Verification of APB VIP

Malyka Awais

Muhammad Waqar

Malik Nauman

December 17, 2024

Contents

List of Figures

Acronyms

UVM Universal Verification Methodology

UVC Universal Verification Component

Chapter 1

Introduction

The Advanced Peripheral Bus (APB) is part of the Advanced Microcontroller Bus Architecture (AMBA) 3 protocol family. It is known for being a cost-effective interface, specifically optimized to minimize power consumption and reduce interface complexity. The non-pipelined advanced peripheral bus (APB) is used to connect to peripherals with limited bandwidth that don't require the high performance AXI protocol. The integration of APB peripherals into a design flow is as simple as connecting a signal transition to the clock's rising edge. Each transfer takes at least two Cycles (setup cycle and access cycle). The peripheral devices' programmable control registers can be accessed using it. The APB can be interfaced with AMBA AHB (Advanced high-performance bus), AMBA AHB Lite (Advanced high-performance bus lite), AMBA AXI (Advanced extensible interface), AMBA AXI4 lite (Advanced extensible interface lite).

1.1 Motivation

The need to ensure communication between low-bandwidth peripherals in complex system-on-a-chip (SoC) architectures is the reason for using UVM (Universal Verification Methodology) to test the AMBA APB protocol. As verification engineers, our primary focus is to ensure that the APB implementation correctly adheres to the specified protocol, thus ensuring efficient and seamless integration.

1.2 Problem Statement

The problem statement for this project is specifically focused on the verification of the AMBA APB protocol, with special target on the master and slave Universal Verification Components. Considering the increasing complexity of modern SoC designs, guaranteeing the proper implementation of an APB protocol is required to ensure that the varied peripherals communicate reliably. The main challenge is to confirm whether the master and slave Universal Verification Components (UVCs) correctly adhere to the APB specification when dealing with cases involving signal integrity, timing anomalies, and protocol adherence. In this case, with no DUT, verification requires an excellent set of test cases and scenarios that check and examine the interaction between master and slave components in more depth. This project shall provide a complete verification environment in UVM to guarantee the performance, functionality, and interoperability of the UVCs in accordance with the APB framework. The outcome will be the delivery of a reliable verification methodology that will improve the integrity of the APB implementation within SoC designs.

1.3 Aims and Objectives

1.3.1 Aims

The main objective of this project is to create a comprehensive verification environment for the AMBA APB protocol, with an emphasis on the verification of master and slave UVCs using UVM. The objective of this project is to ensure that these components work correctly according to the APB specification, allowing reliable communication in SoC designs.

1.3.2 Objectives

- **Develop a Verification Environment:** Create a robust UVM-based verification environment tailored for the master and slave UVCs of the APB protocol, enabling systematic testing and validation.
- **Design Comprehensive Test Cases:** Develop a wide range of test cases that cover various scenarios, including corner cases, to thoroughly evaluate the functionality, performance, and compliance of the master and slave UVCs.
- **Verify Protocol Compliance:** Ensure that both the master and slave components adhere to the APB protocol specifications, addressing aspects such as signal timing, data integrity, and control signal interactions.
- **Enhance Reusability:** Create reusable verification components and methodologies that can be applied to future projects involving APB or similar protocols.

Chapter 2

UVM Framework of UVCs

2.1 Why UVM?

The Universal Verification Methodology (UVM) was selected as the foundation for our APB VIP verification environment due to its numerous advantages:

- **Reusability:** The development of reusable verification components (UVCs), including scoreboards, monitors, agents, and drivers, is encouraged by UVM. By permitting the reuse of components across several verification projects, this drastically cuts down on development time and effort.
- **Modularity:** UVM's hierarchical structure enables the modular design of verification environments, making them easier to understand, maintain, and extend.
- **Constrained-Random Verification (CRV):** With UVM's support for CRV, a large number of test cases with different input stimuli can be generated automatically. This aids in enhancing test coverage and locating possible problems

with the design.

- **Phase-Based Execution:** A systematic method to verification is offered by UVM's phase-based execution model, which guarantees that all required tasks are completed in the right order.
- **Transaction-Level Modeling (TLM):** UVM improves simulation performance and lowers verification complexity by using TLM to represent the communication between various verification components at a higher level of abstraction.
- **Factory Mechanism:** UVM's factory mechanism allows for dynamic object creation and configuration, enabling flexible and configurable verification environments.

2.2 Key UVM Concepts

2.2.1 Transaction-Level Modeling (TLM):

TLM is a modeling methodology that allows for the communication between verification components at a higher level of abstraction than traditional RTL-level modeling. UVM utilizes TLM to model the exchange of transactions between different components, such as agents, drivers, and monitors.

