Verification of APB Protocol and Integrating Tool for Repeaters

A Thesis submitted in partial fulfillment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

Vaibhav Anand

602262027

Under Supervision of

Dr. Harpreet Vohra

Assistant Professor- III

ELECTRONICS AND COMMUNICATION ENGINEERING

DEPARTMENT THAPAR INSTITUTE OF ENGINEERING &

TECHNOLOGY
(A DEEMED TO BE UNIVERSITY), PATIALA,
PUNJAB

JULY 2024

INTEL INDIA PRIVATE LTD

No 23-56, SRR Campus, Outer Ring Road, Devarabeesanahalli,

Varthur Hobli, Bellandur, Bengaluru, Karnataka, 560103., India

CERTIFICATE

Date:19th July,2024

This is to certify that **Vaibhav Anand**, a student of M. Tech (VLSI Design), Thapar Institute of Engineering & Technology, Patiala, pursuing his internship program of 12 months (June 2023 to June 2024) duration **at** INTEL INDIA PRIVATE LTD**, Bangalore.** His title of dissertation is **"Verification of APB Protocol and Integrating Tool for Repeaters".**

Mr. Sravan Chand Jyothi
GPU Design & Verification Engineer
INTEL India Pvt. Ltd.

Regd. Office:
Intel Technology India Private Limited
23-56P, Outer Ring Road,
Devarabeesanahalli, Varthur Hobli website: www.intel.in
Bellandur Post
Bangalore 560 103, India
CIN-U85110KA1997PTC021606

Tel: +91-80-2605 3000
Fax: +91-80-2605 6190

To Whomsoever It May Concern

WWID: 12204901

Employee Name: Vaibhav Anand

Internship Dates: 19-06-2023 to 14-06-2024

The letter is to confirm the mentioned above has undergone internship at Intel Technology India Private Limited.

We wish you all the best for your future assignments.

Yours Sincerely

Sumana Dey Srivastava

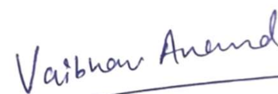
Date:

Place: **Bangalore**

DECLARATION

I, **Vaibhav Anand** hereby declare that the work presented in this project entitled **"Verification of APB protocol and Integrating tool for Repeaters"** in partial fulfilment of the requirement for the award of degree of **Master of Technology (VLSI Design)** submitted at **Electronics & Communication department,** Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under the supervision of **Dr. Harpreet Vohra (Assistant Professor, ECED, Thapar Institute of Engineering & Technology)** from June 2023 to June 2024. The matter presented in this has not been submitted in part or full to any other university or institute for the award of any other degree.

Date: 19th July,2024



Vaibhav Anand

(602262027)

Mr. Sravan Chand Jyothi
GPU Design & Verification Engineer
INTEL India Pvt. Ltd.

Date: 19th July, 2024

Dr. Harpreet Vohra
Assistant Professor- III
Department of Electronics and Communication Engineering
Thapar Institute of Engineering & Technology, Patiala

Date: 19th July, 2024

ACKNOWLEDGEMENT

The Task could not be completed without acknowledging to those who guided and supported continuously to make our efforts successful. Taking this opportunity, I express my deepest gratitude and respect to my supervisor, **Dr. Harpreet Vohra,** Assistant Professor III, Department of Electronics & Communication Engineering, Thapar Institute of Engineering & Technology for his guidance and encouragement throughout this project.

I express my deep gratitude and thanks to **Mr. Ormson T Brian** (Manager Graphics Processing Unit, INTEL), **Mr. Sravan Chand Jyothi** (GPU Design & Verification Engineer, INTEL) for his valuable advice and suggestions with unwavering support throughout the project. Many thanks to the members of team, for their valuable guidance during the project which helped me understand the technical aspects of the project. I would also like to thank **Dr. Kulbir Singh**, Head of Department, Electronics & Communication Engineering, Thapar Institute of Engineering & Technology for giving such an opportunity to do internship at INTEL INDIA Private Ltd and provide all kinds of support throughout the project. Finally, I thank my family, friends and colleagues for their timely help and valuable suggestions.

Vaibhav Anand
(602262027)

ABSTRACT

Grasping the AMBA Advanced Peripheral Bus (APB) protocols is essential for interfacing with low-bandwidth peripherals such as keyboards, UARTs, and timers, which do not demand the high-frequency operations necessary for memory access or processor requests. The early verification of APB protocol can be achieved by developing the APB Protocol with HDL (System Verilog) and conducting verification through Universal Verification Methodologies (UVM) requires a solid comprehension of UVM's built-in classes, functions, and methods. This knowledge facilitates the construction of an effective Test Bench structure.

Additionally, being a member of the Graphics Processing Units (GPUs) team that focuses on tools and methodologies, I recognize the critical role that Repeaters play in minimizing signal delay from the sender to the target. Many microprocessors developed at Intel necessitate the use of signal repeaters at the Full chip level to a certain extent. The implementation of Repeaters is essential for enhancing the speed of a net and for improving signal slope. It is crucial to comprehend the Repeater insertion process thoroughly. Repeaters Insertion is a tedious task by keeping in mind all the constraints like timing closure and minimization of the power but one of the problems we need to avoid is the formation of loops this caused once we done inserting repeater in one partition we cannot come back for the same partition for repeater Insertion. The loops can be avoided by tracing those nodes where loops can be formed, after tracing the nodes we can insert the repeaters by skipping those nodes keeping track of constraints.

Table of Contents

Sr. No.		Page No.
1.	**Introduction**	**1**
	1.1 APB Block Diagram	2
	1.2 List of AMBA APB Signals	3
2.	**Literature Survey**	**4**
	2.1 AMBA Bus	6
	2.2 Repeater Insertion Flow	7
3.	**Verification of APB Protocol**	**10**
	3.1 Introduction to UVM	10
	3.1.1 UVM Test Bench Architecture	11
	3.1.2 Objections	13
	3.1.3 UVM Phases	15
	3.1.4 Reporting Classes in UVM	18
	3.2 APB Test Bench Environment	19
	3.2.1 Algorithm for Read/Write Operations	19
	• Apb_Slave_Dut.sv	19
	• Apb_Tx.sv	20
	• Apb_Base_Seq.sv	20
	• Functional Sequence	21
	• Apb_Intf.sv	22
	• Apb_Sqr.sv	22
	• Apb_Drv.sv	23
	• Apb_Mon.sv	24
	• Apb_Cov.sv	25
	• Apb_Agent.sv	26
	• Apb_Env.sv	27
	• Apb_Test.sv	28
	• Top.sv	29
	3.3 Results and Discussions	30
	3.3.1 Log Results	31
	3.3.2 Waveforms	31
		35
4.	**Integrating Tool for Repeaters**	**36**
	4.1 Avoid Looping in Repeaters	37
5.	Conclusion and Future Scope	39
	References	40

List of Figures

1.1: AMBA Bus Architecture 2

1.2: APB Block Diagram 3

2.2: Flow of Repeaters insertion 8

3.1: Test Bench Architecture UVM 11

3.2: Task for Run_phase 12

3.3 Different phases in Test Bench 15

3.4: Flow of Phases in UVM 17

3.5: End of _elaboration phase (print topology) 17

3.6: Uvm_report_phase log 18

3.7: Design Algorithm of APB Protocol 19

3.8: TB Architecture of APB Protocol 20

3.9: Transaction Packet of APB Protocol 20

3.10: Base Sequence of APB Protocol 21

3.11: Functional Sequence of APB 22

3.12: Interface and Clocking Block of APB Protocol 22

3.13: Sequencer Class of APB Protocol 23

3.14: Sequence Mapping to Sequencer 23

3.15: Driver Class of APN Protocol 24

3.16: Monitor Class of APB Protocol 25

3.17: Coverage Class of APB Protocol 26

3.18: Agent Class of APB Protocol 27

3.19: Environment class of APB Protocol 28

3.20: Test Case Scenarios of APB protocol 29

3.21: Test Top of APB Protocol 30

3.22: Log result for single read/write cycle 31

3.23: Log result for single read/write cycle 32

3.24: Log result for multi read/write cycle 33

3.25: Log result for multi read/write cycle 34

3.26: Waveform for single read/write test 35

3.27: Waveform for multiple read/write test 35

List of Figures

4.1: Block Representation before Repeaters Insertion 36

4.2: Block Representation after Repeaters Insertion 36

4.3: Block Representation for Repeaters forming loop connection 37

4.4: Test Case to trace the looping Node 38

4.5: Trace of node forming Loop connection 38

List of Tables

2.1: Comparison between Bus Architecture **7**

1. Introduction

The remarkable progress in VLSI technology has made it possible to integrate millions of transistors onto a single chip, known as a System on Chip (SoC) [1]. The complete design and verification of the AMBA APB Protocol for SoC applications have been carried out. The AMBA Bus architecture includes several high-performance interfaces such as AHB, ASB, and AXI, which are utilized to connect with lower-performance buses like the APB. The APB is designed for low peripheral bandwidth applications and connects to slave devices such as UARTs, timers, keypads, and interrupt controllers. Traditionally, verification is conducted through simulation. However, as technological advancements increase the complexity of integrated circuits, the time required for verification has also grown [2]. SoC communication significantly influences system performance, power consumption, and time taken to market. Minimizing power dissipation is a primary goal for both system designers and researchers. The design of the Advanced Peripheral Bus regulator facilitates transactions between master and slave devices. The APB protocol synchronizes changes with the positive clock edge, enhancing the integration of APB peripherals into any design flow. Each transaction requires at least two cycles. The AMBA APB (Advanced Peripheral Bus) offers a highly flexible and configurable verification IP that can be easily incorporated into any SoC verification environment. The protocol is designed and implemented using the System Verilog Language, with verification conducted through a UVM Testbench [3].

In System-on-Chip (SoC) architectures utilizing buses, the system features one or more shared buses to which all modules connect, managing all system communications through bus interface protocols. The AMBA bus architecture comprises two primary components: the Advanced High-Performance Bus (AHB) or Advanced System Bus (ASB), and the Advanced Peripheral Bus (APB). Components requiring higher bandwidth, such as high-performance processors, interfaces with high data transmission rates, and direct memory access, typically utilize the AHB or ASB. In contrast, the AMBA APB serves as a low bandwidth and low-performance bus, suited for peripherals with lower transfer speed requirements. These peripherals connect via a bridge that adapts to their specific bandwidth needs. The APB, part of the AMBA 3 protocol suite, offers an economical interface that simplifies connections and reduces power consumption. However, it does not support pipelining, making it ideal for peripherals that do not necessitate the advanced performance of a pipelined bus interface. Integrating APB peripherals into design flows is streamlined, ensuring signal transitions synchronize with the clock's positive edge. The structure of the AMBA bus includes three main parts: the Advanced High-Performance Bus (AHB), Advanced System Bus (ASB), and

Advanced Peripheral Bus (APB), as shown in Figure.1.1. The AHB or ASB is identified as a high-performance bus offering greater bandwidth, facilitating connections to components like the High Bandwidth Memory Interface, High-Performance Processor, On-Chip High Bandwidth RAM, and DMA bus master. On the other hand, the AMBA APB bus is noted for its lower performance and bandwidth, apt for linking with peripheral devices that require reduced bandwidth, such as UARTs, Keypads, Timers, and PIO (Peripheral Input Output). A bridge serves as the connection between the high-performance AHB or ASB bus and the APB bus. Devices linked to the APB bus are designated as slaves to the APB, with the bridge operating as the master [4]. The bridge conveys transactions from the high-performance bus to the peripherals on the APB, enabling interaction between the bus of high-performance and the subsystem devices.

