

A dissertation

On

**“Design and Verification of Advanced Peripheral
Bus Protocol using Universal Verification
Methodology”**

Submitted in Partial Fulfilment of the Requirements for the Degree

Of

Master of Technology

(Embedded Systems and VLSI Design)

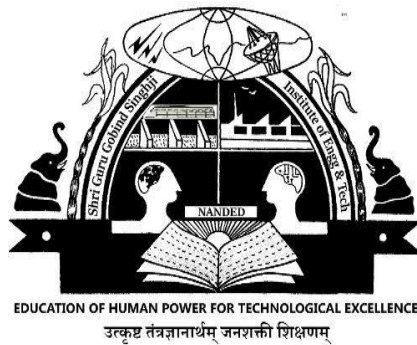
By

Mr. Mayur G. Khairnar

(Reg. No. 2017MVE008)

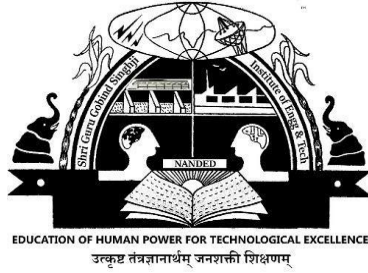
Under the Guidance of

Prof. Dr. Y. V. Joshi



DEPARTMENT OF ELECTRONICS AND TELECOMMUNICATION
ENGINEERING
**SHRI GURU GOBIND SINGHJI INSTITUTE OF
ENGINEERING AND TECHNOLOGY, NANDED (MS)**

July 2019



**Department of Electronics and Telecommunication Engineering
Shri Guru Gobind Singhji Institute of Engineering and Technology
Vishnupuri, Nanded (M.S.), India-431606**

Certificate

This is to certify that the dissertation entitled “**Design and Verification of Advanced Peripheral Bus Protocol using Universal Verification Methodology**” is a dissertation work carried out by **Mr. Mayur G. Khairnar (Reg. No. 2017MVE008)** under our/my supervision and guidance at Department of Electronics and Telecommunication Engineering, Shri Guru Gobind Singhji Institute of Engineering and Technology, Nanded for the award of the degree of **Master of Technology in Embedded system and VLSI Design.**

The content of this dissertation work, in full or parts have not been submitted to any other Institute or University for the award of any other degree or diploma.

Date: /07/2019

Place: SGGSIE&T Nanded.

**Prof. R. R. Manthalkar
M. Tech Coordinator
S.G.G.S.I.E.&T.,
Nanded.**

**Dr. M.B.Kokare
H. O. D, EXTC
S.G.G.S.I.E.&T.,
Nanded.**

**Dr. Y. V. Joshi
Director
S.G.G.S.I.E.&T.,
Nanded.**

DISSERTATION APPROVAL SHEET

The Dissertation entitled “**Design and Verification of Advanced Peripheral Bus Protocol using Universal Verification Methodology**” by Mayur G. Khairnar (Reg. No 2017MVE008) is approved for the degree of Master of Technology in **Embedded system and VLSI design** from **Shri Guru Gobind Singhji Institute of Engineering and Technology, Nanded (M.S) India.**

Prof. Dr. Y. V. Joshi
Project Guide

Examiner(s):

(1)

(2)

DECLARATION

I declare that, the Dissertation entitled “**Design and Verification of Advanced Peripheral Bus Protocol using Universal Verification Methodology**”, is submitted for the award of “Master of Technology” in Embedded Systems and VLSI Design Engineering to the Shri Guru Gobind Singhji Institute of Engineering and Technology, Nanded (MH).

This is a work built on Protocol Specifications given by Arm Inc. I have adequately cited and referenced the original sources. Having adhered to all principles of academic honesty, I also understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Mr. Mayur G. Khairnar

Reg. No.:2017MVE008

M.Tech
(VLSI Design and Embedded System)

ACKNOWLEDGEMENTS

This Dissertation work is in continuum with my internship and all the relevant coursework which I have studied at my institute. Therefore, it cannot be completed until I express my sincere gratitude towards the people whose guided me throughout to make this work possible.

I sincerely express gratitude towards my dissertation guide **Dr. Y. V. Joshi**, *Director, SGGSIE&T, Nanded*, for his support, encouragement and valuable guidance throughout my work and also for always enlightening us about how to bridge the gap between institute & industry. I am thankful to **Mr. C.S.Basha**, *DV Engineer at SION Semi* for his constant support. I am also thankful to **Prof R. R. Manthalkar**, for constant moral boosting during my stay in the campus. I thank **Prof S.N.Talbar**, for guiding as well as organizing expert sessions in Embedded Systems domain. I shall be indebted to **Prof S. S. Gajre** for making us aware about making use of Open Source tools and guiding us about making good presentations & writing thesis. I am extremely thankful to **Dr. Y. V. Joshi**, **Dr. R. R. Manthalkar** and **Dr. S. S. Gajre** for letting us make use of **Centre of Design and Verification Lab** (*with all Mentor softwares*) at institute, a rare advantage over most premier institutes in the country.

I would also like to thank **Dr. A. B. Gonde**, and **Dr. M.B. Kokare** , HOD, Department of Electronics and Telecommunication Engineering, SGGSIE&T Nanded, for providing me all the facilities in college, arranging various workshops and financially support to attend the various technical workshops which help me a lot in enhancing my technical skills and successful completion of this dissertation work.

I would also like to thank all my M. Tech colleagues and seniors for supporting me by their ideas and knowledge every time. Special Thanks to **Mr. Juber Shaikh & Mr. Harshwardhan Deshmukh** for being by my side during my internship days. Finally, I would like to thank my family for unconditionally supporting me financially and emotionally throughout my tenure at SGGSIET, Nanded.

Mayur G. Khairnar

Reg No: 2017MVE008

ABSTRACT

VLSI is no more a buzzword now. Its ULSI everywhere. It is difficult to match pace with time to market along with maximal efficiency. Existing HDLs/HVLs need to incorporate various new ways to result in faster verification of Intellectual Property and System on chip (IP/SOC). Verification of customizable communication IPs like Advanced Peripheral Bus (APB), Universal Asynchronous Receiver Transmitter (UART), Inter-Integrated Circuit (I2C), Universal Serial Bus (USB) and other such Protocols have a huge importance, prior to implementation on SOC. Communication Protocols are mounted on hardware devices like Field Programming Gate Array (FPGA), ARM boards, etc. Design and Verification Environment coding of APB Protocol is done in Mentor Graphics tool Questa Sim-10.4e.

