UVM Based Verification for AHB to APB Bridge

Project Report

ARM® AMBA®

Interconnect Standards

Prepared by

Mohamed Ehab

Contents

1. Introduction

With the increasing complexity of modern SoCs (System-on-Chip), an efficient and standardized communication protocol is essential to ensure seamless data transfer between different components. The **Advanced Microcontroller Bus Architecture** (AMBA**)**, developed by ARM**,** provides a scalable and modular solution for interconnecting various blocks within an SoC. Among its widely adopted protocols are AXI (Advanced eXtensible Interface), AHB (Advanced High-performance Bus), and APB (Advanced Peripheral Bus).

1.1 AMBA Protocol

The AMBA protocol is an industry-standard interconnect for SoCs, ensuring compatibility and performance optimization. It consists of:

- AXI (Advanced eXtensible Interface): A high-performance bus that supports burst transactions, out-of-order execution, and multiple outstanding requests. Used in high-speed, data-intensive applications like processors and memory controllers.

- AHB (Advanced High-performance Bus): A single-clock, pipelined protocol that supports burst transfers, split transactions, and single-cycle bus access. It is commonly used for high-performance embedded systems.

- APB (Advanced Peripheral Bus): A simpler, low-power bus primarily designed for slow peripherals such as timers, GPIOs, and UARTs. Unlike AXI and AHB, APB does not support burst transfers, making it more energy-efficient.

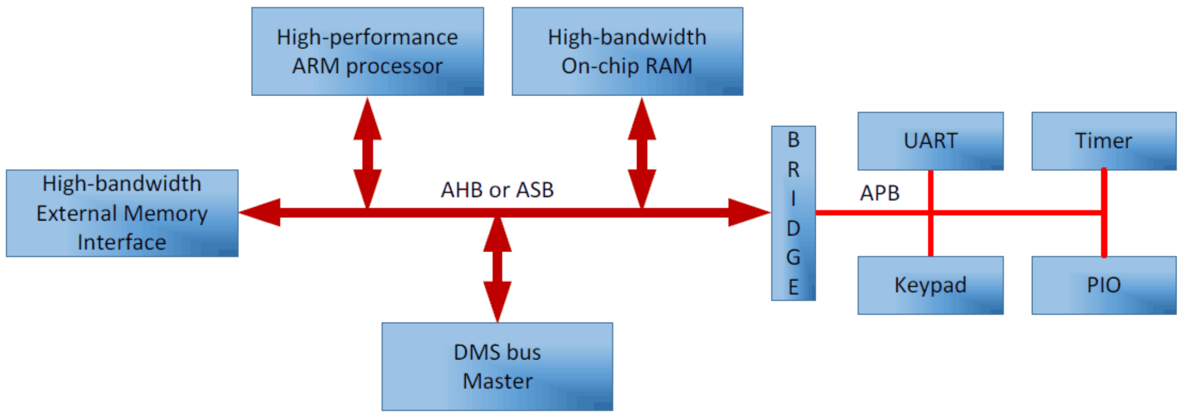

High-performance ARM processor
High-bandwidth On-chip RAM
High-bandwidth External Memory Interface
AHB or ASB
DMS bus Master
BRIDGE
APB
UART
Timer
Keypad
PIO

Fig. (1) typical AMBA system

1.2 Role of AHB-APB Bridge in Systems

The AHB-APB bridge serves as an interface between the high-performance AHB and the low-power APB. It plays a crucial role in translating high-speed AHB transactions into a format that APB peripherals can handle.

Key functions of the bridge include:

- Clock domain crossing (CDC): AHB and APB may operate at different clock speeds, requiring proper synchronization.
- Protocol conversion: Converts burst transactions from AHB into single transfers for APB.
- Address and control signal translation: Ensures proper handshaking between AHB and APB components.
- Transaction buffering: Manages data transfer between master and slave devices.

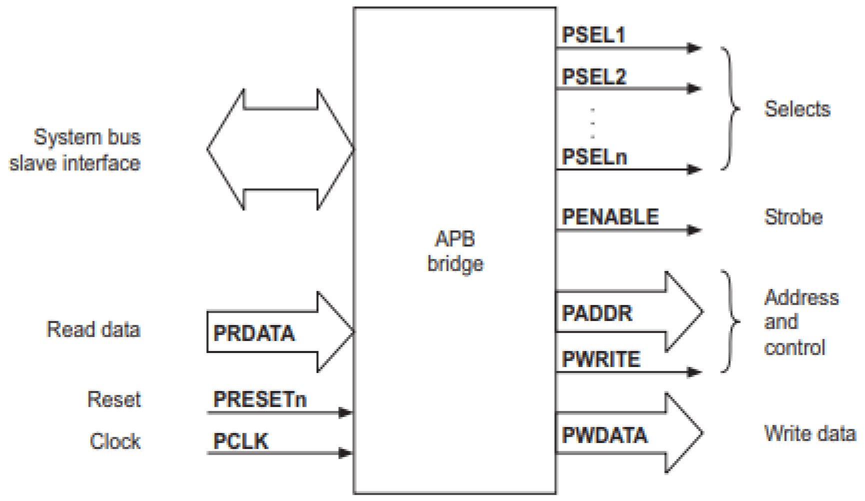


Fig. (2) AHB-APB Bridge Block Diagram

1.3 Main Objectives of the Project

The project aims to:

- Verify the bridge using UVM methodology, ensuring it meets functional requirements.
- Address CDC Challenges, ensuring reliable operation across different clock domains.
- Measure Performance and Coverage, ensuring the bridge works correctly under different conditions.
- Provide a Structured Test Plan, including test cases for different scenarios like burst transactions, error handling, and latency measurements.

2. Design Under Test (DUT)

The AHB-to-APB bridge is the core design under test (DUT) in this project. Its primary function is to enable seamless communication between the high-performance AHB bus and the low-power APB bus by converting AHB transactions into APB-compatible signals. The bridge is responsible for protocol translation, clock domain synchronization, and ensuring reliable data transfer.