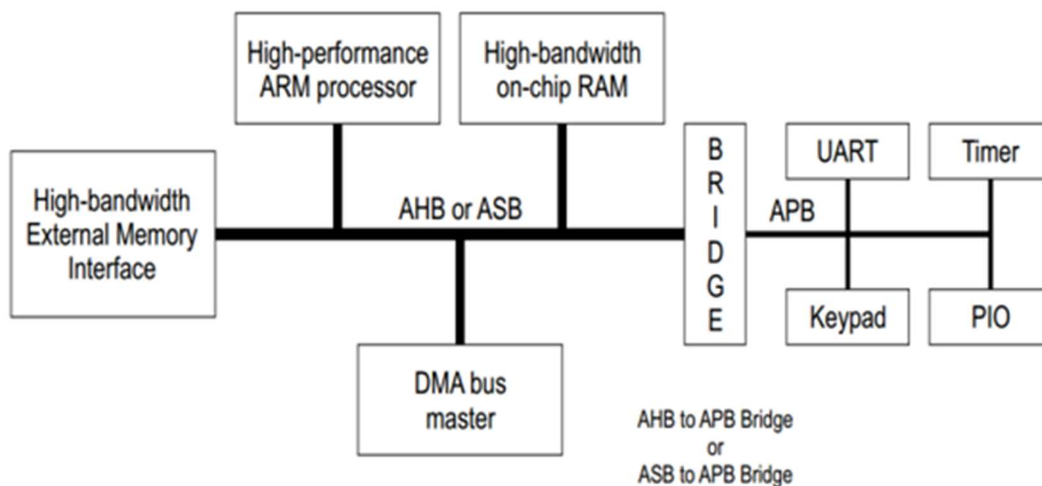


Figure1.1: AMBA Bus Architecture

1.1 APB BLOCK DIAGRAM: The fundamental block diagram representation of the AMBA APB, depicted in figure 1.1, outlines the primary signals according to the design specifications. In this setup, the APB slave receives several input signals, including PSEL, PENABLE, PWRITE (Address and Control), PResetn, and PCLK [1]. The PADDR signals are configured as 5 bits, while PWDATA are accepted as 16 bits input control signals from the bridge. In response, the slave delivers 16 bits of PRDATA as output.

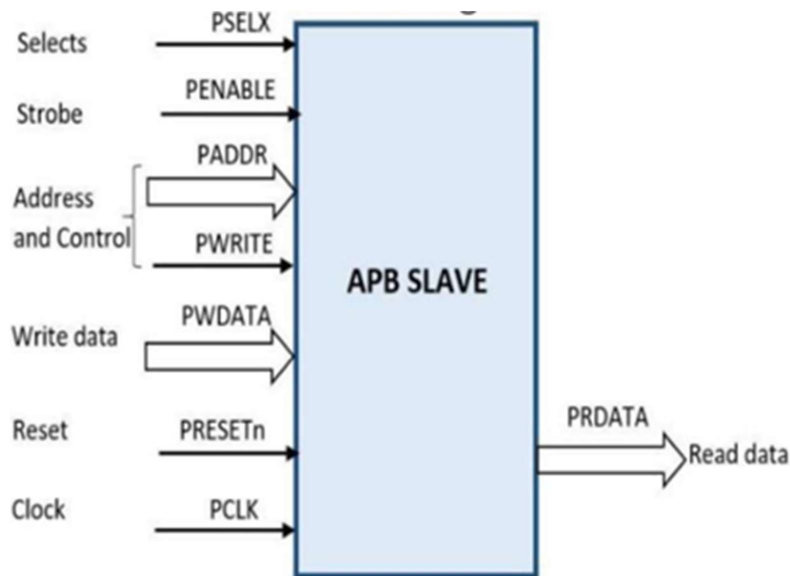


Figure 1.2: APB Block Diagram

1.2 List of AMBA APB Signals: In Figure 1.2: APB Block diagram shows us the IN/OUT signals through APB Slave is as follows:

- PCLK acts as the interface for the system bus slave, usually connected directly to the system's clock.
- PResetn – An asynchronous reset signal that is active when low.
- PADDR [5:0] - An address bus that can be up to 5 bits wide, facilitating communication from Master to Slave.
- PWDATA [7:0] - A write data bus capable of carrying up to 16 bits from the master to the slave.
- PRDATA [7:0] - The read data bus has a maximum width of 16 bits and transfers data from Slave to Master.
- PSELx: Each slave connected to the master is assigned a unique PSEL signal; PSELx serves as the slave select signal.
- PENABLE: This signal indicates the second and subsequent cycles of a transfer. The transfer's ACCESS phase commences once PENABLE is activated.
- PWRITE: The PWRITE signal denotes a write operation when HIGH and a read operation when LOW.
- PREADY: Utilized by the slave to introduce wait states into the transfer process. For instance, if the slave is not ready to complete the transaction, it will lower the PREADY signal, signaling the master to pause.

2. Literature survey

To enhance the reusability of the IP core, the AMBA on-chip bus architecture is extensively utilized as the interconnection standard for System on Chip (SOC) designs. Within the AMBA architecture, the Advanced High-performance Bus (AHB) is linked to an ARM control unit, a memory interface, digital signal processing (DSP), and the Advanced Peripheral Bus (APB), which connects to peripherals such as timers, UARTs, keypads, and Programmable Input/Output (PIO). A bridge serves as a crucial bus-to-bus interface, bridging the communication gap between the AHB and APB and preventing data loss during transfers. Consequently, advancements in technology, tools, and methodologies have necessitated improvements in the verification environment to accommodate these developments [4].

Recent advancements in VLSI technology have enabled the integration of millions of transistors onto a single chip, referred to as a System on Chip (SOC) [5]. The SOC utilizes the Advanced Microcontroller Bus Architecture (AMBA) as its on-chip bus protocol, with the Advanced Peripheral Bus (APB) being a crucial element of this architecture. This study explores the comprehensive design and verification of the AMBA-APB Protocol for SOC applications. The AMBA bus framework includes high-performance components such as AHB, ASB, and AXI, which interface with lower-performance buses like the APB. The APB is specifically designed for low peripheral bandwidth usage and facilitates connections to slave devices such as UARTs, timers, keypads, and interrupt controllers. Traditionally, verification has primarily relied on simulation. However, with the increasing complexity of integrated circuits, verification times have escalated. This [6] paper primarily focuses on designing the APB protocol using Verilog and verifying it using two languages: System Verilog and the Universal Verification Methodology (UVM).

As discussed in [7], the Advanced Peripheral Bus (APB) is tailored for low bandwidth and performance, aimed at linking peripherals like UARTs, keypads, timers, and other devices within the bus architecture. This paper aims to design and implement a bridge for the Advanced Microcontroller Bus Architecture - Advanced Peripheral Bus (AMBA APB) that optimizes system resource usage efficiently. In any digital sequential system design, managing the clock signal is critical. Clock skew, caused by variations in clock signal arrival times across different system parts, presents a significant challenge. To mitigate clock skew, one effective method involves employing a three-bit down ripple counter. This paper details the implementation of an APB Bridge incorporating a clock skew minimization technique, developed using Verilog HDL the Advanced Peripheral Bus (APB) is a component of the The AMBA suite of protocols

is designed for efficiency and simplicity, emphasizing reduced power consumption and interface complexity. Unlike the Advanced High-performance Bus (AHB), the Advanced Peripheral Bus (APB) is a non-pipelined protocol tailored for connecting peripherals with low bandwidth requirements to the System on Chip (SoC). Communication between Master and Slave devices occurs via a 32-bit architecture. In this setup, the Testbench acts as the Master APB, while the Design functions as the Slave APB. The Master initiates Peripheral Bus operations such as Write, Read, and Idle transactions, ensuring compatibility with multiple slaves. Slave responses are simulated through sequences, incorporating wait states signalled by a READY signal and error handling via an ERROR signal. This study's primary objective is to design and evaluate operational aspects of the APB protocol, validating data consistency between write and read operations for specific and randomly indexed addresses using random values. The methodology leverages Verilog and Verilog Testbench for synthesis, design implementation summary, and improving Testbench component reusability through task-based test cases.

Kiran Rawat and collaborators discussed RTL Implementation of the AMBA ASB APB protocol at the SoC level, focusing on synthesis and simulation of the AMBA ASB and APB interface. Their design provides RTL views of the ASB APB module at the SoC level, emphasizing optimized design efficiency based on power reports. Roopa. M and team designed low-bandwidth peripherals utilizing high-performance bus architecture. The APB controller manages transactions between the master and bus peripherals, with the APB protocol tailored for low-bandwidth slave devices. Transactions involve two bus cycles when a select line designates a specific slave. Chinghai Ma and colleagues developed an APB bridge translating AXI4.0-lite transactions into APB 4.0 transactions, facilitating interfaces between the high-performance AXI bus and the low-power APB domain. The design supports up to 16 APB peripherals with 32-bit AXI slave and APB master interfaces. Muhammad Hafeez and team created an IP core for efficient data transmission from a single slave, featuring an APB-SPI controller with adjustable data width and frequency. The SPI operates at a maximum frequency of 16 MHz for data transmission to and from a single slave. This section provides a comparative analysis of WISHBONE, AMBA, and Core Connect, with a primary focus on AMBA and Core Connect due to their widespread adoption in the SoC bus architecture domain. The comparison highlights their common goal of facilitating IP core connectivity, incorporating handshake protocols, and supporting variable data bus sizes.

2.1 AMBA Bus: Developed by ARM, the Advanced Microcontroller Bus Architecture (AMBA) serves as a foundational SoC bus with a standard specification. It comprises three distinct sub-buses: the Advanced High-Performance Bus (AHB), the Advanced System Bus (ASB), and the Advanced Peripheral Bus (APB). AMBA's structure is hierarchical, splitting into System-bus and Peripheral-bus segments connected via a bridge that buffers data and operations. The AHB and ASP bus cater to high-performance devices, posing integration challenges due to their overlapping device targets.

- **Advanced High-Performance Bus (AHB):** Designed for high-performance, high-frequency systems, the AHB acts as a backbone bus, supporting connections to processors, on-chip, and off-chip memories. It enhances bandwidth by accommodating multiple bus masters.
- **Advanced System Bus (ASB):** The inaugural AMBA bus generation, the ASB, allows for one or multiple masters, such as processors and test interfaces. It is commonly used for Direct Memory Access (DMA) or Digital Signal Processors (DSP) as bus masters, with external and internal memories typically serving as ASB slaves.
- **Advanced Peripheral Bus (APB):** Targeting low-power devices, the APB is optimized to minimize consumption of power and complexity of the interface. It functions as a local secondary bus within the AHB or ASB framework.

Core Connect Bus: IBM's Core Connect, an on-chip bus architecture, facilitates the processor for integration, subsystem, and peripheral processing units from various sources into a unified VLSI design. Its hierarchical structure is well-documented, offering specifications for each component, including the Processor Local Bus (PLB), On-chip Peripheral Bus (OPB), Device Control Register (DCR), Arbiter, and 64+ bit extensions. CoreConnect stands out for its comprehensive documentation and technical clarity, providing a structured framework for system components and their interconnections. The DCR bus connects components in a daisy-chain configuration, with the PLB linking all components.

While the basic operations of these buses are similar, the key distinctions lie in their feature sets.

WishBone Bus: Silicore Corporation developed the Wishbone bus architecture, is an open-source hardware specification tailored for digital circuit bus structures. This specification establishes a standardized method for interconnecting and facilitating communication among diverse modules and components within digital systems. By defining specific signals,

protocols, and operational conventions, Wishbone ensures seamless compatibility and interoperability across modules sourced from different vendors. This architecture is extensively utilized in the creation of System-on-Chip (SoC) designs and other digital systems that benefit from modular and adaptable interconnect solutions.