2.2.2 Phases:

Every testbench component understands the phase idea and is developed from uvm_component. Every component passes through a predetermined set of phases, and until every com-

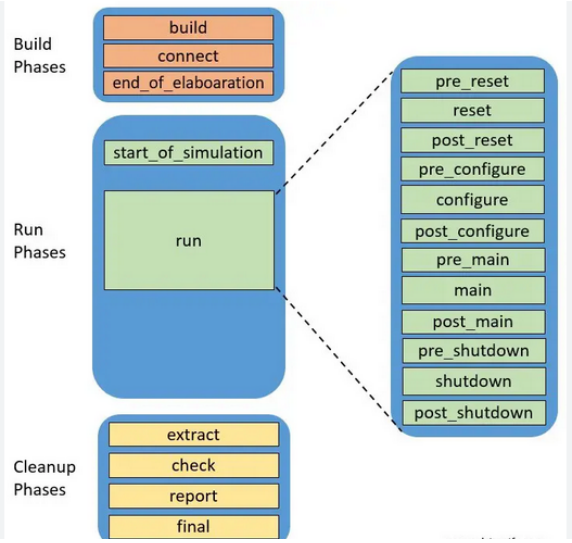

Figure 2.1: Phases

ponent has completed its execution in the current phase, it cannot move on to the next. Thus, UVM phases serve as a synchronizing mechanism throughout a simulation's life cycle.

Classes derived from uvm_component can carry out helpful tasks in the callback phase function since phases are defined as callbacks. Functions are techniques that don't use simulation time, and tasks are techniques that do. Three categories can be used to classify all phases:

- Build time phases
- Run time phases
- Clean-Up phases

2.2.3 FIFOs:

UVM buffers data between various components using FIFOs (First-In-First-Out). They can be applied to enhance simulation performance and decouple the time of data production and consumption.

2.2.4 Factory:

At runtime, objects can be dynamically created and configured thanks to UVM's factory system. This makes it possible to easily customize components and provide flexibility in the design of verification environments.

2.2.5 Configuration Database (ConfigDB):

Configuration parameters are stored and retrieved centrally in the ConfigDB. The number of transactions to be generated, the particular test cases to be run, and the amount of debug information to be printed are just a few of the configuration options available for the verification environment.

Chapter 3

UVM Testing Methodologies

This chapter explores the various testing methodologies employed within the UVM framework to ensure the comprehensive verification. By effectively combining these methodologies, we aim to achieve high levels of functional coverage and performance analysis.

- **Single UVC Verification:** A single UVC can be used to confirm a particular functional block or design component. It contains all of the testbench elements required to carry out focused verification, including the driver, monitor, and scoreboard. This method works best for straightforward designs or when highlighting particular elements of a design.

- **Multi-channel UVCs Verification:** Specialized UVCs known as multi-channel UVCs are able to manage several instances of the same protocol. When confirming designs with several interfaces of the same kind, this is especially helpful. We can decrease the number of UVCs needed by utilizing a single multi-channel UVC, which will optimize the testbench and increase efficiency.

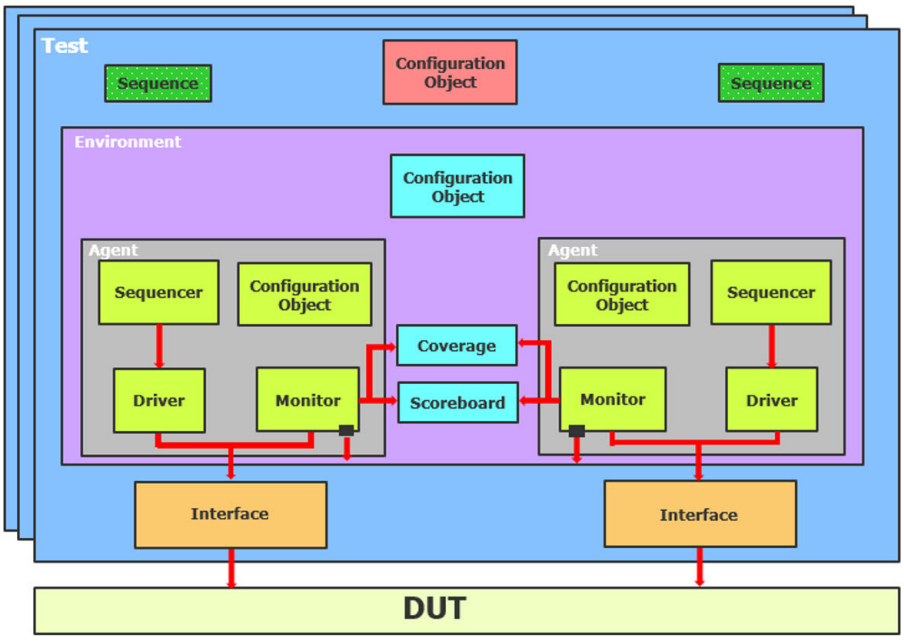

Figure 3.1: Multiple UVC's

Multi-channel UVCs are flexible enough to accommodate a range of design situations since they may be set up to manage varying numbers of channels. However, having many UVCs has a number of benefits. The verification environment can be made more easily comprehensible, maintainable, and reusable by modularizing it. Furthermore, they can be carried out concurrently, which speeds up the verification process. Additionally, UVCs can be reused for other verification projects, which saves time and effort during development. To manage the rising verification effort, more UVCs can be added as the design becomes more sophisticated.

- **Multi-channel sequences Verification:** In a multi-channel UVC, multi-channel sequences are an effective way to create complicated test scenarios

involving numerous channels. They make it possible to generate traffic on many channels in unison, mimicking real-world usage scenarios and revealing possible problems that single-channel testing would miss.