Verilog can be used for design purpose of APB Protocol. Its verification has been implemented with powerful features like OOP used in SV and separation of TB & stimuli using UVM. Entire Testbench is UVM dependent. Its features have been verified by writing various test cases and coverage driven functional verification. Its use can be extended to interface various low bandwidth peripherals which can be remodeled using Verilog/SV.

KEYWORDS-: APB, UART, IP/SOC, HDL/HVL, UVM, SV.

Contents

CERTIFICATE.....	I
DISSERTATION APPROVAL SHEET	II
DECLARATION.....	III
ACKNOWLEDGEMENTS.....	IV
ABSTRACT.....	i
List of Figures.....	v
List of Tables	vii
Timing Diagram Conventions.....	viii
 Chapter 1	 1
INTRODUCTION.....	1
1.1 Background	1
1.2 Objectives of the Dissertation	2
1.3 Literature Review.....	3
1.4 Organization of Dissertation Report	3
 Chapter 2	 4
OVERVIEW OF APB PROTOCOL.....	4
2.1 An AMBA Based Microcontroller Architecture	4
2.2 About Protocols.....	5
2.2.1 Comparison between Protocols.....	5
2.2.2 AMBA APB Protocol	6
2.2.2.1 APB Protocol Write transfer	7
2.2.2.2 APB Protocol Read Transfer.....	8
2.2.2.3 APB Protocol Operating states.....	8
2.2.2.4 Features of APB Protocol.....	9
2.3 Signal description of APB Protocol	10
2.3.1 APB Protocol Signal List	11

2.4. Summary	18
Chapter 3	19
APB PROTOCOL : UVM BASED ARCHITECTURE DESIGN.....	19
3.1. UVM Architecture for APB Protocol	19
3.1.1. APB Transaction	19
3.1.2. APB Driver	20
3.1.3. APB Sequencer	21
3.1.3.1. APB Sequence.....	21
3.2. Advanced UVM based Verification Environment	24
3.2.1. UVM Register Model.....	25
3.2.2. UVM Testbench Basic Component.....	27
3.3. Proposed Verification Environment.....	28
3.3.1. Testbench Component of UART Protocol Design and Verification Environment	28
3.4 Summary	29
Chapter 4	30
IMPLEMENTATION RESULTS	30
4.1. UART Protocol word format poll test.....	30
4.2. UART Protocol word format interrupt test	31
4.3. UART Protocol modem poll test.....	32
4.4. UART Protocol modem interrupt test	33
4.5. UART Protocol Baud rate test	34
4.6. UART Protocol Rx errors test analysis	35
4.7. Coverage Report	36
4.7.1. Word format poll test	36
4.7.2. Word format interrupt test.....	37
4.7.3. Modem poll test.....	37
4.7.4. Modem interrupt test	38
4.7.5. Baud rate test.....	39
4.7.6 Register access test.....	39
4.7.7 Receiver errors int test	40

4.8. Summary	40
Chapter 5	41
CONCLUSIONS AND SCOPE FOR FUTURE WORK.....	41
5.1 Conclusion	41
5.2 Scope for Future Work.....	41
REFERENCES.....	42

List of Figures/Waveforms

Figure : Rise of System Verilog as HDL/HVL

Figure: APB Write transfer without wait state

Figure : APB Write transfer with wait state

Figure: APB Read Transfer without wait state

Figure : APB Read Transfer with wait state

Figure: 2.6 Operating states of APB Protocol

Figure: 2.7 Architecture of UART Protocol

Figure : 3.1 Architecture of UART Protocol

Figure : 3.2 UART serial standard data format

Figure : 3.3 UART Protocol frame formats

Figure-: 4.6 Logs of Modem Polling Test

Figure-: 4.7 Modem Polling test simulation

Figure-: 4.8 Logs of modem interrupt test

Figure-: 4.9 Logs of Baud Rate test

Figure-: 4.10 Receiver error test analysis simulation

Figure-: 4.11 Logs of receiver errors test

Figure-: 4.12 Word format Polling without interrupt coverage

Figure-: 4.13 Word format with interrupt coverage

Figure-: 4.14 Modem poll test coverage

Figure-: 4.15 Modem interrupt test coverage

Figure-: 4.16 Baud Rate test Coverage

Figure-: 4.17 UART Protocol register access test

Figure-: 4.18 Receiver Error test Coverage

List of Tables

Table-: 2.1 APB Protocol Signal Description

Table-: 2.2 List of UART Protocol Functional Signals.....

Table-: 2.3 Interrupt Enable Register (IEN).....

Table-: 2.4 Interrupt Identification Register (IIR).....

Table-: 2.5 FIFO Control Registers (FCR).....

Table-: 2.6 Line Control Register (LCR)...

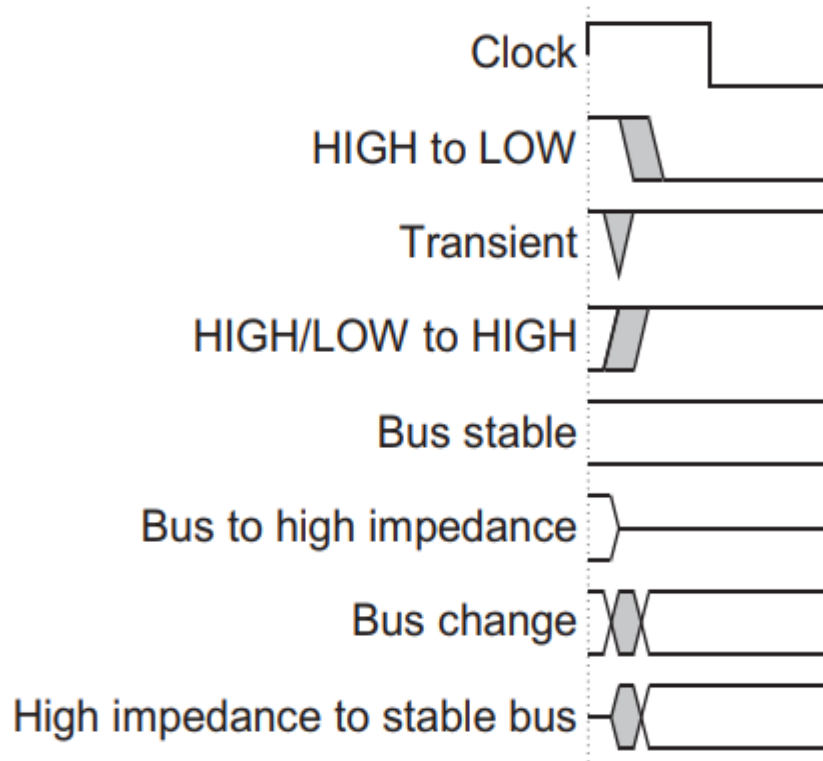
Table-: 2.7 Divisor Register.....