We can see the comparison between AMBA(APB), IBM (Core Connect) and Silicore Corporation (WishBone):

Table 2.1: Comparison between Bus Architecture

Sr.No.	Features	AMBA(APB)	Core Connect	WishBone
1.	Open Architecture	Yes	Yes	Yes
2.	Max. Data Bus Width	1024 bits	128 bits	64 bits
3.	Max. Address Width	32-64 bits	32 bits	64 bits
4.	Split Transactions	Yes	Yes	No
5.	Switch Bus	Yes	Yes	Yes
6.	Multiplexed	Yes	Yes	Yes
7.	RMW transfer	No	No	Yes

2.2 Repeater Insertion Flow: The repeaters can be categorized into two groups: those that alter the signal's functionality, such as latches or flip-flops, and those that do not, like buffers. While changes in functionality necessitate updates in the RTL to ensure correct logical behavior, the need for repeater insertion in cases without functional changes remains debatable [8]. Many modern system-on-chips feature hierarchical cores with multiple levels of design structure, incorporating older generation SoCs as embedded mega cores. This hierarchical arrangement imposes specific constraints on how tests are applied to both the "parent" cores and their "child" cores. The connections after the insertion of repeaters from child to parent is a critical task [9].

The strategy for incorporating repeaters has led to a hybrid method where some repeaters are inserted manually and others automatically. A decision-making diagram provided in the figure below guides the selection of the appropriate method for inserting a specific type of repeater:

Repeater Insertion Process: Regarding naming conventions, it's crucial to track a signal's journey through various repeaters to its destination[10]. This tracking helps in identifying not just the signal's initial source but also the repeaters it has passed through. The naming protocol employed in the Itanium™ processor follows the format: Signalname_[##F]C, where:

- "Signalname" represents the original signal from the source unit.

- "##" denotes the repeater station number.
- "F" indicates the repeater's function, with B for Buffer, I for Inverter, F for Flip-flop, and O for Other functions.
- "C" provides coloring information, reflecting timing phase details.
- "[##F]" is appended to signal names at each repeater station to maintain a record of the signal's path.

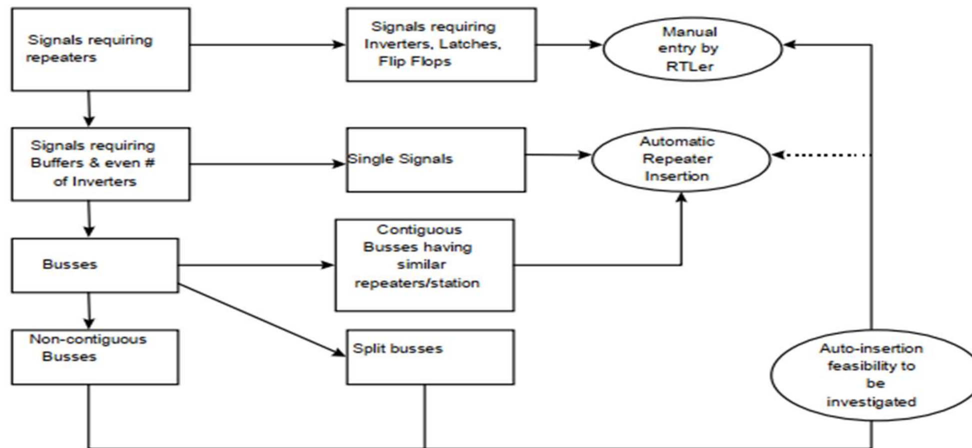


Figure 2.2: Flow of Repeaters insertion

It examines their performance, effectiveness, and integration within superconducting circuits. The study aims to optimize the design and functionality of these transmission lines to improve the overall efficiency and reliability of superconducting systems [11]. The paper addresses global signaling challenges in large-scale RSFQ (Rapid Single Flux Quantum) circuits. It presents solutions for efficient signal distribution and management, focusing on optimizing performance and scalability in high-density superconducting circuits. The work aims to enhance the reliability and functionality of large RSFQ systems [12]. The paper analyzes RF and crosstalk issues in copper and MLGNR interconnects with various repeaters in the sub-10nm technology regime, aiming to optimize signal integrity and performance in advanced semiconductor circuits [13]. The paper discusses repeater insertion strategies to reduce delay and power consumption in copper and carbon nanotube-based nano interconnects, focusing on performance improvements in advanced semiconductor technologies [14].

- Signals Integrity: The paper analyses performance and signal integrity of intercalation-doped MLVGNR interconnects, evaluating their effectiveness in maintaining signal quality and enhancing circuit performance [16]. It provides an analytical study of bundled

multi-walled carbon nanotube (MWCNT) and edged monolayer graphene nanoribbon (MLGNR) interconnects. It examines their effects on propagation delay and area in semiconductor circuits. The study aims to evaluate how these advanced interconnect materials impact performance and spatial efficiency, offering insights into their potential benefits for high-speed and compact electronic designs [17].

- Verification Methodology: It explores the rationale behind using UVM alongside System Verilog, highlighting how UVM complements System Verilog by managing verification complexities and responsibilities that go beyond the core language's capabilities [18]. The paper investigates crosstalk in coupled MLGNR interconnects with various repeater insertion techniques, aiming to optimize signal integrity and performance in advanced semiconductor interconnects [19]. A comprehensive review of Universal Verification Methodology (UVM) concepts for functional verification. It explores UVM's key principles, including its architecture, component hierarchy, and the role of sequences and agents in verification. The review highlights the methodology's advantages in improving testbench reuse, automation, and coverage. It also discusses UVM's impact on streamlining the verification process and enhancing the accuracy of functional verification in complex designs. This review serves as a valuable resource for understanding UVM's implementation and its benefits in modern electronic design verification [20].

3. Verification of APB Protocol

The early verification of APB protocol can be achieved by developing the APB Protocol with HDL (System Verilog) and conducting verification through Universal Verification Methodologies (UVM) requires a solid comprehension of UVM's built-in classes, functions, and methods. This knowledge facilitates the construction of an effective Test Bench structure.

3.1 Introduction to UVM (Universal Verification Methodology):

- UVM is an open-source methodology, UVM .sv files (also called library) are openly available. Users can view the code and use the code without any licensing.
- Developed in collaboration of Mentor Graphics, Synopsis and Cadence. UVM can be used with any of the simulation tools. UVM is essentially SV based classes.
- UVM has evolves in the following manner to create a verification methodology which works universally on any tool by using the pre-defined base classes.
- UVM provides base classes with the common functionality already implemented. These UVM based classes can be used in every TB development. Serves the time in developing test benches.
- Methodologies before UVM:
    - Reference Verification Methodology (Vera based) developed by Synopsis.
    - eRM: e Reference Methodology (Specman E based) developed by Cadence.
    - AVM: Advance Verification Methodology (SV based) developed by Mentor Graphics.
    - OVM: Open Verification Methodology (SV based)

Need for Methodology:

- Reduces the times taken to setup the TB, to develop the test case and debugging of test cases.
- Helps with reusability and modularity, that means we can develop TB for different design using the same predefined classes.
- There are lot of things common in all these testbenches:
    - For every interface in the design, we need to develop BFM, generator, monitor and coverage.
    - For every interface, we need to develop scenario and transactions class.
    - Basic functionality of TB component is same across test benches.
- Methodology brings uniformity into the complete verification flow; Any user can think of in functional verification and TB development.
    - Architecture of TB definition.

- Component coding of TB.
- Connecting TB components.
- Test case coding.
- Scenario development
- Printing messages.
- Checking the design outputs.
- Deciding when to end the test.
- Reporting test status.

3.1.1 UVM Test Bench Architecture:

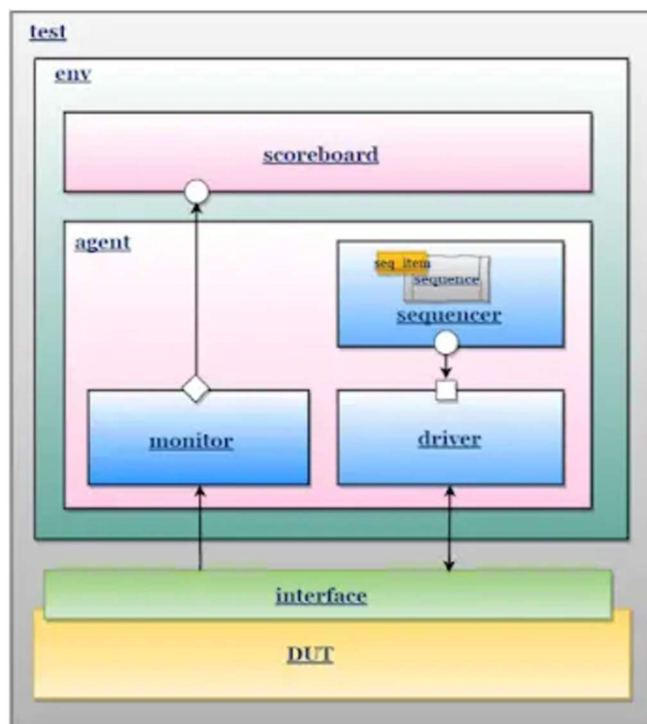


Figure 3.1: Test Bench Architecture UVM

Test or UVM ROOT: The Figure 3.1 show us the Test Bench Architecture of UVM:

- uvm_root is one of the base classes in UVM package.
- The beginning of UVM TB. It is referred as Program block in SV Test bench.
- Singleton class instantiated in global scope and has only one instance in entire TB "uvm_root top".
- Uvm root functionality is followed with user calling run test method in top module.
- "top.run_test" gets called when user calls the run test, whole TB functionality starts off.

- SV run_test method:

```
//program block
`include "environment.sv"
program test(intf i1);
   environment env;
   initial
     begin
       env = new(i1);
       env.run();
       end
endprogram
```

- UVM run_test method:

```
Pseudo code:
// Top block procedure top_block():
 begin
 run_test("apb_n_wr_n_rd_test");
 end
 // Main program
 begin
 top_block();
 end
```

What does "run_test" do?

- Setup testbench for current test case.
    - Call various common phase of the test bench one after the other.
    - Test name is instantiated as uvm_test_top.
- Uvm_root a variable called 'phase' (uvm phase "phase")
    - The phase instance is given to all the TB component as parameters of the uvm common phase.

```
//linking ofsequence and test is done in run phase
task run_phase(uvm_phase phase);
  apb_seq seq;
  seq=apb_seq::type_id::create("seq");
  phase.raise_objection(this); //raising the objection, do not finish the simulation unless my objection is dropped
  phase.phase_done.set_drain_time(this,100); //once the functionality is done, still run for 100 more units
  seq.start(env.agent.sqr); //run this seq on sqr present inside agent
  phase.drop_objection(this); // I am done woth my functionality, now you can end the simulation
  endtask
```

Figure 3.2: Task for Run_phase

Why is phase argument passed to methods?

- Root is the main component which needs to keep track of objections raised and objections dropped.
- It is required an instance through which it gets those raise/drop updates.
- The variable by name 'phase' which is passed to all the components and methods.
- Whenever we want to raise or drop the objection, it is done using phase variable.
- The raise/drop objection is used to decide when the simulation ended.
- If the phase is not given to an argument, how can component convey raise and drop objection information to the root.

3.1.2 Objections:

- Objection is a mechanism used by uvm_root to find the finish time of the simulation. Simulation completes if all the objections which are raised can be dropped.
- An objection can be raised for component to indicate that component functionality has begun.
    - phase. raise objection(this)
- An objection can be dropped for component to indicate that its functionality has completed.
    - phase. drop objection(this)
- Phase is coming from uvm_root, used to keep track of all objections raised and dropped.
- They can only be raised on a phase.
    - Uvm_phase phase defined in uvm_root and passed to all.
    - Component have phases: build_phase, run_phase, end_of_elaboration_phase.
        - function to create build phase by passing argument (uvm phase "phase");
        - task to create run phase by passing argument (uvm phase "phase");
- Objections give a finish to the simulation.
    - $finish results in abrupt end of simulation, other phases will not run.
    - Objections make sure that all the phases of UVM TB complete, only then $finish is called.
    - Hence, extract_phase, check_phase and report_phase also gets called even though drop objection is happening in the run_phase.
- A TB with no objection will cause simulation ends at zero time. Objection is a compulsory concept in UVM TB.

- If $finish called in top module at 50ns, and all the objections are dropped at 75ns, then simulation will end at 50ns since $finish has highest priority.
- Objections are never raised in driver, monitor and scoreboard, why?
    - Driver has forever loop implemented those blocks will never complete, any objection raised will never dropped. Hence, it will result in simulation hang scenario.