- By executing sequences sequentially, we can precisely control the timing of phase raise and drop events on each channel.
- This ensures that the different channels are synchronized and that the test cases accurately reflect the desired behavior.

Chapter 4

Verification Plan For APB VIP

Sr. No.	Test Name	Objective	Test Description	Result
1	Reset Test	Verify that the slave is reset.	Verify that clears memory using zeros placement	Passed
2	Write Test	Verify that a single write transaction is successful.	Assert PSEL, PWRITE, and drive PADDR, PWDATA.	Passed
3	Read Test	Verify a single APB read operation.	Read a single data value from the memory.	Passed
4	Read After Write	Verify data integrity through a write followed by a read.	Write a single data value to the memory and then read the value.	Passed
5	Bulk Read After Write	Verify multiple writes followed by multiple read transactions.	Write a group of data values to the memory and then read the values.	Passed
6	Back-to-Back Writes	Check if consecutive writes are handled without errors.	Perform multiple consecutive writes without idle states.	Passed
7	Back-to-Back Reads	Verify continuous read transactions.	Perform multiple consecutive reads.	Passed
8	Idle Cycle Insertion	Insert idle cycles between APB transactions.	Delay the next transaction to test idle state behavior.	Passed
9	Directed Slave Selection	Assert/deassert PSEL and ensure no misbehavior.	Write or read the transaction to different slave by selecting the Psel	Under Implementation
10	Randomized Slave Selection	Random assert/deassert PSEL and ensure no misbehavior.	Write or read the transaction to different slave by selecting the Psel	Under Implementation
11	Randomized Burst Length	Randomize the number of consecutive transfers.	Perform a random number of APB transactions.	Passed
12	Address Misalignment	Write to misaligned addresses.	Write the transaction to illegal address	Passed
13	Data Corruption Test	Corrupt data during transfer.	Perform a transaction that has an illegal data value.	Passed
14	Multiple Write Bursts	Verify burst write transfers.	Write consecutively incrementing the address for a specified number of transactions	Passed
15	Multiple Read Bursts	Verify burst read transfers.	Read consecutively incrementing the address for a specified number of transactions	Passed
16	Write transaction with delay cycles	Insert wait cycles between the access state and the transfer state.	The wait cycles are dependent on the pready signal that is coming from the slave.	Passed
17	Read the transaction with delay cycles	Insert wait cycles between the access state and the transfer state.	The wait cycles are dependent on the pready signal that is coming from the slave.	Passed

Sr. No.	Test Name	Test Type	Coverage Goals	Features	Signal under test
1	Reset Test	Directed	Reset the slave	Reset	Reset signal
2	Write Test	Randomized	Cover single write.	Write Transaction	PSEL, PWRITE, PWDATA, PADDR, PENABLE, PREADY
3	Read Test	Randomized	Cover single read.	Read Transaction	PSEL, PADDR, PRDATA, PENABLE, PREADY
4	Read After Write	Randomized	Cover single write and signal read	Data Integrity	PSEL, PWDATA, PADDR, PRDATA, PWRITE, PENABLE, PREADY
5	Bulk Read After Write	Randomized	Cover multiple writes and reads	Data Integrity	PSEL, PWDATA, PADDR, PRDATA, PWRITE, PENABLE, PREADY
6	Back-to-Back Writes	Randomized	Cover consecutive writes	Back-to-Back Write	PSEL, PWRITE, PWDATA, PADDR, PENABLE, PREADY
7	Back-to-Back Reads	Randomized	Cover consecutive reads	Back-to-Back Read	PSEL, PRDATA, PWRITE, PADDR, PENABLE, PREADY
8	Idle Cycle Insertion	Randomized	Cover the read and write transaction	handle idle states when there are delays between APB transactions.	PSEL, PWDATA, PADDR, PRDATA, PWRITE, PENABLE, PREADY
9	Directed Slave Selection	Directed	Cover Psel	Check multiple slave	PSEL, PWDATA, PADDR, PRDATA, PWRITE, PENABLE, PREADY
10	Randomized Slave Selection	Randomized	Cover Psel	Check multiple slave	PSEL, PWDATA, PADDR, PRDATA, PWRITE, PENABLE, PREADY
11	Randomized Burst Length	Randomized.	Cover the read and write transaction	Random number of read write transaction	PSEL, PWDATA, PADDR, PRDATA, PWRITE, PENABLE, PREADY
12	Address Misalignment	Directed	Cover all misaligned addresses.	Address Alignment	PADDR, PWRITE, PRDATA
13	Data Corruption Test	Directed	Cover data corruption at all positions.	Data Integrity on Corruption	PWDATA, PRDATA
14	Multiple Write Bursts	Randomized.	Cover all burst lengths.	Write Burst Handling	PSEL, PWRITE, PWDATA, PADDR, PENABLE, PREADY
15	Multiple Read Bursts	Randomized	Cover all burst lengths.	Read Burst Handling	PSEL, PWRITE, PRDATA, PADDR, PENABLE, PREADY
16	Write transaction with delay cycles	Randomized	Cover single write transaction	Write Transaction with wait cycles	PSEL, PWRITE, PWDATA, PADDR, PENABLE, PREADY
17	Read the transaction with delay cycles	Randomized	Cover single read transaction	Read Transaction Initiation with wait cycles	PSEL, PWRITE, PRDATA, PADDR, PENABLE, PREADY