Table-: 2.8 Modem Control Register (MCR).....

Table-: 2.9 Modem Status Register (MCR).....

Table-: 2.10 Line Status Register (LSR)...

Timing Diagram Conventions



Chapter 1

INTRODUCTION

1.1 Background

The need of (IP/SOC) verification now becomes a challenging area, which needs to be improved in aspect of accuracy and speed, to achieve time to market. Verification of testbench becomes necessary because as semiconductor industry increasing, the design becomes more complex and so to test each signal and module manually is complicated. Design takes only 30% of time rather than Verification which requires more than 70% of time to make chips functional. So, to achieve this goal various Hardware Description and Verification languages came into existence, with Accellera and IEEE standards. Verilog has specific features to make design easily, but it is not much effective to make verification testbench due to lack of certain features. Therefore, System Verilog as an HDL/ HVL came into existence around 2005, a superset of Verilog, extending its features and also introducing the concept of Object-Oriented Programming to make testbench more flexible. OVM was existent during the same period which was taken over by UVM which comes with a library of classes and objects that helps in building efficient verification testbench architecture.

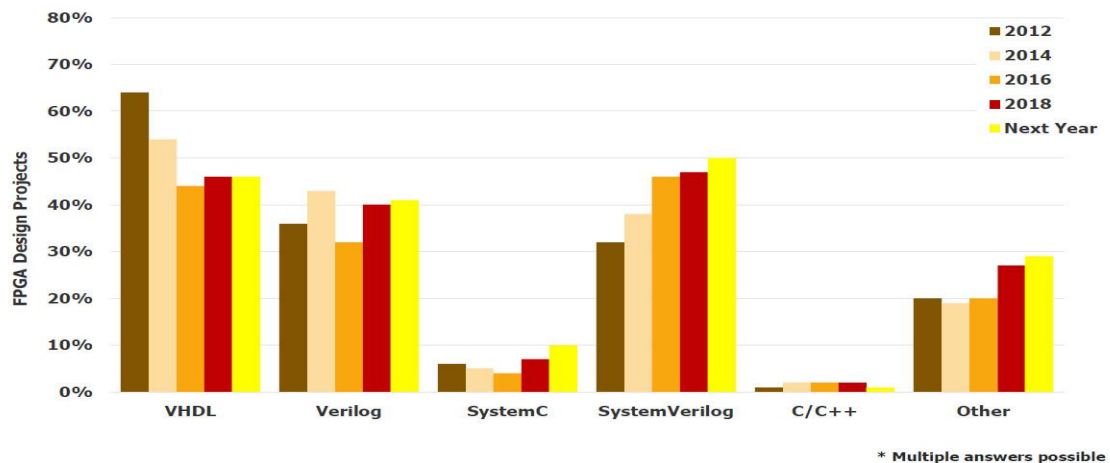


Figure 1.1: Rise of System Verilog as HDL/HVL (Source: Mentor)

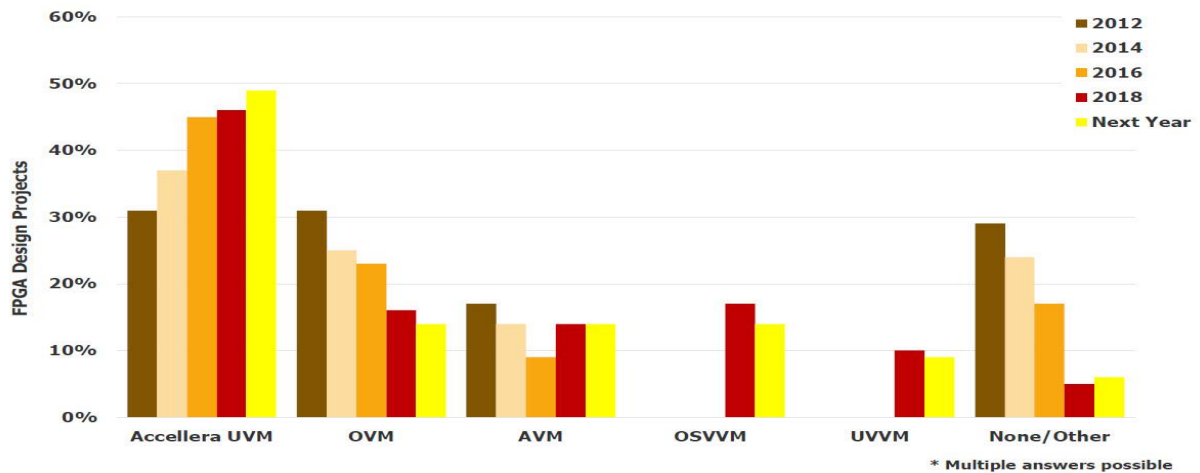


Figure 1.2 : UVM as a growing methodology for Verification (Source: Mentor)

Today's SOC consist of different buses and communication protocols, integrated on a single chip. So, before verifying this complex SOC we have to do functionality and features verification/validation of individual IP block. Communication protocols are one of the important IP blocks in SOC design, which are responsible for making communication of high speed and low speed peripheral devices with the processor through AMBA Bus i.e. AHB, APB, ASB, AXI Protocols. Various communication protocols are Ethernet, USB, I2C, UART and USART. In this dissertation, AMBA Peripheral bus Protocol (APB) is validated using features of SV & UVM.