      Pseudo code:

      ```
      task run phase (uvm phase "phase");
      forever begin
      // Get the next item from the sequence
      Seq item port. Get next item (req);
       // Print the details of the request
      Request print ();
       // Drive the transaction on the memory testbench
       Drive transaction (req);
      Sequence item port item done();
      end
      end task
      ```
- Objection can be raised in test and sequence.
    - Objection raise/drop in "task run_phase":

      ```
      task run_phase (uvm phase "phase");
       apb seq "seq";
      // Create an instance of apb seq
       // Raise objection to indicate the start of functionality
       Phase raise objection (this);
       //Set drain time to continue simulation after functionality is complete
      phase done set drain time (this, 100);
       // Start the sequence 'seq' on the 'sqr' sequencer inside the agent seq start (env agent sqr);
      // Drop objection to indicate completion of functionality
       Phase drop objection (this);
      End task
      ```

    - Objection raise/drop in "sequence body task":

      Pseudo code:

        - ***Pre body method: Raises an objection in the starting phase obtained from get starting phase (). If the phase exists (phase! = null), it raises the objection***

using phase raise objection (this) and sets a drain time of 100 units using phase done set drain time (this, 100).

- ***Post body method: Drops an objection in the starting phase obtained from get starting phase (). If the phase exists (phase! = null), it drops the objection using phase drop objection (this).***

- What if raised objection is not dropped?
  - Simulation will hang and times out at 9200sec unless user has given UVM_TIMEOUT.
    - vsim work.top +UVM_TIMEOUT=1000, will end the simulation at 1000ns

3.1.3 UVM phases:

- UVM TB works in such a way that, calls from the user run test method in the top module.
- Top run test method gets called.
- Top is uvm_root instance.
- Which in turn calls all UVM common phases one after the another.

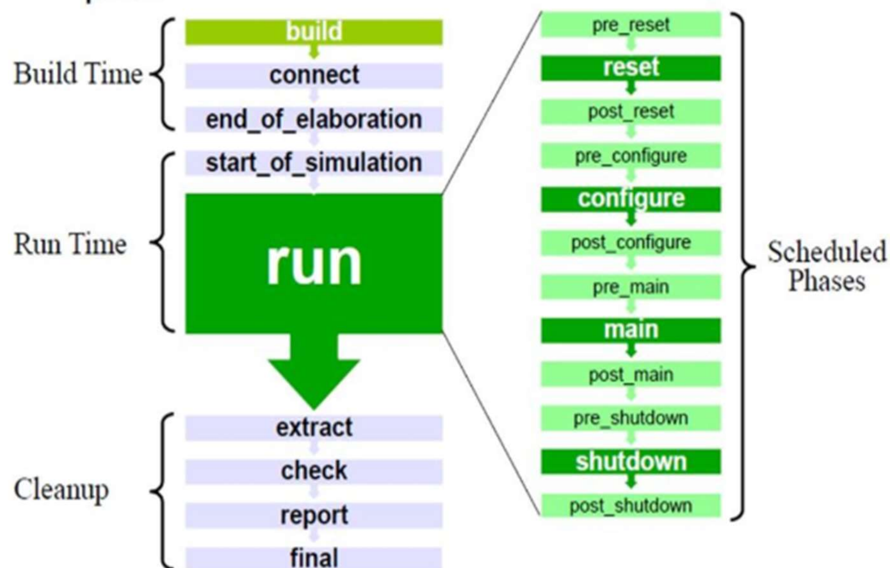


Figure 3.3 Different phases in Test Bench

- Build_phase:
  - Create the component immediately below the current component.

- While creating, name and parent is given as argument.
- Once all components are created, UVM TB knows the complete hierarchy.
- Connect_phase:
  - Some components to have children, there is a connection required between them.
  - Connect various test bench components and their children.
- UVM provide TLM (Transaction Level Modeling) to implement the component connections.
- End_of_elaboration_phase:
- Tool has figured out the structure by connecting the compiled object.
- Print the TB structure (uvm top print topology).
- Factory print method.
  - Start_of_simulation_phase:
- Preload the memories just before simulation starts.
- Set any design variables starts.
- Run_phase:
- Run the TB component functionality by calling the run method.
- Phases in run_phase:
- Reset phase: pre-reset (set control signals to be asserted); reset (reset DUT); post-reset (wait for DUT to be at a known state)
- Configure phase: preconfigure (setup/wait for conditions to configure DUT); configure (configure the DUT); post configure (Wait until the DUT (Device Under Test) is identified during the configuration phase).
- Main phase: pre-main (setup/wait conditions to start testing DUT); main (Test DUT); post-main (typically non-operating state).
- Shutdown phase: pre-shutdown (non-operating stage); shutdown (wait for data to be drained); post-shutdown (output should be initialized to X's and Z's).
  - Extract Phase:
- Extraction of data in verification environment from different points.
  - Check Phase:
- Monitor the verification environment for unforeseen conditions.
- Report Phase:
- reports result of the test.
- Final Phase: $finish executes.

- Log Result: In the Figure 3.4, Flow of phases in a sequential manner are listed. In the Figure 3.5 the end of elaboration phase is shown which tells us the topology of our TB architecture. In Figure 3.6, Report phase is printed in the log Result.

```
# apb_test::build_phase
# apb_env::build_phase
# apb_agent::build_phase
# apb_drv::build_phase
# apb_mon::build_phase
# apb_sqr::build_phase
# apb_agent::connect_phase
# apb_test::connect_phase
```

Figure 3.4: Flow of Phases in UVM

```
# -------------------------------------------------------------
# Name                      Type                    Size  Value
# -------------------------------------------------------------
# uvm_test_top              apb_test                -     @361
#   env                     apb_env                 -     @373
#     agent                 apb_agent               -     @381
#       cov                 apb_cov                 -     @546
#         analysis_imp      uvm_analysis_imp        -     @554
#       drv                 apb_drv                 -     @389
#         rsp_port          uvm_analysis_port       -     @406
#         seq_item_port     uvm_seq_item_pull_port  -     @397
#       mon                 apb_mon                 -     @538
#         ap_port           uvm_analysis_port       -     @564
#       sqr                 apb_sqr                 -     @415
#         rsp_export        uvm_analysis_export     -     @423
#         seq_item_export   uvm_seq_item_pull_imp   -     @529
#         arbitration_queue array                   0     -
#         lock_queue        array                   0     -
#         num_last_reqs     integral                32    'd1
#         num_last_rsps     integral                32    'd1
# -------------------------------------------------------------
#
```

Figure 3.5 End of _elaboration phase (print_topology)

```
# apb_test::report_phase
# UVM_INFO /usr/share/questa/qu
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :    6
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Questa UVM]     2
# [RNTST]     1
# [TEST_DONE]     1
# [UVM/RELNOTES]     1
# [UVMTOP]     1
```

Figure 3.6: Uvm_report_phase log

3.1.4 Reporting Classes in UVM**:** UVM Reporting provides Macros to report messages:

- The UVM provides macros for reporting messages with different levels of severity:
  - `uvm_info (string ID, string MSG, verbosity);
  - `uvm_error (string ID, string MSG);
  - `uvm_warning (string ID, string MSG);
  - `uvm_fatal (string ID, string MSG).
- When using UVM, users can manage message verbosity levels to control the amount of reported information. The uvm_info macro is optional for printing messages, while other macros (uvm_error, uvm_warning, uvm_fatal) are mandatory.
- For extensive messaging, set the verbosity level to UVM_DEBUG.
- For minimal messaging, use UVM_NONE.
- For moderate messaging, select UVM_MEDIUM.
- UVM offers three core classes for implementing message reporting:
  - uvm_report_object: Provides an interface to the UVM reporting system, allowing components to issue various messages during simulation. It handles message formatting and printing.
  - Reports include ID, severity, verbosity, and textual messages.
  - Reports may also include the filename and line number from which the message originated.
- Primary reporting methods in UVM include:
  - uvm_report_info/warning/error/fatal
  - These methods report ID string, severity, verbosity, and text message, optionally including the filename and line number.

3.2 APB Test Bench Environment:

3.2.1 ALGORITHM READ-WRITE OPERATION:

Parameter and Input Declaration

PARAMETER: WIDTH=16; ADDR_WIDTH=6;

INPUTS : clk_i, rst_i; [ADDR_WIDTH-1:0] addr_i; [WIDTH-1:0] wdata_i; wr_rd_i, valid_i;

Output Declaration

OUTPUTS : reg [WIDTH-1:0] rdata_o; reg ready_o;

reg [WIDTH-1:0] mem [DEPTH-1:0];

Read/Write Operation

Write data to a file.

Read data from a file.

always @(posedge clk_i)

Else condition:
If condition (valid_i == 1) :
ready_o = 1;
if condition(wr_rd_i == 1)
mem[addr_i] = wdata_i;
else condition :
ready_o = 0;

if condition (rst_i = 1):
rdata_o = 0;ready_o = 0;
mem[i] = 0;

Else condition:
If condition (valid_i == 1) :
ready_o = 1;
if condition(wr_rd_i == 0)
wdata_i = mem[addr_i] ;
else condition :
ready_o = 0;

Figure 3.7: Design Algorithm of APB

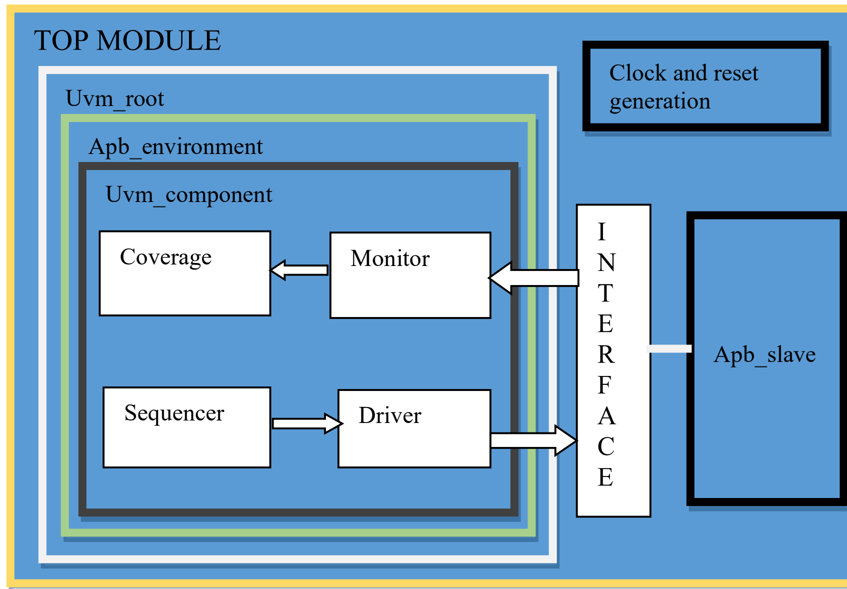


Figure 3.8: Test bench architecture of APB protocol

- Apb_Tx.sv: Transaction packet send from sequencer to driver which drives to the DUT as shown in Figure 3.9.