2.1 RTL Components

The DUT consists of two key components:

1. The AHB Slave Interface, which receives transactions from the AHB master and processes them.

2. The APB Master Interface, which translates the transactions into APB protocol signals and communicates with APB peripherals.

Additionally, a Clock Domain Crossing (CDC) mechanism is implemented to handle clock speed mismatches between AHB and APB, ensuring stable operation.

2.2 RTL Code Repository

The RTL implementation of the AHB-to-APB bridge used in this project is sourced from an open-source GitHub repository. The Verilog implementation ensures modularity, ease of integration, and flexibility in parameterizing data widths and clock configurations.

📎 GitHub Repository : Link

2.3 Clock Domain Crossing Mechanism

Since AHB and APB often operate at different clock frequencies, a Clock Domain Crossing (CDC) mechanism is required to prevent metastability issues when transferring control and data signals. In this design, a synchronization depth (SYNC_DEPTH) of 3 is used to improve metastability robustness.
Key CDC Considerations in the Bridge:

- Metastability Reduction: Three-stage flip-flop synchronization ensures safe signal transfer from the faster AHB domain to the slower APB domain.

- Handshake Mechanism: Signals like HREADY, HWRITE, and HTRANS are properly synchronized to ensure data integrity.

- FIFO Buffers : Used to store AHB transactions temporarily before sending them to APB, preventing data loss due to clock differences.

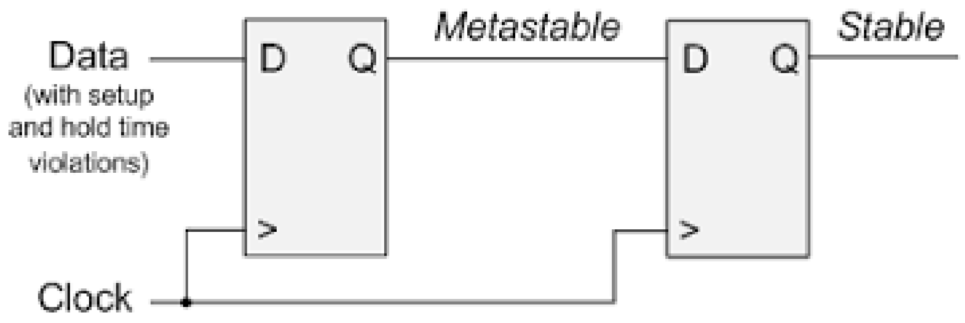


Fig.(3) Synchronizers between the two clock domains

3. **Verification Methodology**

To ensure the correctness, functionality, and robustness of the AHB-to-APB bridge, a Universal Verification Methodology (UVM) testbench is used. UVM provides a scalable, reusable, and modular framework for verifying complex designs by enabling constrained-random testing, functional coverage analysis, and scoreboard-based result checking.

3.1 Scalability and Reusability

UVM allows the creation of generic, reusable verification components, reducing development time for future projects. The modular structure makes it easy to extend the testbench when verifying different AMBA-based designs.

3.2 Coverage-Driven Verification

Instead of manually defining every possible test scenario, UVM leverages constrained-random stimulus generation to explore different test cases efficiently. Functional coverage is used to measure test completeness, ensuring that all features of the DUT are exercised.

3.3 Improved Debugging and Reporting

- UVM provides detailed transaction-level logging to help debug and analyze failures.

- Built-in reporting mechanisms make it easier to track issues, improving verification efficiency.

4. Test Plan

The test plan defines the strategy for verifying the AHB-to-APB Bridge, ensuring that it meets the functional and performance requirements. The bridge plays a crucial role in converting AHB transactions into APB transactions, handling various scenarios such as single and burst transfers, busy conditions, and error responses.

This section outlines the test cases used to validate the bridge's functionality, covering both basic and corner cases. Each test is designed to verify the correct handling of read and write operations, burst transactions, and error conditions, ensuring that the bridge operates as expected under all circumstances.

The test plan includes:

1. **Test Cases** – A detailed list of test scenarios covering different operational conditions.

2. **Coverage Model** – Ensuring that all key aspects of the design are tested, including functional coverage and code coverage.

3. **Assertions** – Utilizing SystemVerilog Assertions (SVA) to check protocol compliance and detect potential issues.

4.1 Test Cases

Test Case	Description	HTRANS	HWRITE	HBURST	PREADY	PSLVERR
single_write_test	Tests a single write operation.	NONSEQ	HIGH	Single	HIGH	LOW
single_read_test	Tests a single read operation.	NONSEQ	LOW	Single	HIGH	LOW
double_write_test	Tests multiple successive write operations in a burst.	NONSEQ → SEQ	HIGH	INCR	HIGH	LOW
double_read_test	Tests multiple successive read operations in a burst.	NONSEQ → SEQ	LOW	INCR	HIGH	LOW
write_busy_test	Tests slave busy response during a write operation.	NONSEQ	HIGH	Single	LOW	LOW
read_busy_test	Tests slave busy response during a read operation.	NONSEQ	LOW	Single	LOW	LOW
read_write_test	Tests back-to-back read and write operations in a burst.	NONSEQ → SEQ	Toggle	INCR4	HIGH	LOW
write_error_test	Tests slave error response during a write operation.	NONSEQ	HIGH	Single	HIGH	HIGH
read_error_test	Tests slave error response during a read operation.	NONSEQ	LOW	Single	HIGH	HIGH

4.2 Coverage Model

To ensure thorough verification of the AHB to APB Bridge, a coverage-driven verification approach is implemented ,ensuring that all corner cases, transactions, and protocol-specific behaviors are exercised. The coverage model is structured as follows:

Covergroup	What it Covers
Reset Coverage	Tracks reset transitions of HRESETn (AHB) and PRESETn (APB).
Bridge Selection Coverage	Checks whether the APB bridge is selected via HSEL.
Operation Type Coverage	Differentiates between read (HWRITE=0) and write (HWRITE=1) operations.
Transfer Size Coverage	Tracks different transfer sizes (HSIZE): Byte, Halfword, Word.
Protection Coverage	Covers different values of the HPROT signal to track security attributes.
Transfer Type Coverage	Covers different types of AHB transactions (HTRANS): IDLE, BUSY, SEQ, NONSEQ.
Ready Signal Coverage	Tracks master (HREADY) and slave (PREADY) readiness.
Error Handling Coverage	Checks for error scenarios by tracking PSLVERR assertion.