Sr. No.	Test Name	Assertions
1	Reset Test	assert property (PSEL) |=> (PENABLE === 1); // Ensure PENABLE is high after PSEl is selected.
2	Write Test	assert property (PSEL) |=> (PENABLE && PWRITE) ##1 (PREADY); // Ensure PREADY is high after PENABLE is selected when there are no wait cycles.
3	Read Test	assert property (PSEL) |=> (PENABLE && !PWRITE) ##1 (PREADY); // Ensure PREADY is high after PENABLE is selected when there are no wait cycles.
4	Read After Write	assert property (PSEL) |=> (PENABLE && PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY) ##1 (PENABLE && !PWRITE && PADDR) ##[1:$] (PREADY); // Ensure write transcation is executed then followd by the read transcation.
5	Bulk Read After Write	assert property (PSEL) |=> (PENABLE && PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY) ##1 (PENABLE && !PWRITE && PADDR) ##[1:$] (PREADY); // Ensure write transcation is executed then followd by the read transcation.
6	Back-to-Back Writes	assert property (PSEL) |=> (PENABLE && PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY) ##1 (PENABLE && PWRITE && PADDR) ##[1:$] (PREADY); // Ensure write transcation is executed then followd by the write transcation.
7	Back-to-Back Reads	assert property (PSEL) |=> (PENABLE && !PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY) ##1 (PENABLE && !PWRITE && PADDR) ##[1:$] (PREADY); // Ensure read transcation is executed then followd by the read transcation.
8	Idle Cycle Insertion	assert property (PSEL) |=> [(PENABLE && PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && !PREADY) ##1][*n] ; // Ensure write transcation is executed then followd by the read transcation.
9	Directed Slave Selection	assert property (PSEL==1) |=> (PENABLE && PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY && PSEL == 2) ##1 (PENABLE && PWRITE && PADDR) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY && PSEL == 3) ##1 (PENABLE && PWRITE && PADDR) ##[1:$] (PREADY);; // Ensure write transcation is executed on one slave and other transcation on another slave
10	Randomized Slave Selection	assert property (PSEL==[*1:3]) |=> (PENABLE && PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY && PSEL == 2) ##1 (PENABLE && PWRITE && PADDR) ##[1:$] (PREADY) ##1 (!PENABLE && PREADY && PSEL == 3) ##1 (PENABLE && PWRITE && PADDR) ##[1:$] (PREADY);; // Ensure write transcation is executed on one slave and other transcation on another slave
11	Randomized Burst Length	assert property (PSEL) |=> [(PENABLE && PWRITE && PADDR && PWDATA) ##[1:$] (PREADY) ##1 (!PENABLE && !PREADY) ##1][*n] ; // Ensure write transcation is executed then followd by the read transcation.
12	Address Misalignment	assert property (PSEL) |=> (PENABLE && (!PWRITE || PWRITE) && PADDR == 'hx) ##1 (PREADY);
13	Data Corruption Test	assert property (PSEL) |=> (PENABLE && (!PWRITE || PWRITE) && PWDATA == 'hx) ##1 (PREADY);
14	Multiple Write Bursts	assert property (PSEL) |=> (PENABLE && PWRITE) ##[1:$] (PREADY) ##1 (!PENABLE) ##1 (PENABLE && PWRITE) ##[1:$]; // Ensure PREADY is high after PENABLE is selected when there are random wait cycles.
15	Multiple Read Bursts	assert property (PSEL) |=> (PENABLE && !PWRITE) ##[1:$] (PREADY) ##1 (!PENABLE) ##1 (PENABLE && !PWRITE) ##[1:$]; // Ensure PREADY is high after PENABLE is selected when there are random wait cycles.
16	Write transaction with delay cycles	assert property (PSEL) |=> (PENABLE && PWRITE) ##[1:$] (PREADY); // Ensure PREADY is high after PENABLE is selected when there are random wait cycles.
17	Read the transaction with delay cycles	assert property (PSEL) |=> (PENABLE && !PWRITE) ##[1:$] (PREADY); // Ensure PREADY is high after PENABLE is selected when there are random wait cycles.

Chapter 5

Development of UVC's

5.1 Sequence Item

We must create two transcatoin classes for each of the two UVCs we are developing since the slave input and output signals differ from the master input and output signals. For instance, the slave UVC receives the signals from the penable and psel, which are output in the master transaction class. Likewise, the slave's availability serves as the master's contribution as well. The master sequence generates the psel, pwdata and paddr, while the slave the sequence generates the pready signal.

5.2 Sequencer

Data transactions are created as class objects by a sequencer and sent to the driver for execution. Since the uvm_sequencer basic class has all the functionality needed to enable communication between a sequence and a driver, it is advised to expand it. The sorts of request and response items that the sequencer can handle parameterize the base class.

5.3 Driver

As we have modified the driver so we have to do the handshake between both the master and the slave. In master driver, first it sets the psel to the virtual interface psel, then on the next clock cycle the penable is asserted, and then it waits for the pready signal which is comming from the slave UVC. After receiving the pready signal, the master completes the transaction by sending or receiving the data on the next clock cycle.