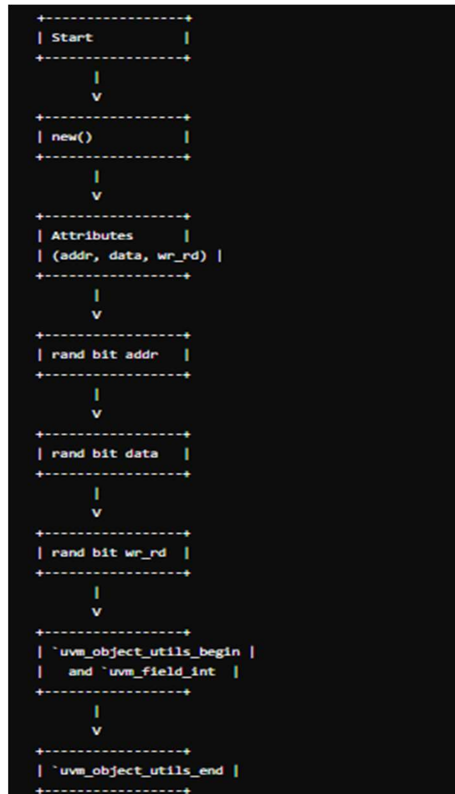


Figure 3.9: Transaction packet of APB protocol

- Apb_Base_Seq.sv: In UVM, sequence refers to sequence of transaction. Base sequence things are common to all the sequence as shown in Figure 3.11.
    - **Start**: Beginning point of the flow chart.
    - **new()**: Constructor of apb_base_seq class.
    - **pre_body()**: Task that performs initialization and prepares for sequence execution.
    - **get_starting_phase()**: Retrieves the starting phase of the sequence.
    - **raise_objection()**: Raises objection to the starting phase, allowing the sequence to proceed.
    - **phase_done.set_drain_time()**: Sets drain time for the phase, indicating how long it will wait before it can be completed.

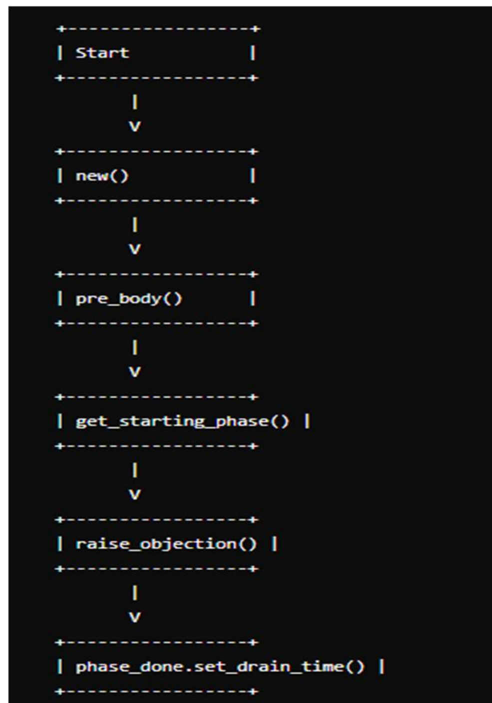


Figure 3.10: Base sequence of APB protocol

- Functional sequence derived from base sequence: the sequence which are different in functionality can be derived using the base sequence by using "extends" keyword as shown in figure 3.12.
    - **Start**: Beginning point of the flow chart.
    - **new()**: Constructor of apb_base_seq class.
    - **pre_body()**: Task that performs initialization and raises objection to the starting phase.
    - **get_starting_phase()**: Retrieves the starting phase of the sequence.
    - **raise_objection()**: Raises objection to the starting phase, indicating that the sequence requires the phase to continue simulation.
    - **phase_done.set_drain_time()**: Sets drain time for the phase.
    - **post_body()**: Task that drops objection to the starting phase after sequence execution.
    - **drop_objection()**: Drops objection to the phase, allowing it to complete and progress to the next phase.

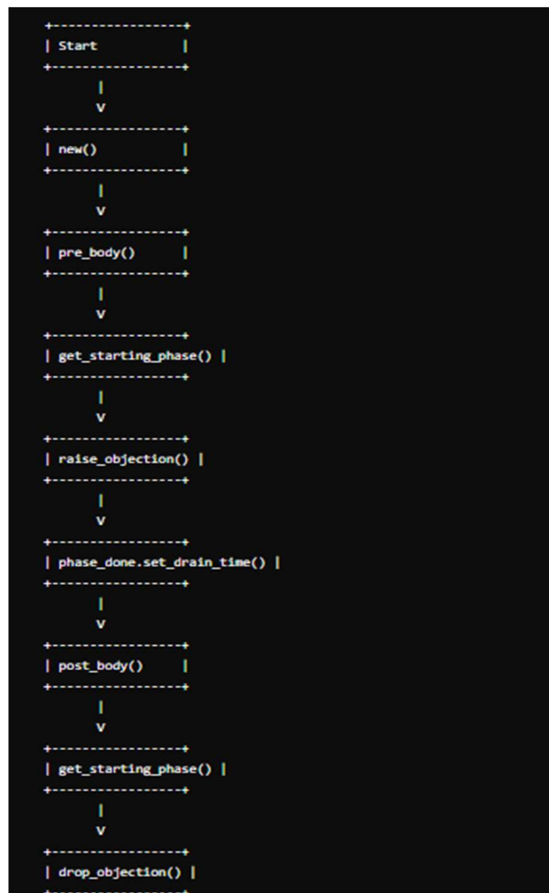


Figure 3.11: Functional Sequence of APB protocol

- Apb_Intf.sv: All the interfaces which are used by the DUT as well as Test Bench are declared in this file. The clocking blocks are declared to avoid the setup and hold time violations as shown in figure 3.13.

```
interface apb_intf (input bit clk_i, rst_i);
bit [`ADDR_WIDTH-1:0] addr_i;
bit [`WIDTH-1:0] wdata_i;
bit [`WIDTH-1:0] rdata_o;
bit wr_rd_i;
bit valid_i;
bit ready_o;


clocking bfm_cb@(posedge clk_i);
  default input #1 output #2;
  input rst_i, rdata_o, ready_o;
  output addr_i, wdata_i, wr_rd_i, valid_i;
endclocking

clocking mon_cb@(posedge clk_i);
  default input #1;
  input rst_i, rdata_o,ready_o;
  input addr_i, wdata_i, wr_rd_i, valid_i;
endclocking
endinterface
```

Figure 3.12: Interface and Clocking block for APB protocol

- Apb_Sqr.sv: Sequencer send the sequences to the DUT. The sequencer mapping from Sequencer to Drive, then Driver to DUT is shown in the below figure. 3.14

```
class apb_sqr extends uvm_sequencer#(apb_tx);
  `uvm_component_utils(apb_sqr)

  function new(string name="", uvm_component parent=null);
    super.new(name, parent);
  endfunction

  function void build_phase(uvm_phase phase);
    super.build_phase(phase); //calling build phase of the uvm
    $display("apb_sqr::build_phase");
  endfunction
endclass
```

Figure 3.13: Sequencer class of APB protocol

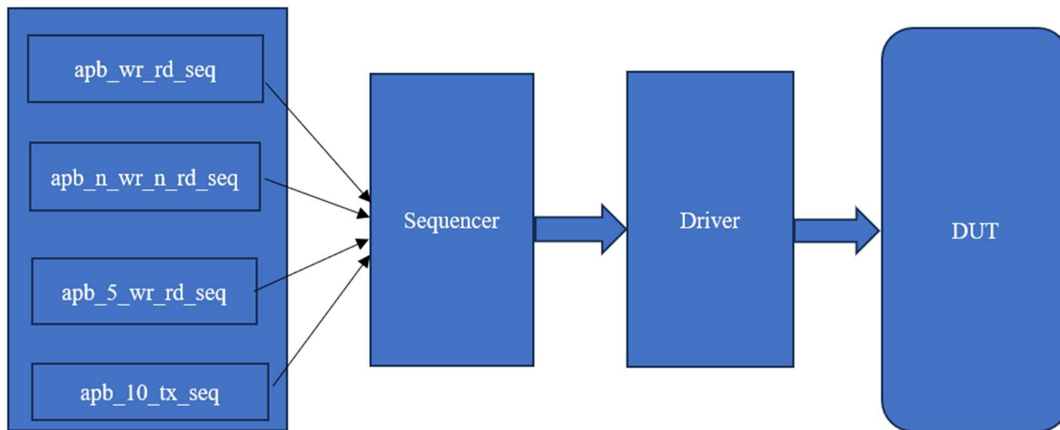


Figure 3.14: Sequence mapping to Sequencer

- Apb_Drv.sv: Creating a task of “run_phase” to get the item from “apb_seq” to drive it as shown in 3.16.
    - **Start**: Beginning point of the flow chart.
    - **build_phase()**: Initializes the UVM driver and reads the virtual interface (vif) from the UVM resource database.
    - **run_phase()**: Executes continuously during the run phase of the UVM framework.
    - **get_next_item()**: Retrieves the next transaction (req) from the sequence item port.
    - **drive_tx()**: Drives the transaction (tx) onto the virtual interface (vif).
    - **reset_inputs()**: Resets the virtual interface inputs after driving the transaction.
    - **item_done()**: Signals completion of processing for the current transaction.
    - **Repeat run_phase()**: Loop back to run_phase() to continue processing next transactions.

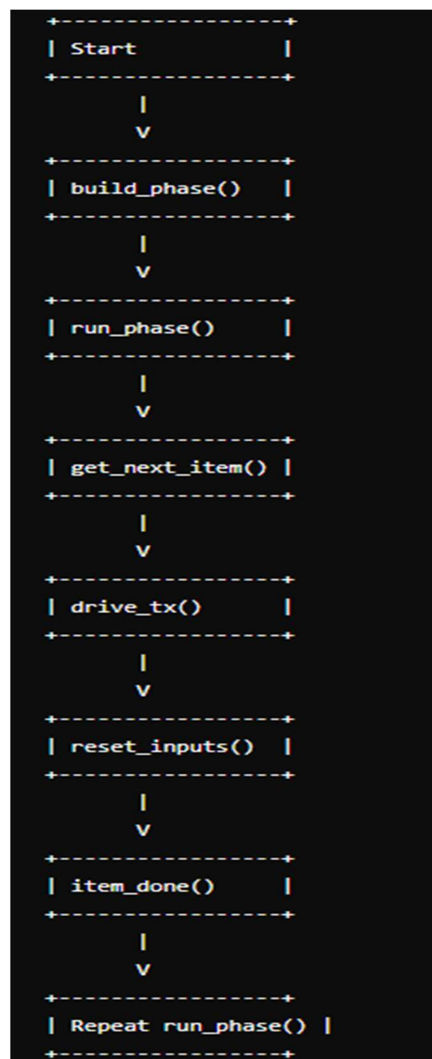


Figure 3.15: Driver class of APB protocol

- Apb_Mon.sv: The APB Monitor extends uvm_monitor base class and inherit its properties. The monitor declared its ports in the function "build_phase". The Analysis port "ap_port" is used for one-to-many connections which helps to send information to Scoreboards, Checker and Coverage for the analysis as shown in figure 3.17.