4.3 Assertions

SystemVerilog Assertions (SVA) used to verify the correct operation of the AHB-APB bridge. The assertions ensure protocol compliance by checking timing relationships, signal stability, and expected responses to various conditions. Each property captures a specific design behavior, and assertions validate whether the design meets these expectations during simulation. Coverage properties are also included to track occurrences of critical events for verification completeness.

Property	Description
p_hreadyout_falls	Ensures HREADYOUT deasserts when HREADY and HSEL are HIGH while HTRANS is NONSEQ or SEQ.
p_pready_low	Ensures APB/AHB signals remain stable when PREADY is LOW.
p_penable_duration	Ensures PENABLE remains HIGH for only one cycle while PREADY is HIGH.
p_pslverr_response	Checks HRESP and HREADYOUT timing after PSLVERR, ensuring HRESP rises after 7 cycles and HREADYOUT rises 1 cycle after HRESP.
p_psel_timing	Ensures PSEL asserts exactly 7 cycles after a valid AHB transfer request.

5. Testbench Envoronment

The verification of the AHB-APB bridge is performed using a UVM-based testbench, which provides a modular and reusable verification infrastructure. The testbench consists of various components that interact to generate stimulus, monitor responses, and verify protocol compliance through assertions and coverage collection. The testbench environment block diagram as shown in fig. (4) illustrates the hierarchical structure and interactions among these components.

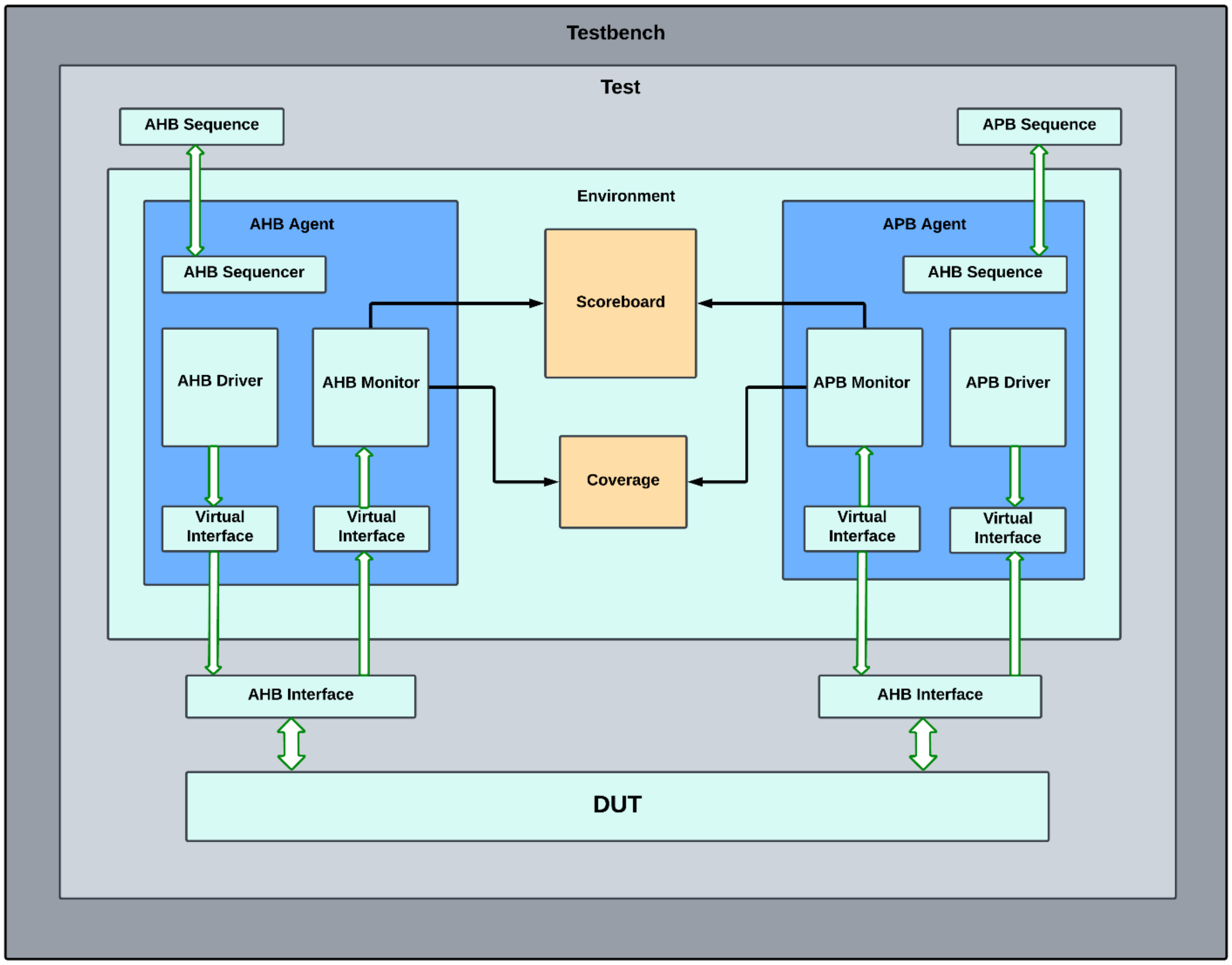


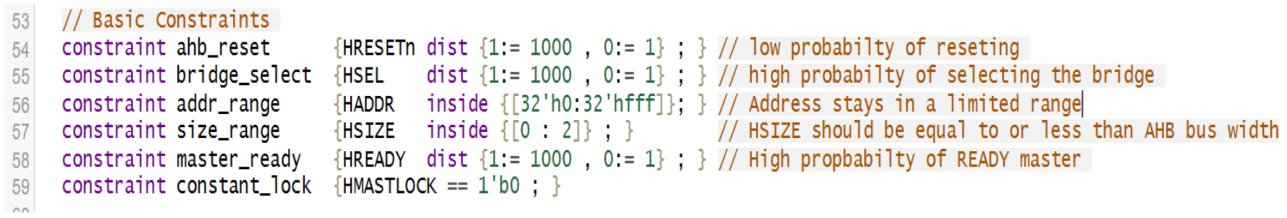
Fig.(4) Testbench environment block diagram