1.2 Objectives of the Dissertation

1. Verification of APB Protocol by transmitting and receiving of variable data formats with and without interrupt.
2. Design of various verification components like monitor, driver, sequencer, agent etc. to implement basic functionality of standard APB Protocol.
3. Proposed UVM Testbench architecture, to verify APB Protocol functionality by minimal efforts.
4. Design of scalable testbench architecture.
5. Incorporating all the verification components into a single Verification environment which is implemented using UVM.

1.3 Literature Review

- Protocol features, register description and configuration with working principle discussed briefly in ARM Inc [1] and other referenced datasheets.
- UVM TB architecture with detailed description of each block described in UVM 1.2 user reference manual [2], DVCON research paper on UVM environment modelling [3]
- UVM cookbook by Mentor Graphics [4]. Design is implemented using Verilog rather because it is easy and flexible to design. Verification is achieved through SV & UVM due to its extended functionality of Verilog and Object-oriented Programming language.
- Springer textbook: “System Verilog for Design” [5] and “System Verilog for Verification” [6] and System Verilog user reference manual of Accellera.
- Detailed concepts of coverage Driven Verification mentioned in Springer textbook, “System Verilog Assertion & Functional Coverage” [7]. All mentioned books, research papers and manual contain detail information of design and verification.

1.4 Organization of Dissertation Report

This report consists of the five chapters.

- **Chapter 1** gives the background knowledge of APB protocol, Testbench Architecture, objective of the dissertation, literature review.
- The detailed description of APB protocol and the basic signal description of the AMBA APB protocol mentioned in **Chapter 2**.
- **Chapter 3** includes basic working of the AMBA APB protocol based on UVM infrastructure.
- Implementation results mentioned in **chapter 4**.
- **Chapter 5** concludes with protocol verification concept and scope for future work in it.

Chapter 2

OVERVIEW OF AMBA APB PROTOCOL

In this chapter AMBA APB Protocol is explained in details with all its configurations and signal descriptions.

2.1 AMBA Based Microcontroller Architecture

The AMBA Architecture base microcontroller is a typical design which defines the interfacing of high-speed communication protocols like I2C, SPI, USB, etc. and low speed communication protocol like UART and other with the high speed and high performance AMBA bus (AHB & AXI) protocols and low power APB Protocol. In this architecture we can see how low speed peripheral devices like keyboard, mouse, modem and other connected with the UART Protocol through the APB interface. Since both high and low speed bus protocol mention in the same architecture to communicate between respective peripherals and protocols, therefore to properly define this interface bridge used between them. A typical AMBA based microcontroller architecture shown in following Figure.

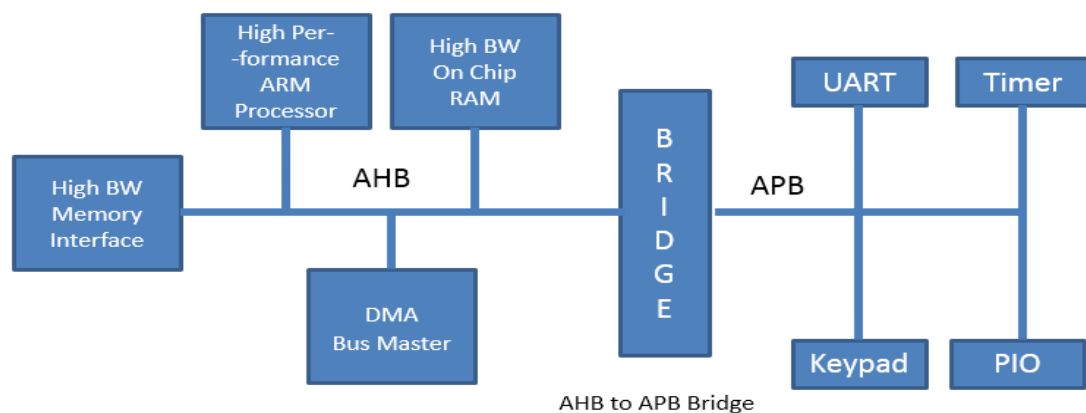


Figure 2.1 : AMBA BUS Architecture

2.2 Protocol

- Protocol is nothing but a set of rules used to define proper communication between two or more devices, so that for whatever goal it designed become fulfilled.
- In this dissertation we are basically dealing with communication protocol. Communication protocols are of several types depending upon serial and parallel, synchronous and asynchronous data communication and speed of operation of peripheral devices. Different communication protocols are UART, USART, SPI, I2C and another, each one of these used according to need of data transmission.
- Asynchronous communication of UART Protocol depends on baud rate therefore it requires low power and bandwidth. USART (Universal Synchronous, Asynchronous Receiver Transmitter), SPI (Serial peripheral interface) and I2C protocols are high speed and requires clock for data communication. So, it requires high power and bandwidth. Therefore, each of these protocols used according to need of peripheral devices and requirement for data communication.

2.2.1 AMBA APB Protocol

APB is the type of the AMBA 3 Protocol family. It provides a minimal cost interface, optimized for less power utilization and reduces the interface complexity nature. It can be used to provide an access to the peripheral that becomes low bandwidth and does not need high performance pipelined architecture. It has unpipelined protocol. Its every signal transition relates to the rising edge of signal transition to active the integration of APB peripherals into any design flow. Its each transfer takes twice cycles to complete every transfer. It basically describes two types of read and writes transfer i.e. with and without wait states.

2.2.2.1 APB Protocol Write transfer

The APB writes transfer happens with and without wait state. The write transfer

initiate with the write data, write signal, address and select signal, all fluctuating after the rising edge of the clock transition. The signal flow graph of the APB writes transfer with and without wait states can be mentioned below.

It is prescribed that the address and write signal are not changed instantly after a transfer but become durable until other access occurs. This has become do reduction in power consumption.