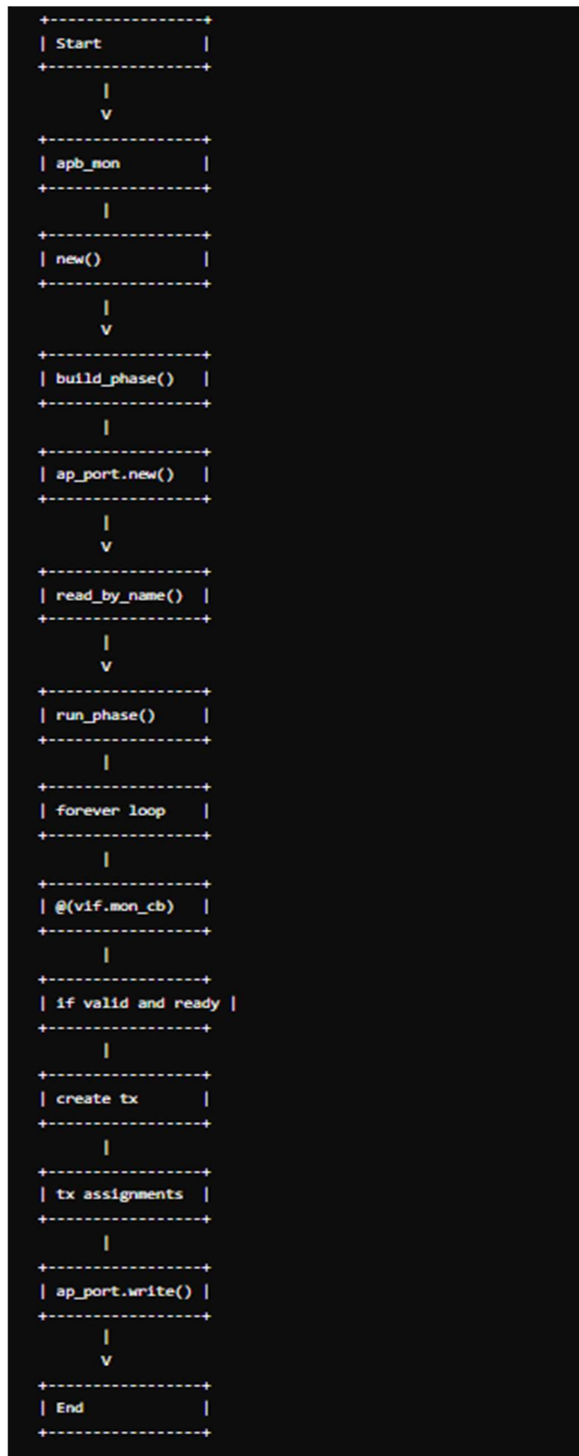


Figure 3.16: Monitor class of APB protocol

- Apb_Cov.sv: The “apb_cov” is subscribing to “apb_mon” whatever apb_mon transmits, “apb_cov” needs to get it, it needs to get apb_tx. The write method will be called by monitor, whenever it wants to give tx to the coverage class as shown in figure 3.18.
    - **Start**: Beginning point of the flowchart.
    - **apb_cov**: Represents the apb_cov coverage class.
    - **new()**: Constructor of apb_cov class.
    - **apb_cg=new()**: Allocates memory for the coverage group (apb_cg).
    - **$cast(tx, t)**: Casts the incoming transaction (t) to the type of tx.
    - **apb_cg.sample()**: Samples the coverage group (apb_cg).
    - **End**: Endpoint of the flowchart.

Figure 3.17: Coverage class of APB protocol

- Apb_Agent.sv: In the “apb_agent” we extend “uvm_agent” to inherit its properties, after that we declare handles for the driver, monitor, sequencer and coverage class. In build_phase we allocate the memory by creating objects and in “connect_phase” we connect all the components in the required manner.
    - **Start**: Beginning point of the flowchart.
    - **apb_agent**: Represents the apb_agent class.
    - **new()**: Constructor of apb_agent class.
    - **build_phase()**: Initializes the components within the agent.
    - **drv=new()**: Creates an instance of apb_drv driver.
    - **sqr=new()**: Creates an instance of apb_sqr sequencer.
    - **mon=new()**: Creates an instance of apb_mon monitor.
    - **cov=new()**: Creates an instance of apb_cov coverage.
    - **connect_phase()**: Establishes connections between components.

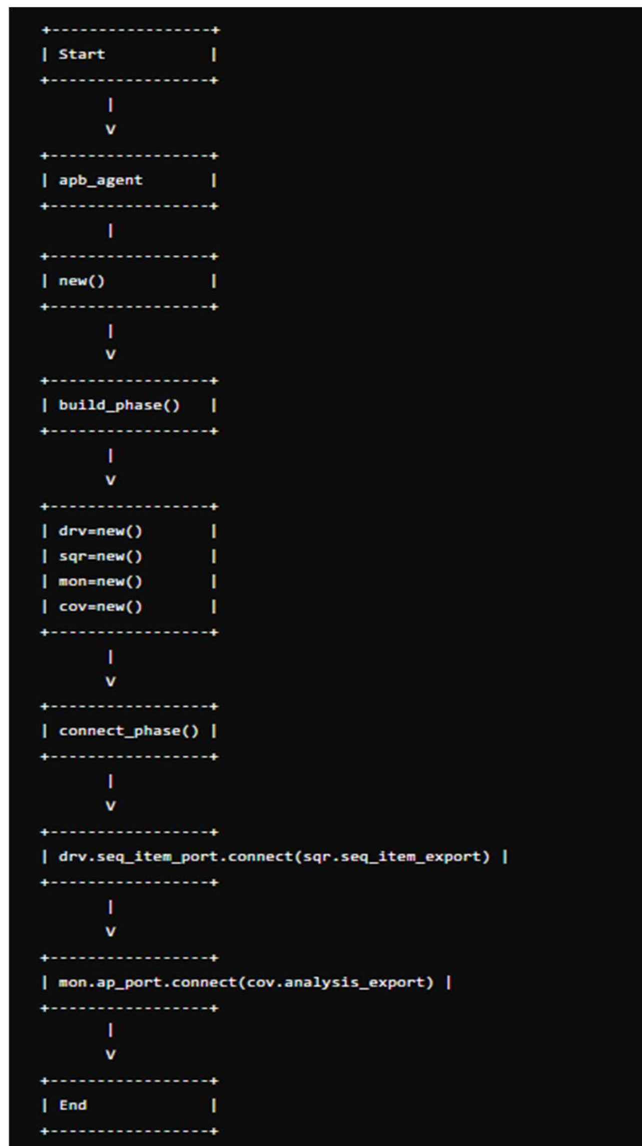


Figure 3.18: Agent class of APB protocol

- Apb_Env.sv: the environment file encapsulates the agent inside it. The “build_phase” create the agent inside the environment and “connect_phase” makes the connection between the two all these phases are inherited by the “uvm_env” base class as shown in 3.20 figure.
  - **Start**: Beginning point of the flowchart.
  - **apb_env**: Represents the apb_env class.
  - **new()**: Constructor of apb_env class.
  - **build_phase()**: Initializes the environment and creates an instance of the apb_agent.
  - **agent=new()**: Creates an instance of the apb_agent class.
  - **connect_phase()**: Placeholder for future connections or setups.
  - **End**: Endpoint of the flowchart.

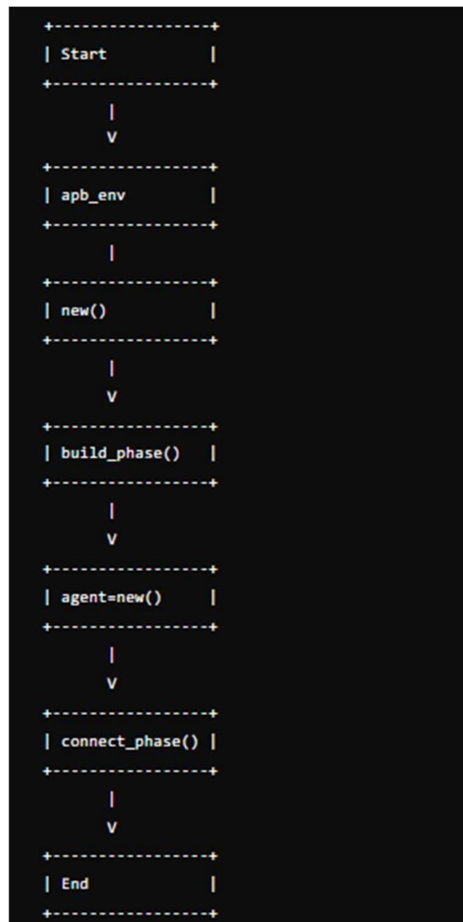


Figure 3.19: Environment class of APB protocol

- Apb_Test.sv
    - **Start**: Beginning point of the flowchart.
    - **apb_base_test**: Represents the apb_base_test class.
    - **new()**: Constructor of the apb_base_test class.
    - **build_phase()**: Initializes the environment.
    - **env=apb_env::type_id::create("env",this)**: Creates an instance of apb_env and assigns it to the env variable.
    - **end_of_elaboration_phase()**: Prints the testbench topology.
    - **uvm_top.print_topology()**: Outputs the hierarchical structure of the testbench.
    - **End**: Endpoint of the flowchart.

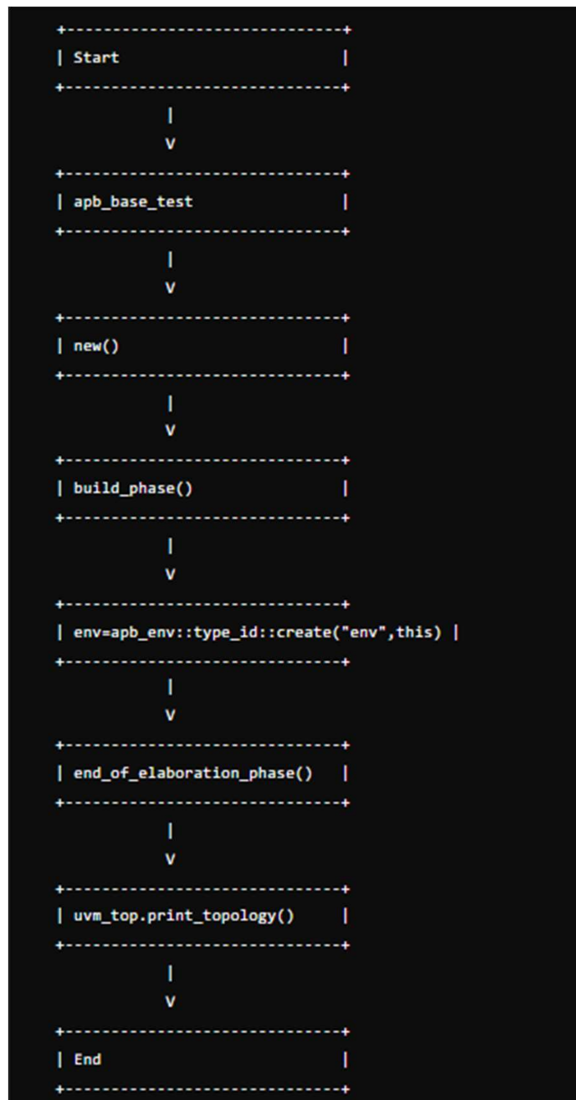


Figure 3.20: Test case scenarios of APB protocol

- Top.sv: The clock and reset are declared in the Top file, also which test to be run by the run_test method is declared here as shown in figure 3.22.

```
+----------------------+
| Start                |
+----------------------+
          |
          v
+----------------------+
| Initial Setup        |
+----------------------+
          |
+----------------------+
| clk_i = 0;           |
| forever #5 clk_i = ~clk_i; |
+----------------------+
          |
          v
+----------------------+
| rst_i = 1;           |
| repeat(2) @(posedge clk_i); |
| rst_i = 0;           |
+----------------------+
          |
          v
+----------------------+
| uvm_resource_db::set("GLOBAL", "VIF", pif, null); |
+----------------------+
          |
          v
+----------------------+
| apb_slave dut        |
| .clk_i(pif.clk_i)    |
| .rst_i(pif.rst_i)    |
| .addr_i(pif.addr_i)  |
| .wdata_i(pif.wdata_i)|
| .rdata_o(pif.rdata_o)|
| .wr_rd_i(pif.wr_rd_i)|
| .valid_i(pif.valid_i)|
| .ready_o(pif.ready_o)|
+----------------------+
          |
          v
+----------------------+
| run_test("apb_n_wr_n_rd_test"); |
+----------------------+
          |
          v
+----------------------+
| $dumpvars();         |
| $dumpfile("1.vcd");  |
+----------------------+
          |
          v
+----------------------+
| End                  |
+----------------------+
```

Figure 3.21: Test Top of APB protocol

3.3 Results and Discussions:

3.3.1 Log Results:

- For single read/write:

```
UVM_INFO @ 0: reporter [RNTST] Running test apb_wr_rd_test...
apb_test::build_phase
apb_env::build_phase
apb_agent::build_phase
apb_drv::build_phase
apb_mon::build_phase
apb_sqr::build_phase
apb_agent::connect_phase
apb_test::connect_phase
UVM_INFO /apps/vcsmx/vcs/U-2023.03-SP2//etc/uvm-1.2/src/base/uvm_root.svh(589) @ 0: reporter [UVMTOP] UVM testbench topology:
---------------------------------------------------------------
Name                      Type                    Size  Value
---------------------------------------------------------------
uvm_test_top              apb_wr_rd_test          -     @340
  env                     apb_env                 -     @353
    agent                 apb_agent               -     @362
      cov                 apb_cov                 -     @546
        analysis_imp      uvm_analysis_imp        -     @555
      drv                 apb_drv                 -     @371
        rsp_port          uvm_analysis_port       -     @390
        seq_item_port     uvm_seq_item_pull_port  -     @380
      mon                 apb_mon                 -     @537
        ap_port           uvm_analysis_port       -     @566
      sqr                 apb_sqr                 -     @400
        rsp_export        uvm_analysis_export     -     @409
        seq_item_export   uvm_seq_item_pull_imp   -     @527
        arbitration_queue array                   0     -
        lock_queue        array                   0     -
        num_last_reqs     integral                32    'd1
        num_last_rsps     integral                32    'd1
---------------------------------------------------------------
```