The Discussion will follow a bottom-up approach, starting from the fundamental sequence items and building up to higher-level components that form the complete testbench

5.1 Sequence Item

5.1.1 AHB Sequence Item

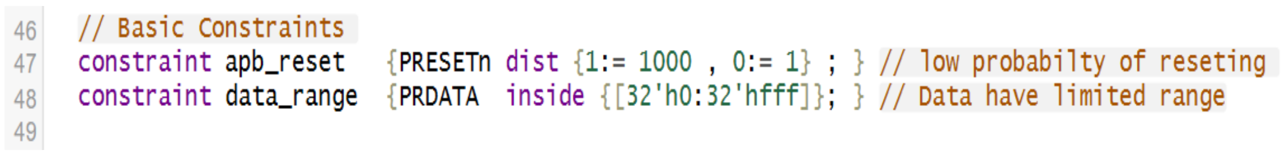
Includes randomized AHB interface signals representing an AHB transaction and incorporates basic constraints to ensure proper protocol compliance and valid communication.



```
// Basic Constraints
constraint ahb_reset      {HRESETn dist {1:= 1000 , 0:= 1} ; } // low probabilty of reseting
constraint bridge_select  {HSEL    dist {1:= 1000 , 0:= 1} ; } // high probabilty of selecting the bridge
constraint addr_range     {HADDR   inside {[32'h0:32'hfff]}; } // Address stays in a limited range
constraint size_range     {HSIZE   inside {[0 : 2]} ; }        // HSIZE should be equal to or less than AHB bus width
constraint master_ready   {HREADY  dist {1:= 1000 , 0:= 1} ; } // High propbabilty of READY master
constraint constant_lock  {HMASTLOCK == 1'b0 ; }
```

5.1.2 APB Sequence Item

Includes randomized APB interface signals representing an APB transaction and defines basic constraints to maintain proper protocol adherence and reliable data transfer.



```
// Basic Constraints
constraint apb_reset   {PRESETn dist {1:= 1000 , 0:= 1} ; } // low probabilty of reseting
constraint data_range  {PRDATA  inside {[32'h0:32'hfff]}; } // Data have limited range
```

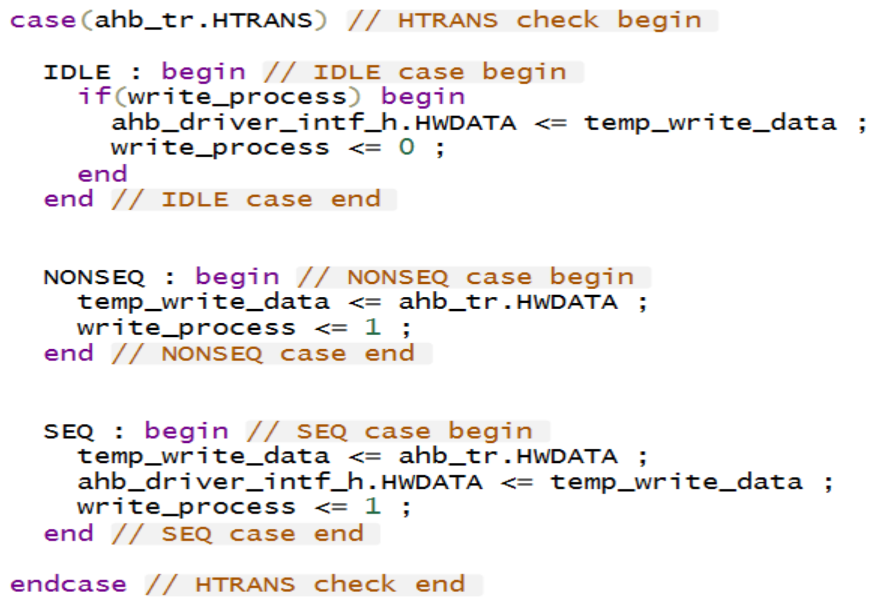
5.2 Agent

The agent is responsible for generating, driving, and monitoring transactions on the respective bus interfaces. It consists of four key components: Driver, Monitor and Sequencer. The AHB and APB agents operate independently, each handling transactions specific to its respective protocol.

5.2.1 AHB Agent

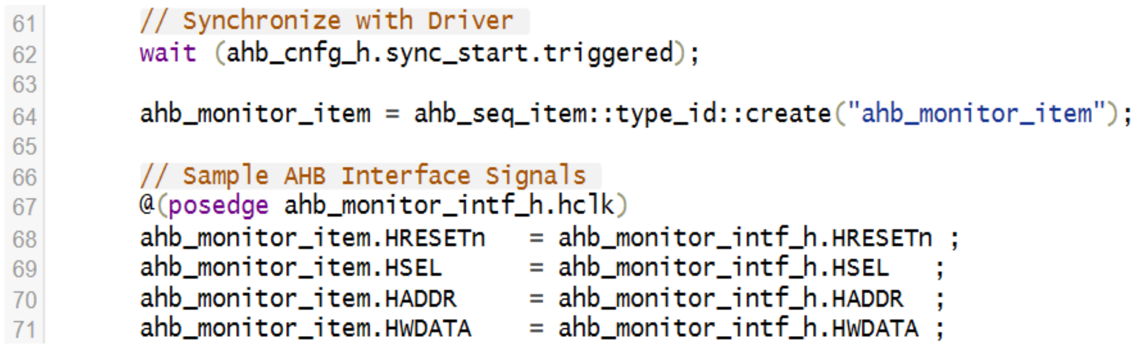
The AHB agent is responsible for managing AHB transactions. It ensures that transactions adhere to the AHB protocol while interfacing with the testbench components. It consists of the following:

- **AHB Driver**: Mimics the behavior of an AHB **master** by driving transactions onto the AHB interface. It converts sequence items from the sequencer into signal activity while handling handshake mechanisms and protocol timing requirements.



```
case(ahb_tr.HTRANS) // HTRANS check begin

  IDLE : begin // IDLE case begin
    if(write_process) begin
      ahb_driver_intf_h.HWDATA <= temp_write_data ;
      write_process <= 0 ;
    end
  end // IDLE case end


  NONSEQ : begin // NONSEQ case begin
    temp_write_data <= ahb_tr.HWDATA ;
    write_process <= 1 ;
  end // NONSEQ case end


  SEQ : begin // SEQ case begin
    temp_write_data <= ahb_tr.HWDATA ;
    ahb_driver_intf_h.HWDATA <= temp_write_data ;
    write_process <= 1 ;
  end // SEQ case end

endcase // HTRANS check end
```

- **AHB Monitor**: Observes AHB transactions passively, extracting meaningful data for functional coverage and scoreboard comparison. It synchronizes with the driver using **events** to prevent sampling incomplete or unwanted transactions.



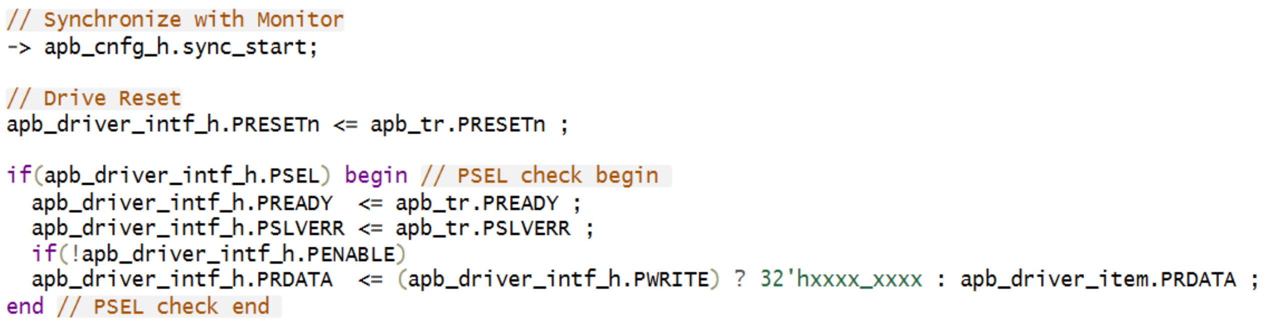
```
// Synchronize with Driver
wait (ahb_cnfg_h.sync_start.triggered);

ahb_monitor_item = ahb_seq_item::type_id::create("ahb_monitor_item");

// Sample AHB Interface Signals
@(posedge ahb_monitor_intf_h.hclk)
ahb_monitor_item.HRESETn   = ahb_monitor_intf_h.HRESETn ;
ahb_monitor_item.HSEL      = ahb_monitor_intf_h.HSEL    ;
ahb_monitor_item.HADDR     = ahb_monitor_intf_h.HADDR   ;
ahb_monitor_item.HWDATA    = ahb_monitor_intf_h.HWDATA  ;
```

- **AHB Sequencer**: Acts as an intermediary between the test sequences and the driver, supplying transaction-level stimulus for execution.

5.2.2 APB Agent

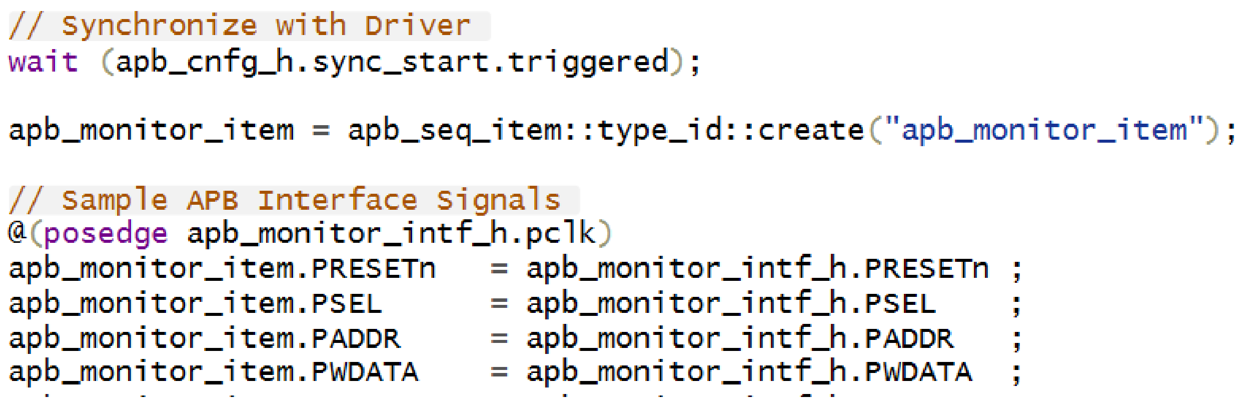
The APB agent is responsible for handling transactions on the APB bus, ensuring correct protocol execution in a simpler, non-pipelined manner. It consists of the following components:

- **APB Driver**: Mimics the behavior of an APB slave, responding to transactions initiated by the AHB master. It ensures proper control signal timing and executes read/write operations as per APB protocol requirements.



```
// Synchronize with Monitor
-> apb_cnfg_h.sync_start;

// Drive Reset
apb_driver_intf_h.PRESETn <= apb_tr.PRESETn ;

if(apb_driver_intf_h.PSEL) begin // PSEL check begin
  apb_driver_intf_h.PREADY  <= apb_tr.PREADY ;
  apb_driver_intf_h.PSLVERR <= apb_tr.PSLVERR ;
  if(!apb_driver_intf_h.PENABLE)
  apb_driver_intf_h.PRDATA  <= (apb_driver_intf_h.PWRITE) ? 32'hxxxx_xxxx : apb_driver_item.PRDATA ;
end // PSEL check end
```