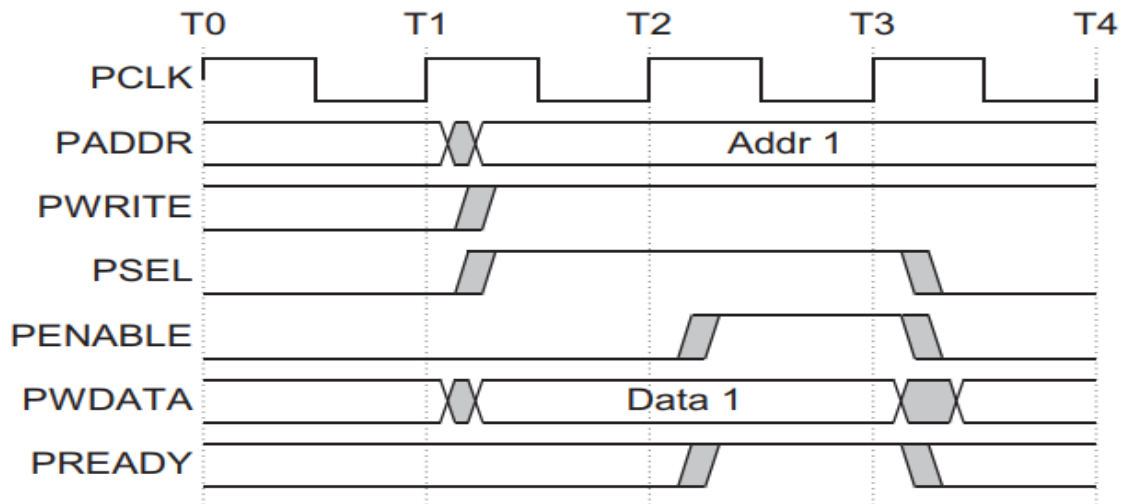


Figure 2.2: APB Write Transfer (No-Wait)

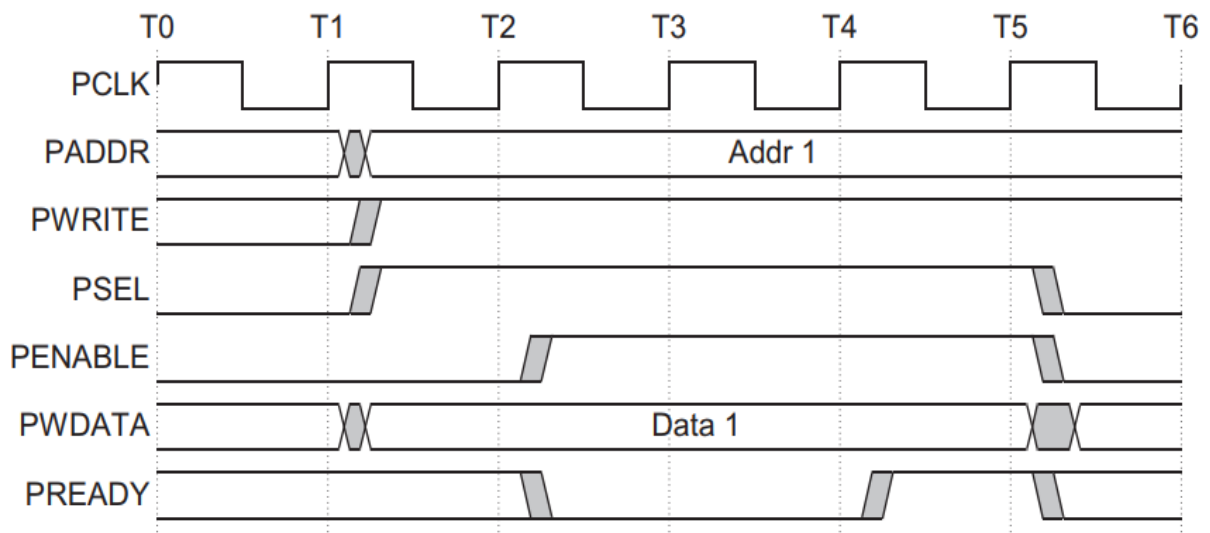


Figure 2.3: APB Write Transfer (with Wait)

2.2.2.2 APB Protocol Read Transfer

The APB read transfer happens with and without wait states. Below figure 2.4 shows

that timing of address, write, select and enable signals, as described in write transfer mentioned in above figure 2.2. The slave brings the data previous to the end of the read transfer. Figure 2.5 below show in what way the PREADY signal can elongate if PREADY is driven low during an access phase. The protocol makes sure address, write, select, and enable signal not to change for additional cycle.

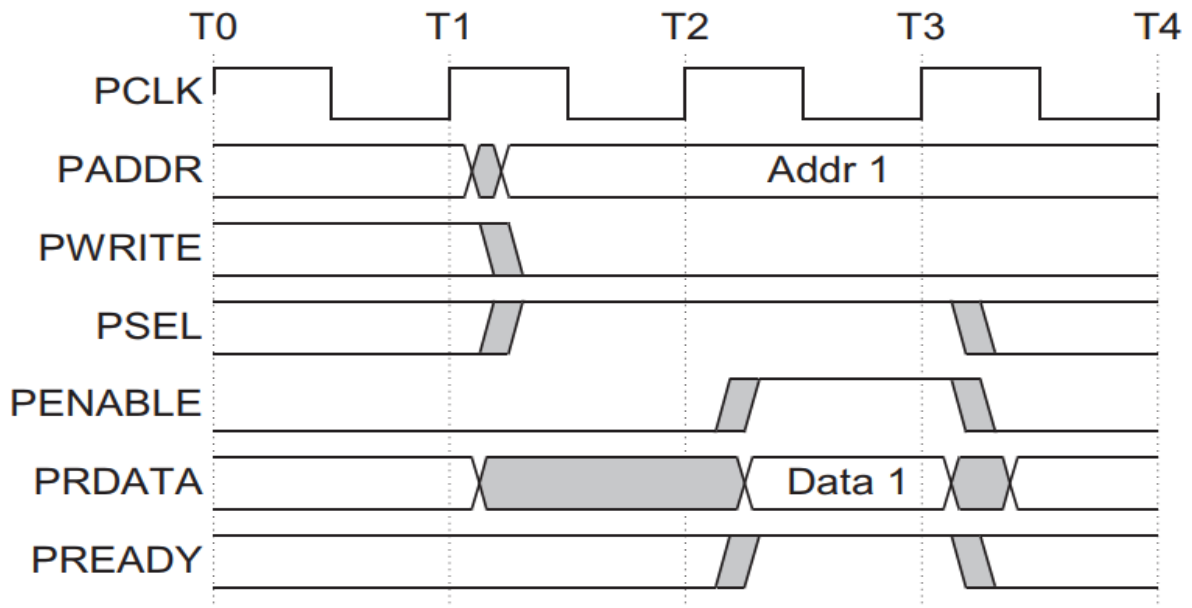


Figure 2.4: APB Read transfer (with No-Wait)