Figure 3.22: Log result for single read/write cycle

```
---------------------------------------------------------------------------
Name                          Type      Size  Value
---------------------------------------------------------------------------
req                           apb_tx    -     @604
  addr                        integral  6     'h24
  data                        integral  16    'hcbce
  wr_rd                       integral  1     'h1
  begin_time                  time      64    0
  depth                       int       32    'd2
  parent sequence (name)      string    3     seq
  parent sequence (full name) string    30    uvm_test_top.env.agent.sqr.seq
  sequencer                   string    26    uvm_test_top.env.agent.sqr
---------------------------------------------------------------------------
---------------------------------------------------------------------------
Name                          Type      Size  Value
---------------------------------------------------------------------------
req                           apb_tx    -     @638
  addr                        integral  6     'h24
  data                        integral  16    'hbcc4
  wr_rd                       integral  1     'h0
  begin_time                  time      64    45
  depth                       int       32    'd2
  parent sequence (name)      string    3     seq
  parent sequence (full name) string    30    uvm_test_top.env.agent.sqr.seq
  sequencer                   string    26    uvm_test_top.env.agent.sqr
---------------------------------------------------------------------------
UVM_INFO /apps/vcsmx/vcs/U-2023.03-SP2//etc/uvm-1.2/src/base/uvm_objection.svh(1276) @ 185: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
apb_test::report_phase
UVM_INFO /apps/vcsmx/vcs/U-2023.03-SP2//etc/uvm-1.2/src/base/uvm_report_server.svh(904) @ 185: reporter [UVM/REPORT/SERVER]
--- UVM Report Summary ---

** Report counts by severity
UVM_INFO :    4
UVM_WARNING :    0
UVM_ERROR :    0
UVM_FATAL :    0
** Report counts by id
[RNTST]    1
[TEST_DONE]    1
[UVM/RELNOTES]    1
[UVMTOP]    1
```

Figure 3.23: Log Result for single read/write cycle

- For n-writes and n-reads:

```
UVM_INFO @ 0: reporter [RNTST] Running test apb_n_wr_n_rd_test...
apb_test::build_phase
apb_env::build_phase
apb_agent::build_phase
apb_drv::build_phase
apb_mon::build_phase
apb_sqr::build_phase
apb_agent::connect_phase
apb_test::connect_phase
UVM_INFO /apps/vcsmx/vcs/U-2023.03-SP2//etc/uvm-1.2/src/base/uvm_root.svh(589) @ 0: reporter [UVMTOP] UVM testbench topology:
-------------------------------------------------------------
Name                      Type                    Size  Value
-------------------------------------------------------------
uvm_test_top              apb_n_wr_n_rd_test      -     @340
  env                     apb_env                 -     @353
    agent                 apb_agent               -     @362
      cov                 apb_cov                 -     @546
        analysis_imp      uvm_analysis_imp        -     @555
      drv                 apb_drv                 -     @371
        rsp_port          uvm_analysis_port       -     @390
        seq_item_port     uvm_seq_item_pull_port  -     @380
      mon                 apb_mon                 -     @537
        ap_port           uvm_analysis_port       -     @566
      sqr                 apb_sqr                 -     @400
        rsp_export        uvm_analysis_export     -     @409
        seq_item_export   uvm_seq_item_pull_imp   -     @527
        arbitration_queue array                   0     -
        lock_queue        array                   0     -
        num_last_reqs     integral                32    'd1
        num_last_rsps     integral                32    'd1
-------------------------------------------------------------
```

Figure 3.24: Log Result for multiple read/write cycle

```
---------------------------------------------------------------------------
Name                          Type      Size  Value
---------------------------------------------------------------------------
req                           apb_tx    -     @608
  addr                        integral  6     'h24
  data                        integral  16    'hcbce
  wr_rd                       integral  1     'h1
  begin_time                  time      64    0
  depth                       int       32    'd2
  parent sequence (name)      string    3     seq
  parent sequence (full name) string    30    uvm_test_top.env.agent.sqr.seq
  sequencer                   string    26    uvm_test_top.env.agent.sqr
---------------------------------------------------------------------------
---------------------------------------------------------------------------
Name                          Type      Size  Value
---------------------------------------------------------------------------
req                           apb_tx    -     @642
  addr                        integral  6     'h4
  data                        integral  16    'h2866
  wr_rd                       integral  1     'h1
  begin_time                  time      64    45
  depth                       int       32    'd2
  parent sequence (name)      string    3     seq
  parent sequence (full name) string    30    uvm_test_top.env.agent.sqr.seq
  sequencer                   string    26    uvm_test_top.env.agent.sqr
---------------------------------------------------------------------------
---------------------------------------------------------------------------
Name                          Type      Size  Value
---------------------------------------------------------------------------
req                           apb_tx    -     @650
  addr                        integral  6     'h6
  data                        integral  16    'h9b02
  wr_rd                       integral  1     'h1
  begin_time                  time      64    85
  depth                       int       32    'd2
  parent sequence (name)      string    3     seq
  parent sequence (full name) string    30    uvm_test_top.env.agent.sqr.seq
  sequencer                   string    26    uvm_test_top.env.agent.sqr
---------------------------------------------------------------------------
---------------------------------------------------------------------------
```

Figure 3.25: Log result for multiple read/write cycle

3.3.2 Waveforms:

- For single read/write:

Figure 3.26: Waveform for single read/write test

- For n-writes and n-reads:

Figure 3.27: Waveform for multiple read/write test

4. Integrating Tool for Repeaters

Intel specific tools is used for inserting repeaters in the path between senders and targets. The tool consumes file which has location of repeaters to be inserted in the RTL path. This file is developed by Physical Design to be used by front end tools which reduces the time of verification. The repeaters inserted are not to be on the exact place, but it reduces the time for early verification.

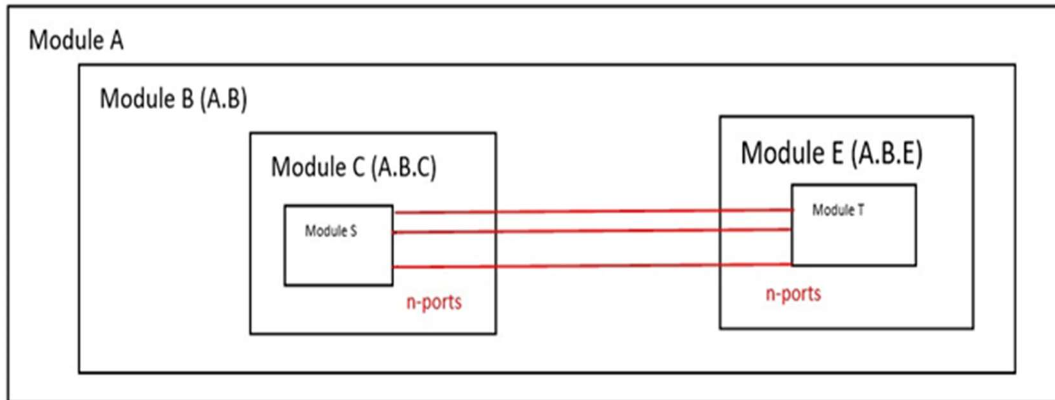


Figure 4.1: Block representation before Repeaters Insertion

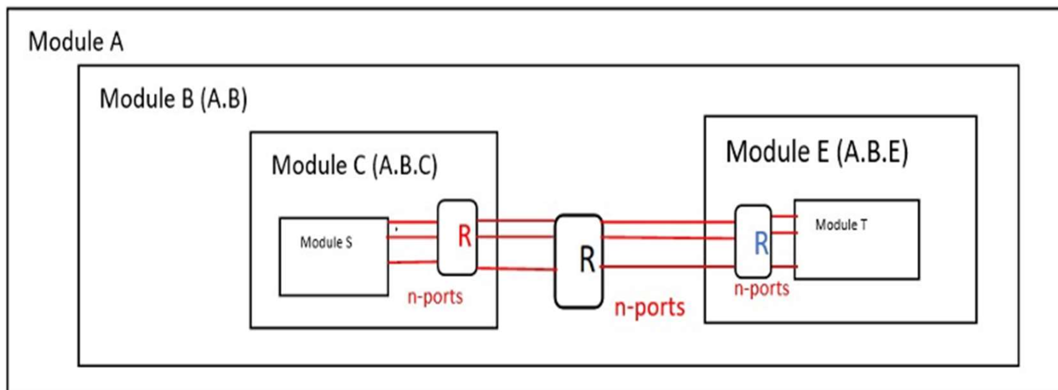


Figure 4.2: Block representation after Repeaters Insertion

4.1 Avoid Looping in Repeaters

Problem Statement: To identify the repeater insertion is not forming any loop connection if it is needed to avoid that condition.

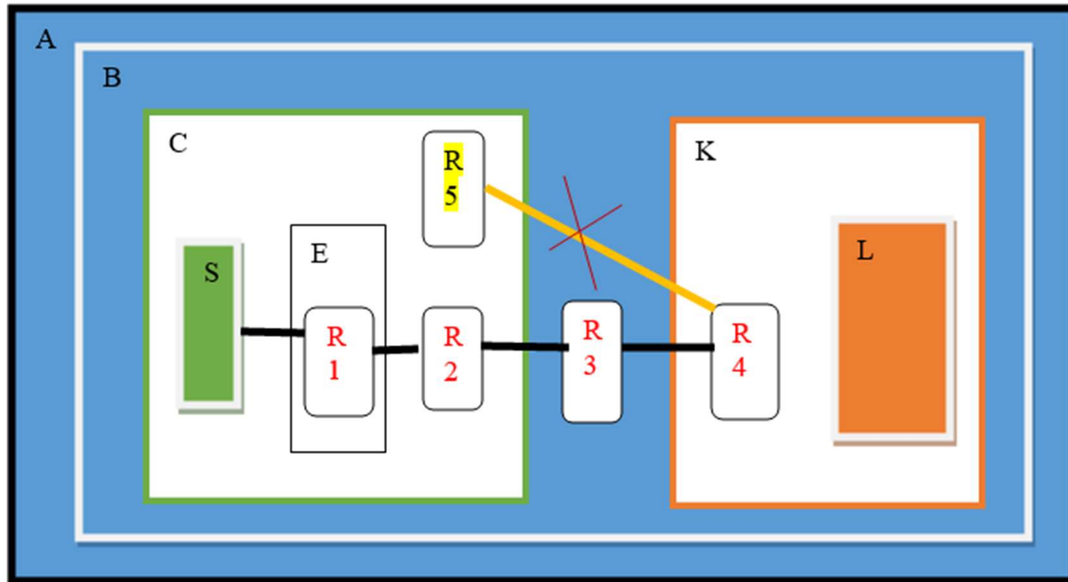


Figure 4.3: Block representation of Repeaters forming Loop connection

Let the repeater insertion be in the following manner:
Module S is Sender and Module L is Target
Repeater Tree:
ModuleA.ModuleB.ModuleC.ModuleS
ModuleA.ModuleB.ModuleC.ModuleE- rep_id-“R1”
ModuleA.ModuleB.ModuleC- rep_id-“R2”
ModuleA.ModuleB- rep_id-“R3”
ModuleA.ModuleB.ModuleK- rep_id-“R4”
ModuleA.ModuleB.ModuleC- rep_id-“R5”
ModuleA.ModuleB.ModuleK.ModuleL