- **APB Monitor**: Passively captures and analyzes transactions on the APB bus. Like the AHB monitor, it utilizes events to synchronize with the driver and prevent sampling of incomplete or unintended transactions.



```
// Synchronize with Driver
wait (apb_cnfg_h.sync_start.triggered);

apb_monitor_item = apb_seq_item::type_id::create("apb_monitor_item");

// Sample APB Interface Signals
@(posedge apb_monitor_intf_h.pclk)
apb_monitor_item.PRESETn   = apb_monitor_intf_h.PRESETn ;
apb_monitor_item.PSEL      = apb_monitor_intf_h.PSEL    ;
apb_monitor_item.PADDR     = apb_monitor_intf_h.PADDR   ;
apb_monitor_item.PWDATA    = apb_monitor_intf_h.PWDATA  ;
```

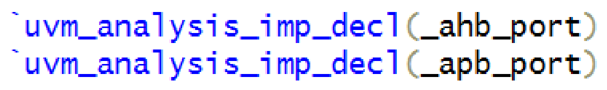
- **APB Sequencer**: Generates and schedules sequence items for the APB driver, facilitating controlled and systematic transaction execution.

5.3 Scoreboard

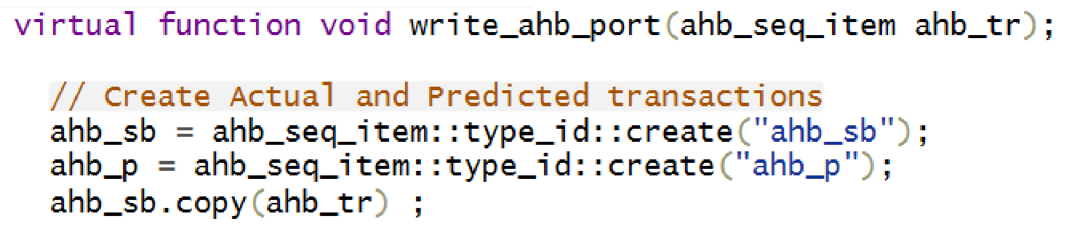
The scoreboard is a crucial component of the UVM testbench, responsible for functional correctness checking by comparing expected and actual transactions. In the AHB-APB bridge verification, the scoreboard ensures that data integrity is maintained as transactions pass from the AHB interface to the APB interface.

The scoreboard receives transactions from both the AHB monitor and APB monitor. It maintains a reference model that predicts the expected output transactions based on the AHB input transactions. These predicted transactions are then compared with the actual transactions observed on the APB interface. Any mismatches indicate a potential design issue. If there is a mismatch, the scoreboard logs an error, providing details such as incorrect address, data corruption, or unexpected response signals.

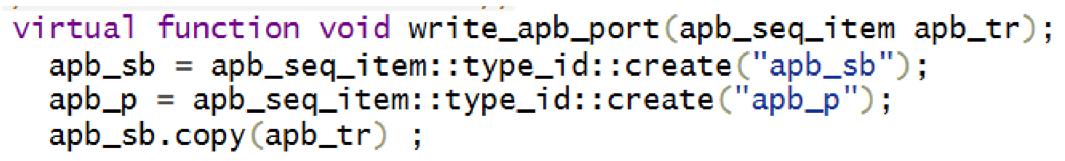
To handle transactions from both the AHB monitor and APB monitor, we use separate analysis ports. This allows the scoreboard to receive transactions independently from each interface. These macros define two distinct analysis ports:



```
`uvm_analysis_imp_decl(_ahb_port)
`uvm_analysis_imp_decl(_apb_port)
```



```
virtual function void write_ahb_port(ahb_seq_item ahb_tr);

  // Create Actual and Predicted transactions
  ahb_sb = ahb_seq_item::type_id::create("ahb_sb");
  ahb_p = ahb_seq_item::type_id::create("ahb_p");
  ahb_sb.copy(ahb_tr) ;
```



```
virtual function void write_apb_port(apb_seq_item apb_tr);
  apb_sb = apb_seq_item::type_id::create("apb_sb");
  apb_p = apb_seq_item::type_id::create("apb_p");
  apb_sb.copy(apb_tr) ;
```

This separation ensures that each monitor sends its transactions to the appropriate processing function inside at different clock speed, making transaction comparison and debugging easier.

5.4 Coverage

The coverage class in UVM is used to measure how well the testbench exercises different scenarios in the AHB-APB bridge. It ensures that all corner cases and protocol-specific behaviors are tested.

```
// Trasfer Size Covergroup
covergroup Trasfer_Size_cg ;
  Trasfer_Size : coverpoint ahb_cov_item.HSIZE {bins Byte     = {0} ;
                                                bins Halfword = {1} ;
                                                bins Word     = {2} ;}
endgroup

// Protection Covergroup
covergroup Protection_cg ;
  Protection : coverpoint ahb_cov_item.HPROT {bins hprot[] = {[0:15]}; }
endgroup

// Transfer Type Covergroup
covergroup Transfer_Type_cg ;
  Transfer_Type : coverpoint ahb_cov_item.HTRANS {bins Idle   = {IDLE}   ;
                                                  bins Busy   = {BUSY}   ;
                                                  bins Seq    = {SEQ}    ;
                                                  bins Nonseq = {NONSEQ} ;}
endgroup

// Ready Covergroup
covergroup Master_Ready_cg ;
  Master_Ready : coverpoint ahb_cov_item.HREADY {bins Master_Ready = {1} ;
                                                 bins Master_Busy  = {0} ;}
endgroup
```

