data received from the driver (**tx1.Hwdata**) matches the data in the memory model at the appropriate address (**mem\_tb[temp\_addr]**). If the assertion fails, an error is raised, indicating a failure in the verification process.

For read operations, the scoreboard checks if the data read from the DUT (sourced from the monitor) matches the data stored in the memory model. This is checked by asserting that the data received from the DUT (**tx2.Pwdata**) matches the data in the memory model at the designated address (**mem\_tb[temp\_addr]**). A failure in this assertion flags a discrepancy in the read data and raises an error.

By handling both write and read operations, and rigorously cross verifying the DUT's data against a memory model, the **ahb\_apb\_scoreboard** serves as a crucial checkpoint in the verification process, ensuring the accuracy and reliability of the DUT's functionality.

## Test:

The **test** class plays a pivotal role in a SystemVerilog verification environment. It integrates all the test cases defined in the environment and manages their execution over the course of a simulation run.

Here's what it does:

1. **function new(virtual ahb\_apb\_bfm\_if i):** This function creates a new instance of the environment class, passing the interface object **i** to the environment.
2. **task run():** This task starts the environment and performs the test sequences repeatedly over multiple clock cycles. It starts with the display statement, "in test", and creates the environment with **env.create()**.

Following this, it enters into a repeat loop that executes five times. In each loop iteration, two methods from the environment class are called:

**env.env\_write\_single\_halfword\_nonseq\_single\_Htransfer\_okay()** and **env.env\_read\_single\_byte\_nonseq\_single\_Htransfer\_okay()**, along with **env.env\_read\_single\_word\_nonseq\_single\_Htransfer\_okay()**. These test cases are written to and read from the environment.

The other test cases are commented out, but they can be included in the test sequence by uncommenting them. This repeat loop with multiple test sequences simulates multiple scenarios and states of the AHB APB Bridge to verify its functionality under these different conditions.

After the completion of each repeat loop iteration, the run task waits for the positive edge of the clock (**@(posedge i.clk)**), ensuring that each set of test cases is executed in sync with the clock cycles. This design emulates real-world conditions, as hardware transactions in actual systems occur synchronously with a clock signal.

By managing and executing these tests, the **test** class forms the backbone of the verification process, simulating various operating conditions to ensure the AHB APB Bridge performs as expected under all scenarios.

```

1 class test;
2     environment env; // creates handle
3
4 function new(virtual ahb_apb_bfm_if i);
5     env = new(i);
6 endfunction : new
7
8 task run();
9
10     $display("in test");
11     env.create();
12
13     repeat(50)
14     begin
15
16         $display("in test repeat");
17
18         #5; env.env_write_single_halfword_nonseq_single_Htransfer_okay();
19         // #5; env.env_read_single_byte_nonseq_single_Htransfer_okay();
20         #5; env.env_read_single_halfword_nonseq_single_Htransfer_okay();
21
22         #5; env.env_write_single_byte_nonseq_single_Htransfer_error();
23         #5; env.env_read_incr_halfword_nonseq_incr_Hburst_okay();
24         #5; env.env_write_incr_word_nonseq_incr_Hburst_okay();
25         #5; env.env_read_wrap4_byte_nonseq_wrap4_Hburst_okay();
26         #5; env.env_write_wrap4_halfword_nonseq_wrap4_Hburst_okay();
27         #5; env.env_read_wrap4_word_nonseq_wrap4_Hburst_okay();
28         #5; env.env_write_incr4_byte_nonseq_incr4_Hburst_okay();
29         #5; env.env_read_incr4_halfword_nonseq_incr4_Hburst_okay();
30         #5; env.env_write_incr4_word_nonseq_incr4_Hburst_okay();
31         #5; env.env_read_wrap8_byte_nonseq_wrap8_Hburst_okay();
32         #5; env.env_write_wrap8_halfword_nonseq_wrap8_Hburst_okay();
33         #5; env.env_read_wrap8_word_nonseq_wrap8_Hburst_okay();

```



## Results:

```
# run -all
# in top
# in test
# in test repeat
#          0   write_single_halfword_nonseq_single_Htransfer_okay task in generator
# driver tx262154
# Scoreboard check...
# Write taking place at Time: 0 Address: 00000
# Input Address: c9465
# Input Write Data: eacd5432
# Data Stored: eacd5432
#
#          5   read_single_word_nonseq_single_Htransfer_okay task in generator
# Scoreboard read
# Read taking place at Time: 5 Address: c9465
#          10  write_single_byte_nonseq_single_Htransfer_error task in generator
# Scoreboard check...
# Write taking place at Time: 10 Address: a4684
# Temp address = 10065
# Read data from DUT 00000000
# Data from TB memory 00000000
#
# Input Address: 10065
# Input Write Data: 2b50f3ef
# Data Stored: 2b50f3ef
#
#          15  read_incr_halfword_nonseq_incr_Hburst_okay task in generator
# Scoreboard read
# Read taking place at Time: 15 Address: 10065
# Temp address = 3708b
# Read data from DUT 00000000
```