- Here the first repeater R1 is inserted in Module E, which is inside the sender partition. So first we need to identify whether the first repeater inserted inside a partition or a partition itself.
- If any time the repeater insertion is completed inside a partition and exit that partition, and then another partition hierarchy started
    - for ex in the above Repeater tree case repeater insertion is completed for ModuleC and repeater insertion started Module B and ModuleB.ModuleK :
        - ModuleA.ModuleB.ModuleC- rep_id-“R2”
        - ModuleA.ModuleB- rep_id-“R3”
        - ModuleA.ModuleB.ModuleK- rep_id-“R4”
- Then again Repeater insertion started in Module C, which is a Loop connection.
        - ModuleA.ModuleB.ModuleK- rep_id-“R4”
        - ModuleA.ModuleB.ModuleC- rep_id-“R5”

Test case:

```
We need to identify, because this will form a loop kind of connection which Grip will drop that repeater tree.
#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <set>
using namespace std;
multimap<string,string> module_map;
string parmodule;
string sendmodule;

void Exit_Partition(vector<string> parvec, vector<string> repvec)
{
   for(int itr=0; itr<parvec.size(); itr++)
   {
      for(int its=itr+1; its<parvec.size(); its++)
      {
         if(parvec[itr] == parvec[its])
         {
            parmodule = parvec[itr];
            for(int its=0; its!=repvec.size(); its++)
            {
               sendmodule = repvec[its];
               module_map.insert(pair<string,string>(parmodule,sendmodule));
            }
         }
      }
   }
}

int main()
{
   vector<string> parvec{"M_A.M_B.M_C.M_E", "M_A.M_B.M_C", "M_A.M_B", "M_A.M_B.M_K", "M_A.M_B.M_C"};
   vector<string> repvec{"R5"};
   Exit_Partition(parvec,repvec);
   for(auto itr= module_map.begin(); itr!=module_map.end(); ++itr){
      cout<<(*itr).first<<" "<<(*itr).second<<endl;
   }
   return 0;
}
```

Figure 4.4: Test Case to Trace the Looping node

Results: As we need to trace the path which was causing the loop connection and we able to trace it successfully.

```
/tmp/wbLEIjVL5h.o
M_A.M_B.M_C R5


=== Code Execution Successful ===
```

Figure 4.5: Trace of node forming Loop connection

5. Conclusion and Future Scope

The functional verification of the AMBA APB (Advanced Peripheral Bus) bus architecture is a critical step in ensuring the reliability and performance of System on Chip (SoC) designs that utilize this protocol. The AMBA APB is designed for low bandwidth and low performance peripheral devices, making it essential for connecting components like UARTs, timers, and other peripheral units within an SoC. Given its role, the thorough verification of the APB bus architecture is paramount to the overall functionality and efficiency of the system.

The verification of the AMBA APB bus architecture is a comprehensive process that requires meticulous planning, detailed testing, and thorough analysis. By employing a structured verification methodology using UVM methods and classes help us in developing an effective testbench, ensuring protocol compliance, and rigorously testing performance under various conditions. Ultimately, the success of functional verification lies in its ability to identify and rectify issues before production, ensuring that the final product meets the required specifications and performs reliably in real-world applications.

APB is a widely used low-power interface in System-on-Chip (SoC) designs, particularly in ARM-based architectures. The future of the APB protocol could involve enhancements aimed at increasing efficiency, reducing power consumption, and improving data throughput.

Efforts might focus on improving interoperability with other bus protocols or interfaces, enabling easier integration within heterogeneous computing environments.

The repeater insertion tool represents a pivotal innovation in the field of electronic design automation (EDA). Its ability to validate the need for repeaters and automate their insertion addresses a critical aspect of design optimization, ensuring signal integrity and performance across complex circuits. As electronic designs continue to evolve in complexity and speed, tools like these become indispensable for designers seeking to achieve optimal performance, reliability, and efficiency in their projects. The repeater insertion tool not only simplifies the design process but also enhances the overall quality and effectiveness of electronic systems.

The future of a front-end tool for repeater insertion in SOC design enhances timing closure, optimizes performance, improves power efficiency, automates repetitive tasks, scales with design complexity, and offers flexibility to meet diverse design requirements. These advantages collectively contribute to more efficient, reliable, and competitive SOC designs in today's semiconductor industry.

References

[1] N.Venkateswara Rao, PV Chandrika, Abhishek Kumar and Sowmya Reddy,“ Design of AMBA based AHB2APB protocol for efficient utilization of AHB and APB”, International Research Journal of Engineering and Technology (IRJET), Volume 07, Issue 03, pp. 2395-0072, 2020.

[2] B H Niharika; S Ramesh,“ The Configuration and Verification Analysis of AMBA-Based AHB2APB Bridge”, Publisher: IEEE, Published in: 2022 IEEE 2nd International Conference on Mobile Networks and Wireless Communications (ICMNWC)

[3] Vaishnavi R.K, Bindu.S and Sheik Chandbasha, “Design and Verification of APB Protocol by using System Verilog and Universal Verification Methodology”, International Research Journal of Engineering and Technology (IRJET), Volume 06, Issue 06, pp. 2395- 0072, 2019.

[4] M. Kiran Kumar, Amrita Sajja and Dr. Fazal Noorbasha, “Design and FPGA Implementation of AMBA APB Bridge with Clock Skew Minimization Technique”, IOSR Journal of VLSI and Signal Processing (IOSR-JVSP), Volume 7, Issue 3, pp. 42-45, 2017.

[5] Kiran Rawat, Kanika Sahni and Sujata Pandey, “RTL Implementation for AMBA ASB APB Protocol at System on Chip level”, IEEE International Conference on Signal Processing and Integrated Networks (SPIN).

[6] Muhammad Hafeez and Azilah Saparon, “IP Core of Serial Peripheral Interface (SPI) with AMBA APB Interface”, IEEE 9th Symposium on Computer Applications & Industrial Electronics (ISCAIE), 2019.Manu B.N and Prabhavathi P, “Design and Implementation of AMBA.

[7] Proceedings of the Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV 2021). IEEE Xplore Part Number: CFP21ONG-ART; 978-0-7381-1183-4,“Design and Verification of Advanced Microcontroller Bus Architecture-Advanced Peripheral Bus (AMBA-APB) Protocol”

[8] Methodology for Repeater Insertion Management in the RTL, Layout, Floorplan and Full chip Timing Databases of the Itanium™ Microprocessor.

[9] Harpreet Vohra, Debbrat Ghosh, “Optimal Test Solution for Hierarchical Core Based System-on-Chip”, International Journal of Computer Science and Information Security (IJCSIS), Vol. 18, No. 4, April 2020.

[10] Flip-Flop and Repeater Insertion for Early Interconnect Planning Ruibing Lu, Guoan Zhong, Cheng-Kok Koh ECE, Purdue University West Lafayette, IN, 47907, USA flur, zhongg, chengkokg@ecn.purdue.edu Kai-Yuan Chao Intel Corporation Hillsboro, OR 97124, USA kchao@ichips.intel.com.

[11] “Predictive technology model (PTM).” Accessed: Feb. 2019. [Online]. Available: http://ptm.asu.edu/

[12] T. Jabbari, G. Krylov, S. Whiteley, J. Kawa, and E. G. Friedman, “Global signaling for large scale RSFQ circuits,” in Proc. Govt. Microcircuit Appl. Crit. Technol. Conf., Mar. 2019, pp. 1–6.

[13] L. Schindler, P. le Roux, and C. J. Fourie, “Impedance matching of passive transmission line receivers to improve reflections between RSFQ logic cells,” IEEE Trans. Appl. Supercond., vol. 30, no. 2, Mar. 2020, Art. no. 1300607.

[14] Manjit Kaur, Sanjeev Kumar, Balwinder Raj, Neena Gupta, Arun Kumar Singh, “RF and Crosstalk Analysis of Copper and MLGNR Interconnects Using Different Repeaters in Sub-10nm Regime”, March 2nd, 2021.

[15] Zhao, W.S., Liu, P.W., Yu, H., Hu, Y., Wang, G., Swaminathan, M.: Repeater insertion to reduce delay and power in copper and carbon nanotube-based nano interconnects. IEEE Access, 7, 13622-13633 (2019).
[16] Kumari, B., Sahoo, M.: Performance and signal integrity analysis of intercalation-doped MLVGNR interconnects. IET Circuits, Devices & Systems, 14(2), 192-199 (2019).
[17] J. Bromley "If System Verilog Is So Good, Why Do We Need the UVM? Sharing Responsibilities between Libraries and the Core Language".
[18] Kaur, M., Gupta, N., Singh, A.K.: Crosstalk analysis of coupled MLGNR interconnects with different types of repeater insertion. Microprocessors and Microsystems, 67, 18-27 (2019).
[19] Kumbhare, V.R., Paltani, P.P., Venkataiah, C., Majumder, M.K.: Analytical study of bundled MWCNT and edged MLGNR interconnects: impact on propagation delay and area. IEEE Transactions on Nanotechnology, 18, 606-610 (2019).
[20] S. Raghuvanshi, V. Singh "Review on Universal Verification Methodology (UVM) Concepts for Functional Verification" International Journal of Electrical, Electronics and Data Communication, ISSN: 2320-2084 Volume-2, Issue-3.

602262027_Report_Intro-_Conclusion.docx

ORIGINALITY REPORT

12%	9%	7%	5%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

#	Source	%
1	Padmaprabha Jain, 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), 2021 Publication	2%
2	www.slideshare.net Internet Source	2%
3	B H Niharika, S Ramesh. "The Configuration and Verification Analysis of AMBA-Based AHB2APB Bridge", 2022 IEEE 2nd International Conference on Mobile Networks and Wireless Communications (ICMNWC), 2022 Publication	1%
4	Submitted to CSU, San Jose State University Student Paper	1%
5	usermanual.wiki Internet Source	1%

#	Source	Type	Match
6	fr.scribd.com	Internet Source	1%
7	docshare.tips	Internet Source	1%
8	www.researchgate.net	Internet Source	<1%
9	Submitted to University of Southampton	Student Paper	<1%
10	Jalle, Javier, Jaume Abella, Eduardo Quinones, Luca Fossati, Marco Zulianello, and Francisco J. Cazorla. "AHRB: A high-performance time-composable AMBA AHB bus", 2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014.	Publication	<1%
11	vdocument.in	Internet Source	<1%
12	doczz.net	Internet Source	<1%
13	patents.justia.com	Internet Source	<1%
14	www.grossarchive.com	Internet Source	<1%
15	Submitted to Manipal University	Student Paper	<1%

16	Submitted to University of Surrey Student Paper	<1%
17	www.ijert.org Internet Source	<1%
18	B.N Manu, P Prabhavathi. "Design and implementation of AMBA ASB APB bridge", 2013 International Conference on Fuzzy Theory and Its Applications (iFUZZY), 2013 Publication	<1%
19	Submitted to California State University, Sacramento Student Paper	<1%
20	medium.com Internet Source	<1%
21	Submitted to International Islamic University Malaysia Student Paper	<1%
22	caesjournals.org Internet Source	<1%
23	pdfcoffee.com Internet Source	<1%
24	verificationacademy.com Internet Source	<1%
25	en.wikipedia.org Internet Source	<1%

26	www.semanticscholar.org Internet Source	<1 %
27	Coppola, . "On-Chip Bus vs. Network-on-Chip", System-on-Chip Design and Technologies, 2008. Publication	<1 %
28	Nikita Deshpande, Ranjit Sadakale. "AMBA AHB to APB Bridge Protocol Verification Using System Verilog", 2023 First International Conference on Advances in Electrical, Electronics and Computational Intelligence (ICAEECI), 2023 Publication	<1 %
29	Vaibbhav Taraate. "Digital Logic Design Using Verilog", Springer Science and Business Media LLC, 2022 Publication	<1 %

Exclude quotes Off
Exclude bibliography Off
Exclude matches Off