5.5 Environment

The UVM environment class serves as the top-level verification component that integrates various testbench components, including agents, the scoreboard, and the coverage collector. It ensures proper communication between these components and facilitates transaction-level verification.

```
// Build Phase
  function void build_phase(uvm_phase phase);
   super.build_phase(phase);
    ahb_agent_h   = ahb_agent   ::type_id::create("ahb_agent_h"  ,this);
    apb_agent_h   = apb_agent   ::type_id::create("apb_agent_h"  ,this);
    scoreboard_h = scoreboard ::type_id::create("scoreboard_h",this);
    coverage_h    = coverage    ::type_id::create("coverage_h"   ,this);
  endfunction :build_phase
```

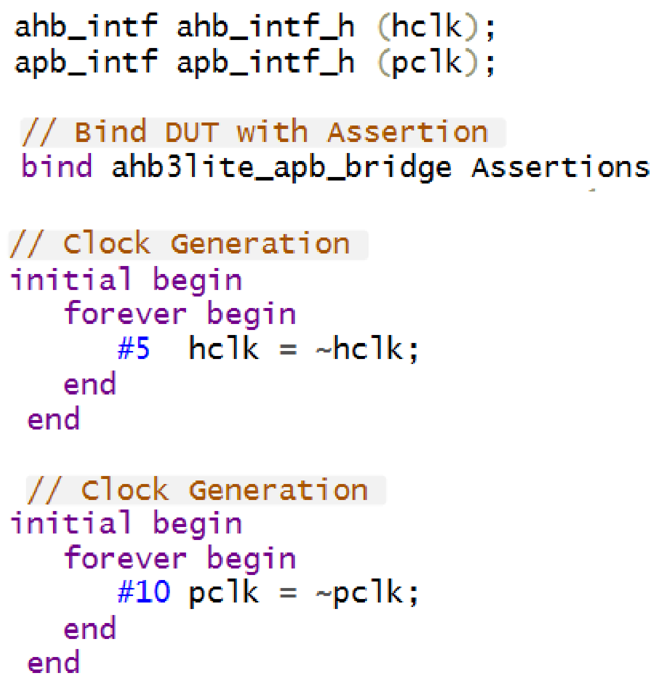
```
// Connect Phase
  function void connect_phase (uvm_phase phase);
    super.connect_phase(phase);
    ahb_agent_h.ahb_monitor_h.ahb_monitor_ap.connect(scoreboard_h.ahb_ap)   ;
    ahb_agent_h.ahb_monitor_h.ahb_monitor_ap.connect(coverage_h.ahb_cov_ap) ;
    apb_agent_h.apb_monitor_h.apb_monitor_ap.connect(scoreboard_h.apb_ap)   ;
    apb_agent_h.apb_monitor_h.apb_monitor_ap.connect(coverage_h.apb_cov_ap) ;
endfunction :connect_phase
```

5.6 Testbench Top

The Testbench Top serves as the integration point for all verification components, including:

- DUT instantiation and interface connections
- Clock generation
- Assertion module binding
- UVM environment integration

It ensures that the DUT operates in a realistic simulation environment and enables functional verification through the UVM methodology.



```
ahb_intf ahb_intf_h (hclk);
apb_intf apb_intf_h (pclk);

 // Bind DUT with Assertion
 bind ahb3lite_apb_bridge Assertions

// Clock Generation
initial begin
   forever begin
      #5  hclk = ~hclk;
   end
 end

 // Clock Generation
initial begin
   forever begin
      #10 pclk = ~pclk;
   end
 end
```

6. Results

The verification process for the AHB-APB bridge involved running multiple test scenarios to ensure protocol compliance and functional correctness. The results are categorized into two key aspects: coverage features and detected bugs.

6.1 Covered Features

The coverage model was designed to track all possible scenarios of AHB-APB transactions, ensuring complete verification of the design. The following key features were successfully covered:

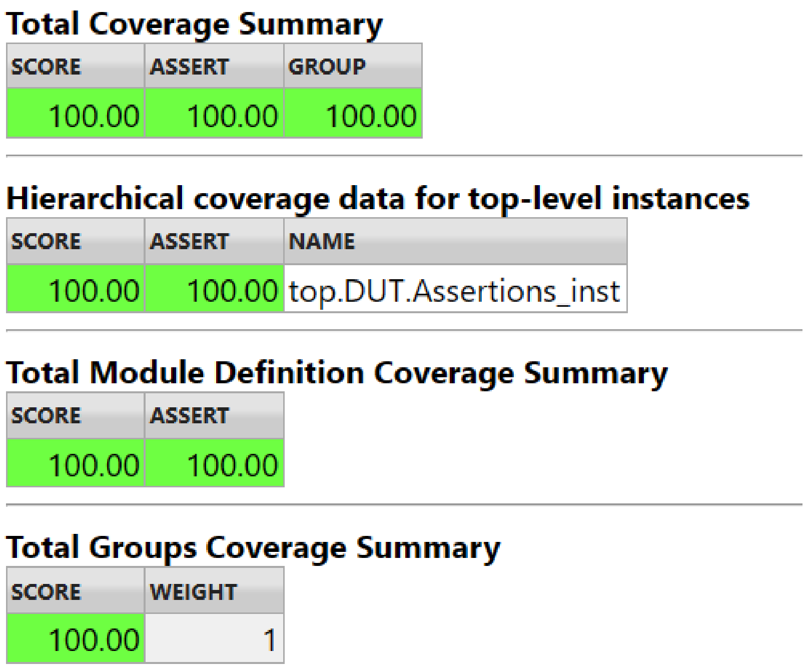
- **Single and burst transactions:** Verified correct data transfer for both read and write operations in single and burst modes (INCR, INCR4, WRAP4, etc.).

- **Back-to-back read and write operations:** Ensured proper sequential execution of transactions across the bridge.

- **Busy and error scenarios:** Simulated scenarios where PREADY is low or PSLVERR is high, confirming the correct response handling.

- **Clock domain crossing (CDC) validation:** Ensured safe AHB to APB clock domain transfer using synchronization techniques.