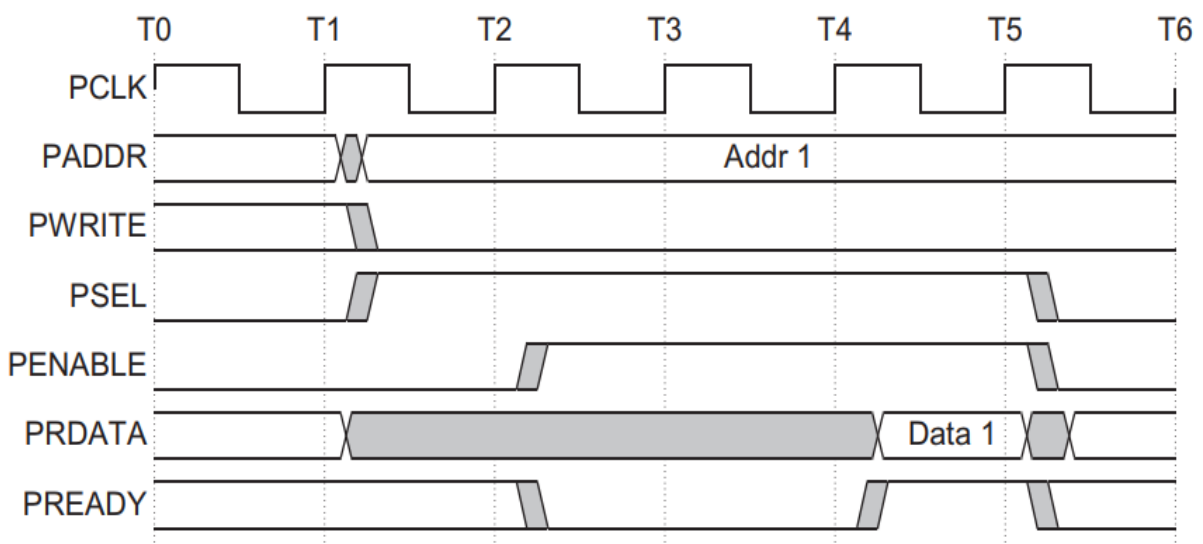


Figure 2.5: APB Read Transfer (With-Wait)

2.2.2.3 APB Protocol Operating states

The APB Protocol operation execute in different states mentioned in below figure

- Idle State**:- This is defined as the initial phase of APB Protocol operation.
- Setup State**:- At the time of data transfer it moves into setup phase, in which appropriate select signal, PSELx, is asserted. It became in the SETUP state for one clock cycle and every time moves to the ACCESS state on the next rising transition of the clock.
- Access State**:- The enable signal, Penable, is asserted in this state. The address, write, select and write data signals must remain steady during the transition from Setup to Access state is controlled by the Pready signal from slave. If Pready is low by the slave, then it became in Access state. If Pready became driven high by the slave, then the Access state is enhanced and the bus came to the Idle state if no other transfers are needed. In other way, the bus moves directly to the Setup state if another transfer follows.

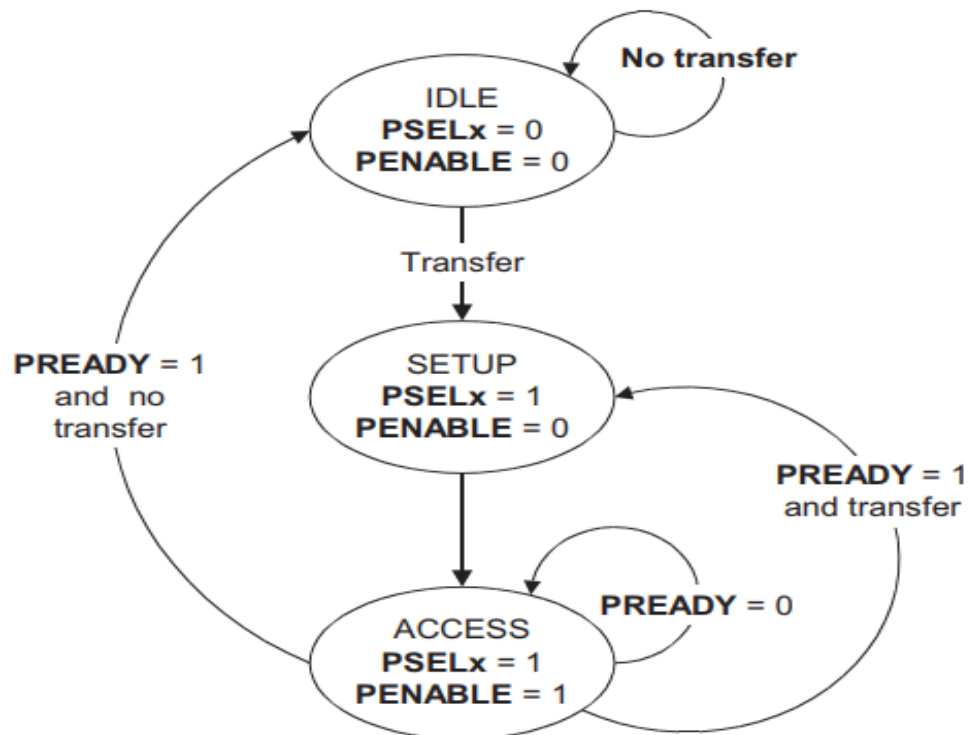


Figure 2.3: Operating States of APB Protocol

2.2.2.4 Features of APB Protocol

Following mentioned are the important features of AMBA APB Protocol

- The APB is part of the AMBA 3 protocol family. It provides a low-cost interface that is optimized for minimal power consumption and reduced interface complexity.
- All signal transitions are only related to the rising edge of the clock to enable the integration of APB peripherals easily into any design flow. Every transfer takes at least two cycles.
- The APB can interface with the AMBA Advanced High-performance Bus Lite (AHB-Lite) and AMBA Advanced Extensible Interface (AXI). You can use it to provide access to the programmable control registers of peripheral devices.
- The APB interfaces to any peripherals that are low-bandwidth and do not require the high performance of a pipelined bus interface. The APB has unpipelined protocol. Simple Interface.