Similarly, the counterpart is done in the slave driver. First we wait for the psel signal, and then on the next clock cycle it gets the penable signal. After receiving the penable signal, the slave can wait a random number of cycles before generating the pready signal for the master driver.

5.4 Monitor

The monitor is used to observe the values from the interface side. In our case, the master monitor is receiving the slave transaction and the slave monitor is receiving the master transaction.

5.5 Agent

By instantiating and linking the Sequencer, Driver, and Monitor via TLM interfaces, an agent combines them into a single entity.

5.6 Environment

Several reusable verification components are included in a UVM environment, which also specifies the default configuration for each component based on application requirements.

5.7 Scoreboard

A UVM Scoreboard is a verification component that stores expected results and compares them to actual results captured from the Device Under Test (DUT) via Transaction Level Modeling (TLM) Analysis Ports. But in our case, as we do not have the dut, we would be taking the actual packet from the slave UVC instead of the dut. Similarly, we would take the expected packet from the master UVC and then compare it with the actual packet.

In our scoreboard, we have declared two decl imp macros because the transaction can come from two UVC's. One from the master and the other from the slave. So, there would be two write tasks, one for the master write and the other for the slave write. When the master monitor sends the transaction to the scoreboard, then the master write task is executed in which it first writes the transaction to the queue and then it checks that the pwrite is high. If pwrite is high, it writes the pwdata to the array. Otherwise, ignores the read transaction. Similarly, when the slave sends a transaction to the scoreboard, it executes the slave write task in which it first pops out the transaction for the queue and then checks for the pwrite to be low. If pwrite is low, it compares the prdata with the data that is stored in the memory.

5.8 Test Bench

In our testbench class, we have instantiated master and slave UVCs as well as scoreboard and multichannel sequencer. The TLM analysis port conections are made in the connect phase of environment class, along with connecting the individual sequencers with the multichannel sequencer.

5.9 Test Library

We make a test library file, in which we include the demo base test in which we add the basic necessary constructs, for example, setting the drain time in run phase. In addition, we add multiple test classes against each sequence that we made in the sequence class.

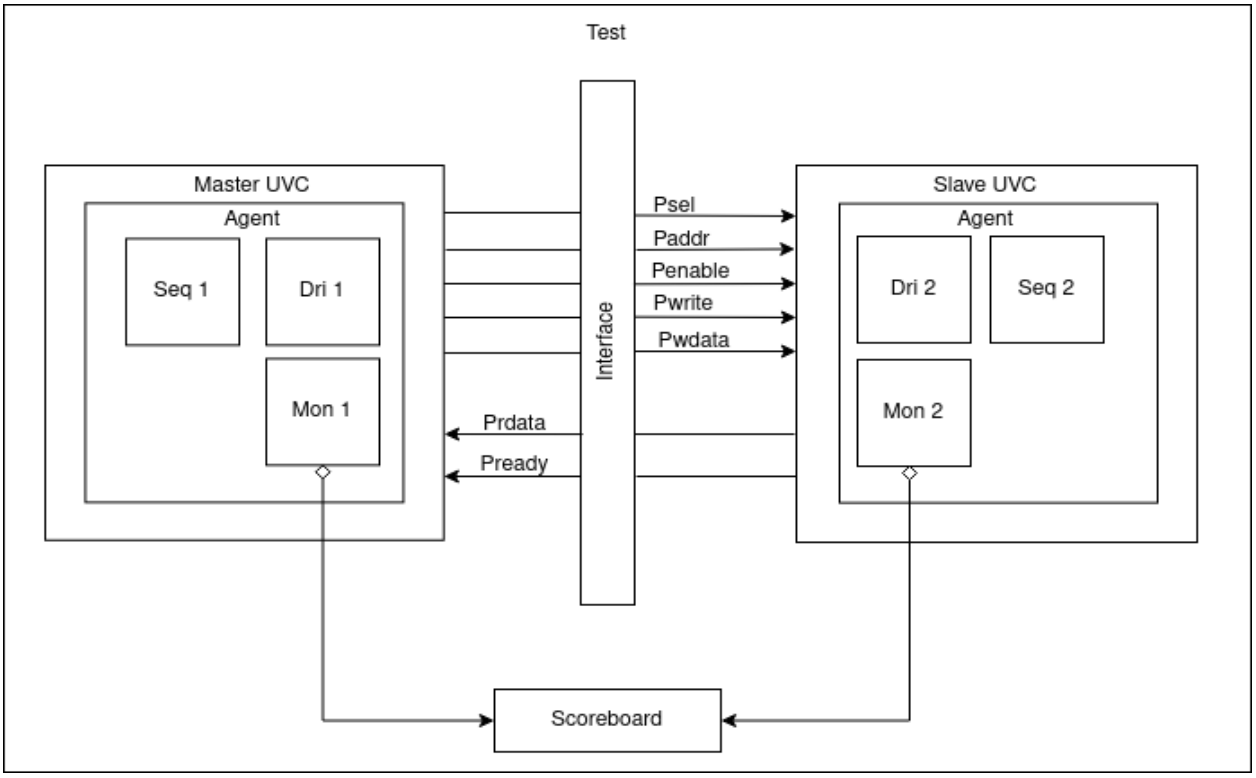

Figure 5.1: Integrated Test Bench of APB

Chapter 6

Assertion and Coverage

6.1 Assertion

Assertions are a powerful verification technique used to formally specify and verify the intended behavior of a design. They serve as guidelines or restrictions that the design must follow. By adding claims to the design, we are able to:

- **Early Detection of Errors:** Early specification problems can be found with assertions, which speeds up debugging and rectification.
- **Improved Design Quality:** They help in ensuring that there are no unexpected behaviors and that the design satisfies its functional requirements.
- **Enhanced Verification Coverage:** Functional coverage can be tracked using assertions, guaranteeing that every facet of the design has been extensively tested.
- **Reusability:** Reusing assertions in various verification settings saves time and effort during development.

- **Accelerated Regression Testing:** Regression testing can be strengthened with assertions, allowing for the quicker detection of flaws and regressions.

6.1.1 Types of Assertion

There are three kinds of assertions supported by SVA.

- **Immediate Assertion:** Immediate assertions are a type of assertion that are executed immediately, similar to a procedural statement. They are evaluated in the context of the current simulation time and can be used to check the validity of a condition at a specific point in time.

- **Concurrent Assertion:** Concurrent assertions, on the other hand, are designed to verify timing-related properties of a design. They operate on clock edges, allowing for the specification of complex sequences of events that occur over multiple clock cycles.

- **Deferred Assertion:** In order to solve the problem of false triggers brought on by signal glitches, deferred assertions are an improved form of immediate assertions. Deferred assertions wait until the end of the simulation timestep to evaluate expressions, making sure that all signals have stabilized before doing so, in contrast to immediate assertions, which analyze expressions as soon as they are encountered. This delay lessens the possibility of false positives brought on by erratic signal levels.

To ensure the valid sequence of state transitions in APB protocol, we have used four assertions that are used to verify the correct behavior of control signals for

various tests that have been performed. These assertions monitor the states of control signals in between multiple cycles for each transaction that is performed through the sequences run by both the drivers of both master and slave UVCs. The assertion are given below.

Assertion 1: *penable_high_after_one_cycle_when_psel_high;*

```
@(posedge clk) disable iff (!rstn)
  ($rose(psel) && (paddr !== 'x) && (pwrite ? (pwdata !== 'x &&
  pwdata !== 'z) : 1) && (psel == 1))|=> (penable == 1) [*1:$];
endproperty
```

Explanation: This assertion checks that penable must be high after one cycle of when the psel is asserted.

Assertion 2: *pready_high_after_N_clockcycles_of_penable_is_asserted*;

```
@(posedge clk) disable iff (!rstn)((penable)||($past(penable)))|->
( ##[0:$] (pready == 1));
endproperty
```

Explanation: This assertion checks that pready must be high after the random number of cycles after the penable becomes high. The pready signal is generated by the slave, So it all depends on slave whether to accept a new transaction or not.

Assertion 3: *prdata_is_X_when_pwrite_high*;

```
@(posedge clk) disable iff (!rstn) ((pwrite) && (pready))
    |-> (prdata === 'hx);
  endproperty
```

Explanation: This assertion checks that prdata must be x when the pwrite is high. Because when the transaction is of write then the prdata is invalid.

Assertion 4: *prdata_valid_when_write_low*;

```
@(posedge clk) disable iff (!rstn) ((!pwrite) && (pready))
|-> (prdata !== 'hx);
  endproperty
```

Explanation: This assertion checks that prdata must be valid when the pwrite is low. Because when the transaction is of read then the prdata is valid.

6.2 Coverage

Functional coverage quantifies how thoroughly a group of tests has tested a design's features and functionalities. Evaluating how well a regression test suite has covered certain features is very helpful in constrained random verification (CRV).

In an APB protocol, the pwdata, prdata, and the paddr are important in terms of coverage. These signals are included in our coverage goals and substantial coverage is ensured by running multiple sequences. Specific sequences are used to cover the whole range of addressable slave memory space, which is referred by the paddr signal.

Furthermore, randomized values of prdata and pwdata using exhaustive sequences are used to achieve sufficient coverage of these signals.

Chapter 7

Results

Now, it's time to see the results of what we have implemented. The following two figures are of the assertion whether it is passed or failed. In our case all the assertions had been passed.