- **Protocol compliance:** Ensured HTRANS, HWRITE, HREADY, PSEL, PENABLE, and PSLVERR followed the correct transitions and behaviors.



Total Coverage Summary

SCORE	ASSERT	GROUP
100.00	100.00	100.00

Hierarchical coverage data for top-level instances

SCORE	ASSERT	NAME
100.00	100.00	top.DUT.Assertions_inst

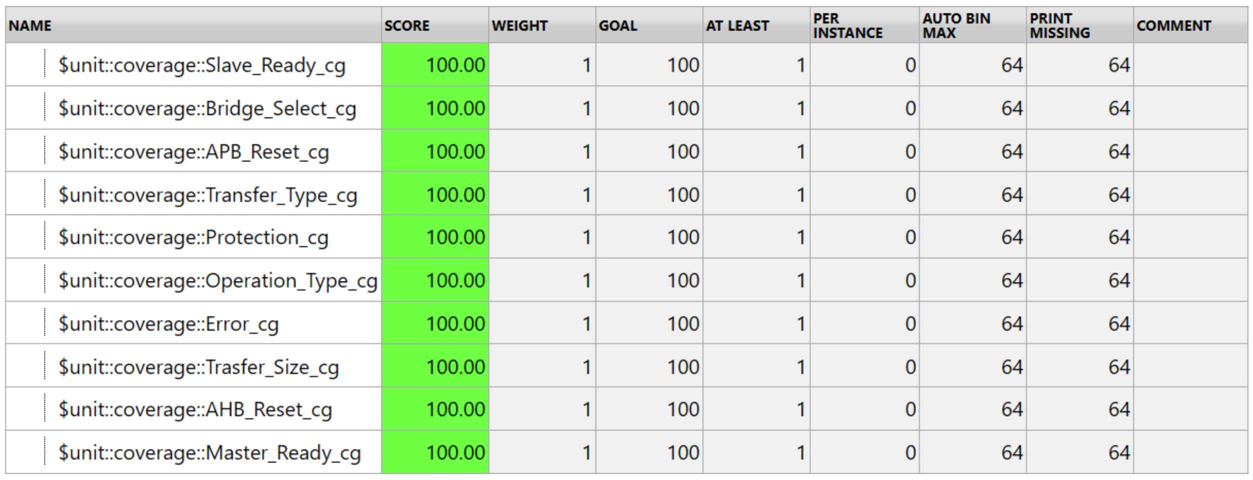
Total Module Definition Coverage Summary

SCORE	ASSERT
100.00	100.00

Total Groups Coverage Summary

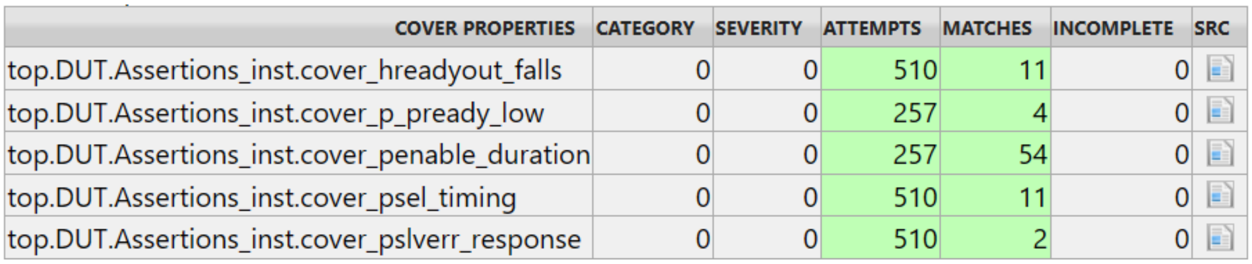
SCORE	WEIGHT
100.00	1

Fig.(6) Total Coverage



NAME	SCORE	WEIGHT	GOAL	AT LEAST	PER INSTANCE	AUTO BIN MAX	PRINT MISSING	COMMENT
$unit::coverage::Slave_Ready_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Bridge_Select_cg	100.00	1	100	1	0	64	64	
$unit::coverage::APB_Reset_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Transfer_Type_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Protection_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Operation_Type_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Error_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Trasfer_Size_cg	100.00	1	100	1	0	64	64	
$unit::coverage::AHB_Reset_cg	100.00	1	100	1	0	64	64	
$unit::coverage::Master_Ready_cg	100.00	1	100	1	0	64	64	

Fig.(7) Covered groups



COVER PROPERTIES	CATEGORY	SEVERITY	ATTEMPTS	MATCHES	INCOMPLETE	SRC
top.DUT.Assertions_inst.cover_hreadyout_falls	0	0	510	11	0	
top.DUT.Assertions_inst.cover_p_pready_low	0	0	257	4	0	
top.DUT.Assertions_inst.cover_penable_duration	0	0	257	54	0	
top.DUT.Assertions_inst.cover_psel_timing	0	0	510	11	0	
top.DUT.Assertions_inst.cover_pslverr_response	0	0	510	2	0	

Fig(8) Covered properties

6.2 Detected Errors

During the verification, an issues were identified in the initial design implementation. The APB slave was expected to respond with PSLVERR in 2 clock cycles, but it took 3 cycles instead. This resulted in an additional delay, violating protocol timing constraints.

7. Conclusion

The AHB-APB bridge verification successfully covered all expected features, validating the design against protocol requirements. However, critical timing violations and transaction misalignment issues were identified, necessitating fixes in the RTL implementation. Further optimizations can improve performance and protocol compliance in future iterations.

8. References

- Giridhar, Perumalla, and Priyanka Choudhury. "Design and Verification of AMBA AHB." *2019 1st International Conference on Advanced Technologies in Intelligent Control, Environment, Computing & Communication Engineering (ICATIECE)*. IEEE, 2019.

- Jain, Padmaprabha, and Satheesh Rao. "Design and verification of advanced microcontroller bus architecture-advanced peripheral bus (AMBA-APB) protocol." *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*. IEEE, 2021.