2.3 APB Protocol Signal List

Following mentioned table shows the detailed description of APB Protocol interface signals and the overview of each signal functionality.

Signals	Directions	Description
PCLK	INPUT	Rising edge clock signal
PRESETn	INPUT	It has reset signal
PADDR	INPUT	32-bit wide APB address bus signal
PSELx	INPUT	Slave select signal. It became use to select the slave from multiple slaves
PENABLE	INPUT	Enable sign, it indicates the 2 nd cycle of APB transfer
PWRITE	INPUT	This is write signal. It indicates that it has to be write or not
PWDATA	INPUT	It is 32-bit write data signal need to transfer 32 bits of data
PREADY	OUTPUT	It is Ready signal, slave used this signal to extends APB transfer
PRDATA	OUTPUT	This is Read data signal. It became use to driven by selected slave in read mode when write mode become zero
PSLVERR	OUTPUT	This APB Slave error indicating a transfer failure in APB protocol data transfer.

Figure 2.7: APB Protocol Signals

Chapter 3

APB PROTOCOL TESTBENCH AND ARCHITECTURE DESIGN

3.1 Setting up UVM infrastructure for APB Protocol

This Chapter describes the working functionality of APB Protocol with its testbench architecture and modules design. It also describes about UVM testbench architecture with detail of its various components.

3.1.1 UVM Architecture

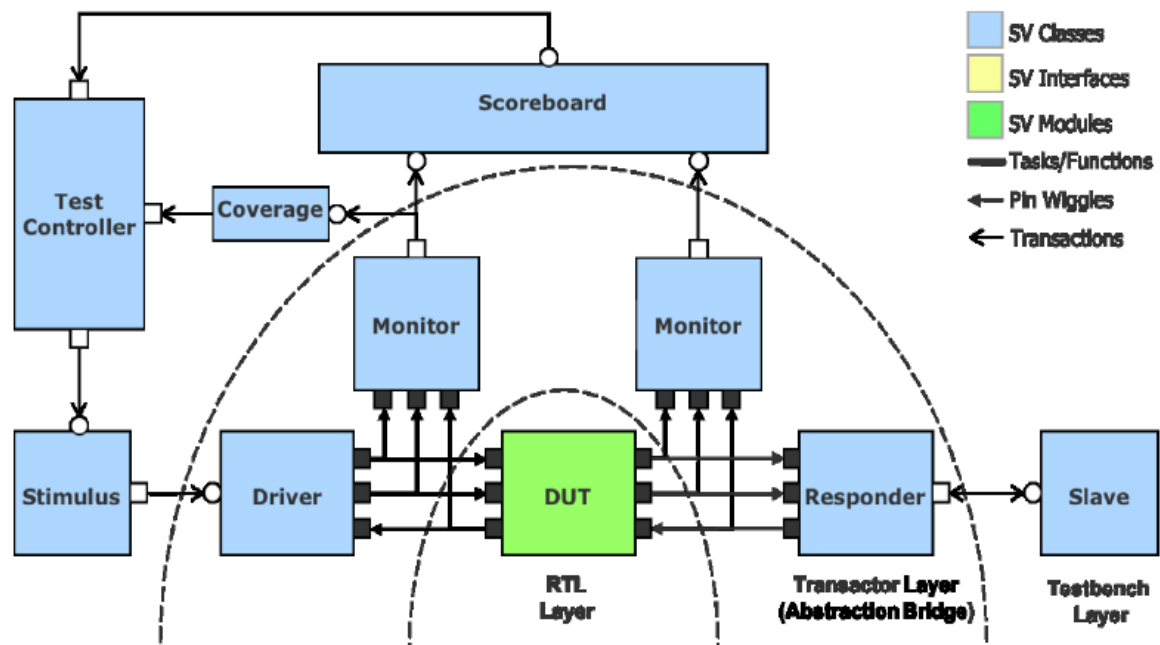


Figure 3.1: UVM Test bench Architecture (Source:Accellera IEEE Standard for UVM)

A UVM testbench is built using System Verilog (dynamic) class objects interacting with System Verilog (static) interfaces and modules in a structured hierarchy. The hierarchy is composed of layers of functionality. At the center of the testbench is the Design Under Test (DUT). It is connected to a layer of transactors (drivers, monitors, responders). The transactors communicate with the DUT at the pin level by driving and sampling DUT signals, and with the rest of the UVM testbench by passing transaction objects. They convert data between pins and transactions, i.e. from/to signal to/from transaction level. The testbench layer above the transactor layer consists of components that interact exclusively at the transaction level, such as scoreboards, coverage collectors, stimulus generators, etc.

The transactor and testbench layers are conventionally built purely from System Verilog classes. However, that construction style limits portability by only targeting a System Verilog simulator. A somewhat alternate style, or architecture, that utilizes System Verilog classes as well as System Verilog interfaces, can increase portability between execution engines. Taking advantage of both these System Verilog constructs, a natural split can be made in the transactor layer between transaction level communication grouped on one side separate from timed, signal level communication grouped on the other side.

3.1.2 Proposed testbench architecture for APB Protocol

The `uvm_monitor` is responsible for passively observing the pin-level behavior on the DUT interface, converting it into sequence items and providing those sequence items to analysis components in the agent or elsewhere in the testbench such as coverage collectors or scoreboards. UVM Agents also have a *configuration object* that allows the test writer to control aspects of the agent as the testbench is assembled and executed.

By providing a uniform interface to the testbench, a UVM Agent isolates the testbench and the UVM Sequence from details of the interface implementation. A sequence that provides data packets, for example, can be reused with different UVM Agents that may implement AHB, PCI or other protocols. A UVM testbench will typically have one agent per DUT interface.

For a given design, the UVM Agents and other components are encapsulated in a `uvm_env` environment component, which is typically design-specific. Like an agent, an environment typically has a configuration object associated with it that allows the test to control aspects of the environment as well as to control the agents instantiated in the environment. Because environments are themselves UVM components, they can be assembled into a higher-level environment.

As block-level designs are assembled into subsystems and systems, the block-level UVM environment associated with the block may be reused as a component in the subsystem-level environment, which can itself be reused in the system-level testbench.