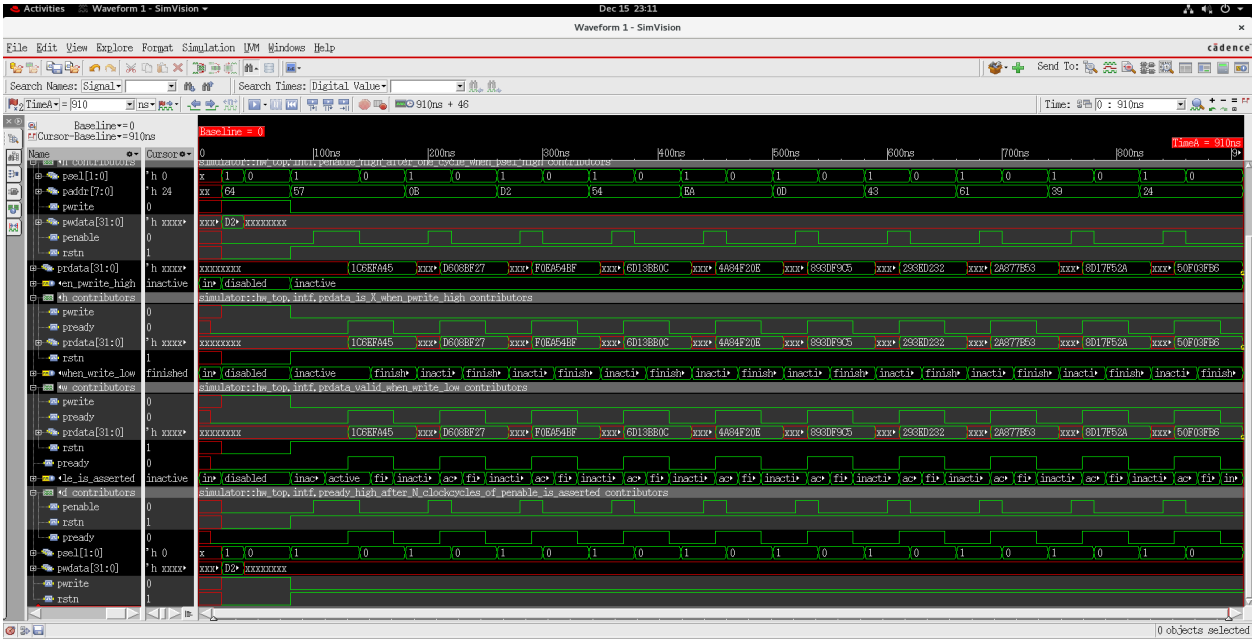

Figure 7.1: Assertions Passed

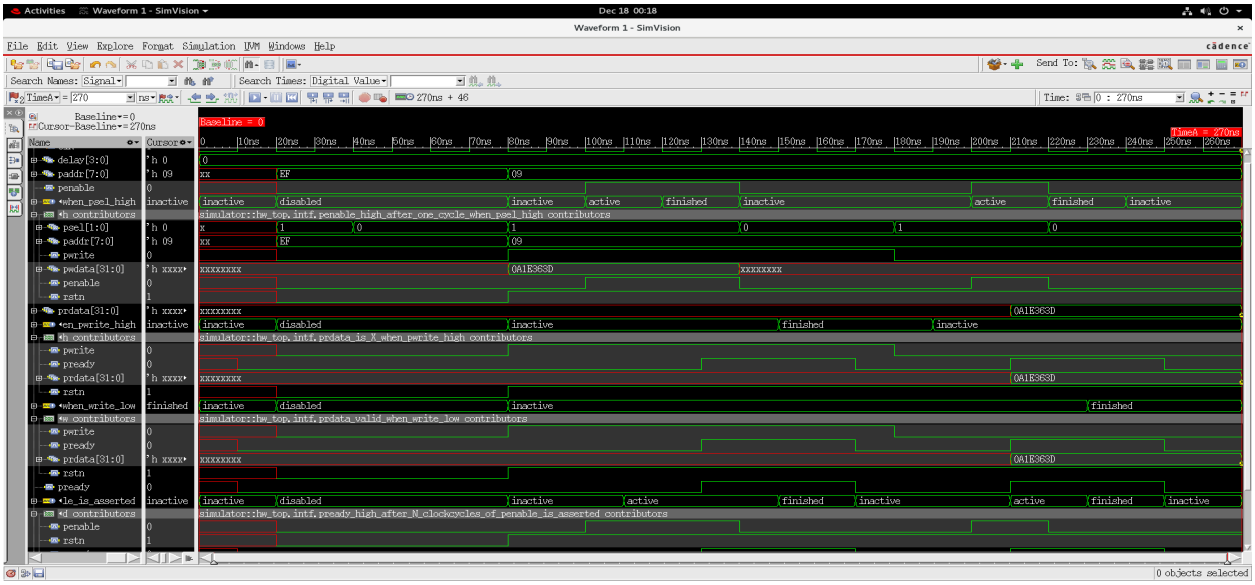

Figure 7.2: Assertion Passed

In the scoreboard, when packets arrive from both the master monitor and the slave monitor, they are compared, and a count is incremented based on whether the packets match or not. The master monitor captured a total of 21 packets, while the slave monitor captured 20 packets, resulting in a total of 40 packets received at the scoreboard. The missing packet corresponds to a reset transaction, which was

```
UVM_INFO ../sv/apb_scoreboard.sv(95) @ 3310: uvm_test_top.tb.apb_module_env_handle.apb_scoreboard_handle [apb_scoreboard] Report: Scoreboard Packet Stats: Received: 40
UVM_INFO ../sv/apb_scoreboard.sv(96) @ 3310: uvm_test_top.tb.apb_module_env_handle.apb_scoreboard_handle [apb_scoreboard] Report: Scoreboard Packet Stats: Correct: 20
UVM_INFO ../sv/apb_scoreboard.sv(97) @ 3310: uvm_test_top.tb.apb_module_env_handle.apb_scoreboard_handle [apb_scoreboard] Report: Scoreboard Packet Stats: Bad: 0
UVM_INFO ../APB_slave/sv/apb_slave_monitor.sv(193) @ 3310: uvm_test_top.tb.slave_uvc.agent.monitor [apb_slave_monitor] Report: Slave APB Monitor Collected 41 Packets

--- UVM Report catcher Summary ---
```

Figure 7.3: Scoreboard Result

not forwarded to the scoreboard but was still collected at the master monitor. As a result, the master monitor recorded 41 transactions. However, only 20 packets are

considered as passed because the scoreboard checks only for read transactions, while write transactions are ignored.