### Read implementation in the actual DUT:

In the context of this project, the Device Under Test (DUT) implements the read operation in a unique manner. Rather than retrieving the data previously written to a specific location, the DUT returns a constraint random value. This is due to the fact that the Prdata signal, which is intended to hold the data read from a peripheral, is not connected to any actual data-storing peripheral.

Typically, a read operation would involve the system accessing the specific address where data was written and returning the value stored at that location. However, in this DUT, the absence of a connection between the Prdata signal and a data-storing peripheral means that it does not have access to any previously written data. Consequently, whenever a read operation is performed, the DUT generates a constraint random value.

This characteristic of the DUT necessitated a different approach to verifying the read operation. In a standard scenario, one would write data to a specific location, perform a read operation, and then verify that the data returned matches the data that was written. However, given that the DUT always returns a random value when a read operation is performed, this traditional form of verification was not feasible.

It's important to note that this doesn't imply a flaw in the DUT. It's possible that this DUT is designed to operate in this manner, perhaps because it's part of a larger system where the actual data storage is managed elsewhere. However, it does mean that the verification process had to be adjusted to accommodate this behavior.

```

1 module APB_Interface(Pwrite,Pselx,Penable,P
2
3     input Pwrite, Penable;
4     input [2:0] Pselx;
5     input [31:0] Pwdata, Paddr;
6
7     output Pwriteout, Penableout;
8     output [2:0] Pselxout;
9     output [31:0] Pwdataout, Paddrout;
10    output reg [31:0] Prdata;
11
12    assign Penableout=Penable;
13    assign Pselxout=Pselx;
14    assign Pwriteout=Pwrite;
15    assign Paddrout=Paddr;
16    assign Pwdataout=Pwdata;
17
18    always @(*)
19    begin
20        if (~Pwrite && Penable)
21            Prdata=($random)%256;
22        else
23            Prdata=0;
24        end
25    end
26 endmodule

```

```

#
# 20 write_incr_word_nonseq_incr_Hburst_okay task in generator
# Scoreboard check...
# Write taking place at Time: 20 Address: 3708b
# Input Address: 86813
# Input Write Data: d417cec8
# Data Stored: d417cec8
#
# 25 read_wrap4_byte_nonseq_wrap4_Hburst_okay task in generator
# Scoreboard read
# Read taking place at Time: 25 Address: 86813
# Temp address = 559d6
# Read data from DUT 00000000
# Data from TB memory 00000000
#

```

```

# Temp address = 004b0
# Read data from DUT 00000000
# Data from TB memory 144705f0
# ** Error: Data reading failed
#   Time: 50 ns   Scope: top_sv_unit.ahb_apb_scoreboard.data_read File: scoreboard.sv Line: 59
#
#       50 monitor tx='{trans_type:AHB_READ, Haddr:0, Hwdata:0, Hwrite:0, Htrans:0, Hsize:0, Hl
:0, Pwdata:0, Pwrite:0, Pselx:0, hresp:0, hreset:0, cov_cg:{ref to covergroup #cov_cg#}}
#       50 monitor tx='{trans_type:AHB_READ, Haddr:0, Hwdata:0, Hwrite:0, Htrans:0, Hsize:0, Hl
:0, Pwdata:0, Pwrite:0, Pselx:0, hresp:0, hreset:0, cov_cg:{ref to covergroup #cov_cg#}}
# ** Note: $stop    : top.sv(74)
#   Time: 100010 ns Iteration: 0   Instance: /ahb_apb_top
# Break at top.sv line 74

```

Two different Testbench implementations. One that attempts to read the randomly generated Prdata value from the DUV and one that attempts to read the randomly generated written data to the DUV Pwdata (reading the same data we wrote).

## Debugging

During the verification of the AHB APB Bridge, we faced several debugging challenges that taught us valuable lessons and guided us towards a comprehensive understanding of our testbench. Two main challenges we encountered were related to coverage issues and timing issues.

Initially, we were confronted with a coverage issue where our coverage was stagnating at 25%. After detailed analysis, we discovered that the issue was rooted in an error in a typedef enumeration inside the Transaction class. This error resulted in incorrect handling and generation of the transactions, leading to poor coverage.