Now, for the coverage, the threshold had been achieved. The coverage report

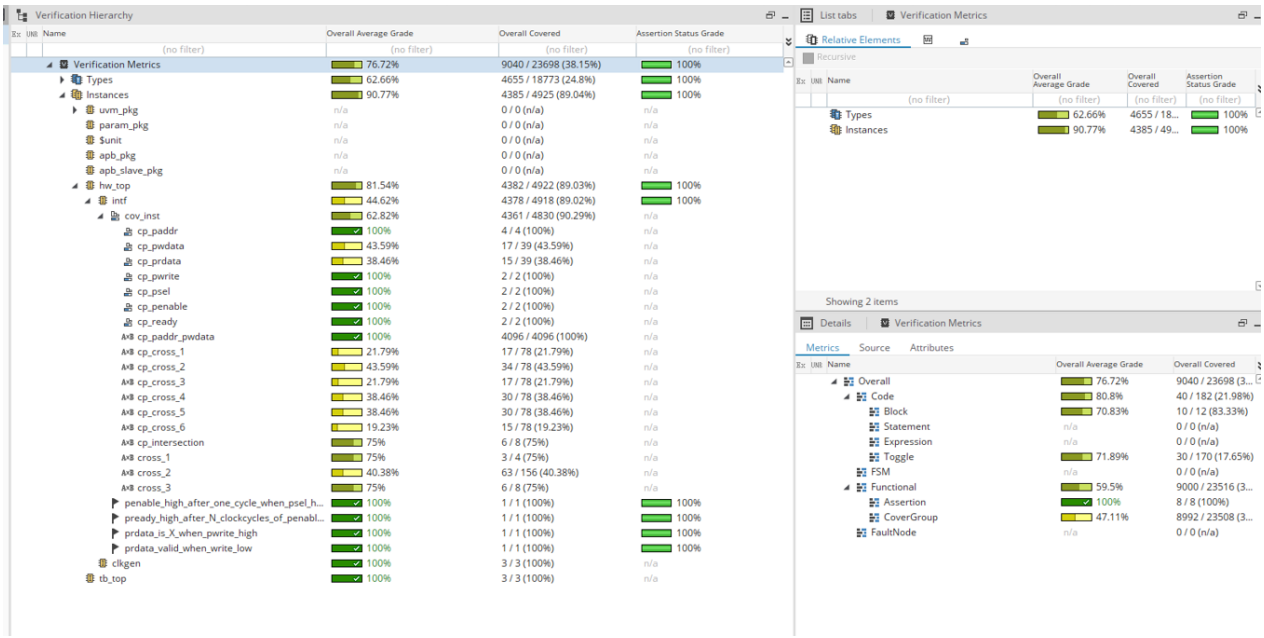

Figure 7.4: Coverage

provides insight into the overall verification progress and areas of the design that are exercised during simulation.

- **Overall Coverage Metrics**
  - **Overall Average Grade: 76.72%** This percentage reflects the total coverage achieved across all verification metrics, including code and functional coverage. There is still room for improvement to reach full coverage.
  - **Overall Covered: 9040 / 23698 (38.15%)** This indicates that 38.15% of all the coverable elements have been exercised, with 9040 items covered

out of 23698.

- **Code Coverage**
  - **Block Coverage: 40 / 182 (21.98%)** Indicates that only 22% of the basic blocks in the design code have been exercised. Further testing may be required to target untested blocks.
  - **Toggle Coverage: 71.89%)** Indicates that around 72% of all possible signal transitions (high-to-low and low-to-high) have been toggled during simulation.
- **Functional Coverage**
  - **Overall Grade: 59.5%** Functional coverage is lagging behind, with only 59.5% of the specified functional scenarios exercised.
  - **Assertions: 100% (8/8)** All assertions defined in the testbench or design have passed successfully. This indicates that the design meets the functional checks defined through assertions.

Chapter 8

Conclusion

By developing this VIP we have successfully verified the master UVC as well as slave UVC of APB protocol by executing exhaustive sequences and monitoring the assertions' status as well as functional coverage. This VIP can be utilized to verify both master and slave DUTs. The integration of optional pready in our slave UVC is also a new feature that is absent in most APB VIPs. This makes our VIP more flexible and reusable as compared to the currenlty available APB VIPs.

References

- www.chipverify.com/tutorials/uvm.
- www.chipverify.com/verification/verification-plan.
- https://developer.arm.com/documentation/ihi0024/latest
- https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture.
- https://verificationforall.wordpress.com/apb-protocol.
- http://dx.doi.org/10.1088/1757-899X/1084/1/012050.