This was a challenging issue that necessitated detailed debugging. We used various debugging tools and techniques, such as waveform viewing and stepping through the code execution in a simulator, to isolate the problem. Eventually, through meticulous debugging, we corrected the typedef enumeration in the transaction class, and this led to an improvement in our coverage.

The second major issue was a timing problem where all the transactions were being sent at time zero. This was a non-realistic scenario since, in an actual system, transactions occur over different clock cycles. This issue could have caused inaccuracies in our verification process and led to false positives or negatives.

We tackled this issue by closely studying the transaction generator and the sequence item classes. With the assistance of our teaching assistant, Phanindra Vemireddy, we managed to correct this timing issue. His input was instrumental in helping us understand the root cause and fix these issues.

For efficient debugging, we commented out most of the test cases and focused on one or a few tests at a time. This allowed us to concentrate on each test case in depth and understand the exact flow of transactions, thereby making the debugging process more manageable and effective.

These debugging experiences emphasized the importance of thorough code examination and taught us how to systematically approach and resolve complex issues. Despite the challenges, these debugging processes significantly contributed to our understanding of the system, making us better verification engineers.

We also found minor issues in the DUT such that a signal was listed twice in the port list (the Hreadyout signal)

```
'include "APB_Interface.v"
'include "APB_Controller.v"
'include "AHB_Slave_Interface.v"
'include "AHB_Master.v"

module
bridge_Top(Hclk,Hresetn,Hwrite,Hreadyin,Hreadyout,Hwdata,Haddr,Htrans,Prdata,Penable,Pwrite,Pselx,Paddr,Pwdata,Hreadyout,
input Hclk,Hresetn,Hwrite,Hreadyin;
input [31:0] Hwdata,Haddr,Prdata;
input[1:0] Htrans;
output Penable,Pwrite,Hreadyout;
output [1:0] Hresp;
output [2:0] Pselx;
output [31:0] Paddr,Pwdata;
output [31:0] Hrddata;
```

## Summary

Our verification journey began with a straightforward testbench that utilized modules and transactions without the complexities of classes, mailboxes, or an environment. As we progressed, we gradually transformed this structure into one that is based on Object Oriented Programming (OOP). This allowed us to create classes and instantiate them as objects whenever necessary, providing us with a more flexible and scalable verification environment.

Our verification IP (VIP) was constructed based on the ARM specifications, which served as the blueprint for our AHB to APB bridge. This approach allowed us to focus on verifying the design against the intended specifications, rather than the RTL Design of the bridge itself.

The components of our verification environment included a Bus Functional Model (BFM) interface, a driver, a generator, an environment, a monitor, a scoreboard, and test classes. These components were all encapsulated within a top-level module, creating a comprehensive and robust verification framework.

As our verification environment evolved, we began to uncover potential issues within the design and the verification environment itself. This led us to the debugging phase, where

we iteratively refined our VIP. Throughout this process, we developed four different versions of our VIP. These versions can be broadly categorized into two groups: those that attempt to read the Prdata signal from the Device Under Test (DUT), and those that do not.

Through our debugging efforts, we discovered that our specific DUT returned constraint random data when we attempted to read from Prdata. This was because it was not connected to any actual peripheral, and therefore did not store the data we wrote to it. As a result, any attempts to read data from a peripheral after writing data into it were unsuccessful, as the read data was always random.

Despite this challenge, we successfully performed write seq, non-seq, reset, and other operations for a byte, halfword, and word. We achieved 100% coverage in our VIP and hit all the bins we had initially set, demonstrating the effectiveness of our verification process.

To further ensure the correctness of our design, we implemented two assertions in the scoreboard and numerous in-line SystemVerilog Assertion (SVA) assertions in the DUT Finite State Machine (FSM). These assertions were crucial in verifying the correct transitions within the FSM, based on the stimulus input we sent to the DUT.

## Conclusion

In conclusion, our verification environment, built using SystemVerilog, provides an efficient, reliable, and comprehensive solution for the validation of AHB-APB Bridge designs. The architecture and its components, as detailed in this report, collectively contribute to a robust verification strategy capable of handling the complexities and nuances of modern digital design validation. The use of SystemVerilog classes, interfaces, and tasks, along with the OOP paradigm, facilitates scalability, reusability, and maintainability of the testbench, significantly enhancing the effectiveness of the verification process. However, as the industry steadily migrates towards an even more structured and uniform verification approach, the transition to a Universal Verification Methodology (UVM) based environment becomes essential. The subsequent report will detail the transformation of this SystemVerilog based verification environment into a UVM-based verification environment, offering an even higher level of abstraction, automation, and standardization, thereby enhancing productivity, interoperability, and reusability.