Once the environment has been defined, the `uvm_test` will instantiate, configure and build the environment, including customizing key aspects of the overall testbench, including

- ..*choosing variations of components to be used in the environment
- ..*choosing UVM Sequences to be run either in the background or as the main portion of the test
- ..*defining configuration objects for the environment, sub-environment(s) (if any) and agent(s) in the testbench

The UVM test is started from an initial block in the top-level HVL module by calling `run_test()`.

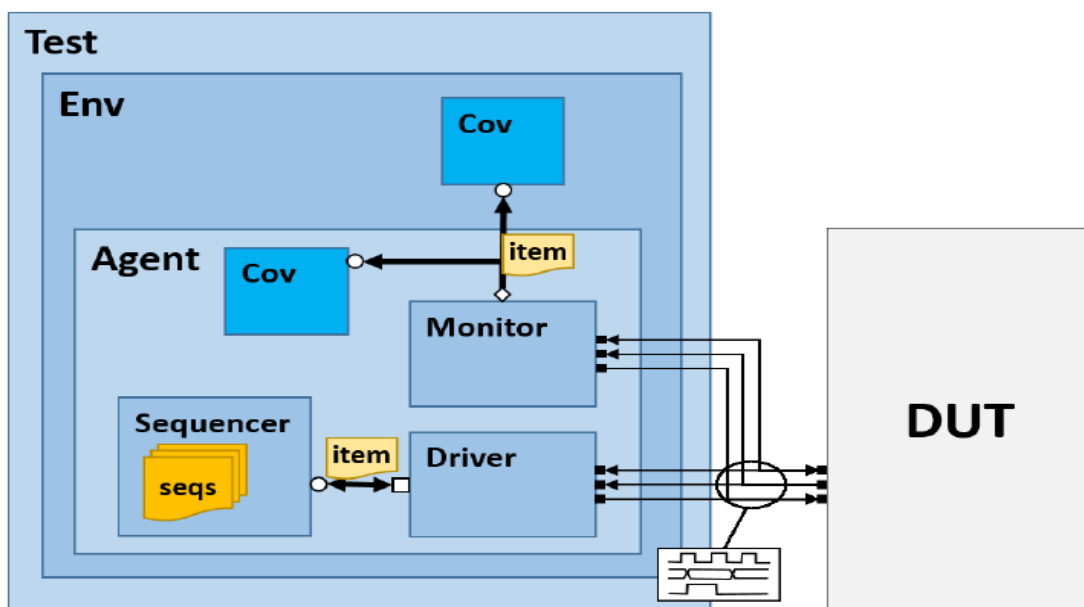


Figure 4.2 : Proposed UVM Architecture for APB Protocol

3.1.3 Testbench Components

The APB protocol testbench consists of various components used to perform its functionality. Each component behavior mentioned in below sections.

3.1.3.1 UVM Inheritance

1. A UVM testbench is composed of component objects extended from the `uvm_component` base class. When a `uvm_component` derived class object is created, it becomes part of the testbench hierarchy which persists for the duration of the simulation. This contrasts with the sequence branch of the UVM class hierarchy which involves transient objects - objects that are created, used and destroyed (i.e. garbage collected) once dereferenced.
2. The (quasi) static `uvm_component` hierarchy is used by the UVM reporting infrastructure to print the scope of a component issuing a report message, by the configuration process to determine which components can access a configuration object, and by the UVM factory to apply factory overrides. This component hierarchy is represented by a linked list built up incrementally as each component is created. The hierarchical location of each component is determined by the name and parent arguments passed to its create method at the time of construction.
3. The `uvm_component` class inherits from the `uvm_report_object` class which lies at the heart of the UVM Messaging infrastructure. The reporting process uses the component static hierarchy to add the scope of a component to the report message string.
4. The `uvm_component` base class template has a virtual method for each of the UVM Phases and these are to be implemented by the user as required. A phase level virtual method that is not implemented results in the component effectively not participating in that phase.
5. Also embedded into the `uvm_component` base class is support for a configuration table to

store configuration objects that are relevant for the component's child nodes in the testbench hierarchy. When the `uvm_config_db` API is used, this static hierarchy is employed as part of the path mechanism to control which components may access a given configuration object.

6. In order to provide flexibility in configuration and to allow the UVM testbench hierarchy to be built in an intelligent way, `uvm_component` are registered with the UVM factory. When a UVM component is created during the build phase, the factory is used to construct the component object. The UVM factory enables a component to be swapped for another of a compatible, derived type using a factory override. This is a useful technique for altering the functionality of a testbench without changing the testbench source code directly, which would require recompilation and hinder reuse. There are a number of coding conventions required for the factory to work and these are outlined in the article on the UVM Factory.

7. The UVM package contains a number of extensions (i.e. derived classes) of the `uvm_component` base class for common testbench components. Most of these extensions are very "thin", i.e. they are literally just a small extension of the `uvm_component` class to add a new name space. While these are non-critical and in principle a `uvm_component` class could be used instead still, they do help with "self-documentation" as they indicate clearly the kind of component that is actually represented, such as a driver or monitor. Additionally, there are analysis utilities available that also use these extraneous base classes as clues to help establish a picture of the testbench hierarchy. On the other hand, some of the pre-built `uvm_component` extensions are in fact building blocks providing more profound added value, by instantiate concrete sub-components. The following table summarizes the available UVM component classes directly derived from the `uvm_component` base class:

Class	Description	Contains sub-components?
uvm_driver	Encapsulates sub-components for sequence communication with the uvm_sequencer	No
uvm_sequencer	Encapsulates sub-components for sequence communication with the uvm_driver	No
uvm_subscriber	Encapsulates a uvm_analysis_export and associated virtual <i>write</i> method to implement analysis transaction processing	No
uvm_env	Basis for aggregating verification components around a DUT, or other envs in case of vertical (sub-)system integration	Yes
uvm_test	Basis for a concrete top level test	Yes
uvm_monitor	Basis for a concrete monitor transactor	Yes
uvm_scoreboard	Basis for a concrete scoreboard	Yes
uvm_agent	Basis for concrete agent including a sequencer-driver pair and a monitor	Yes

Figure 4 Summary of UVM Components

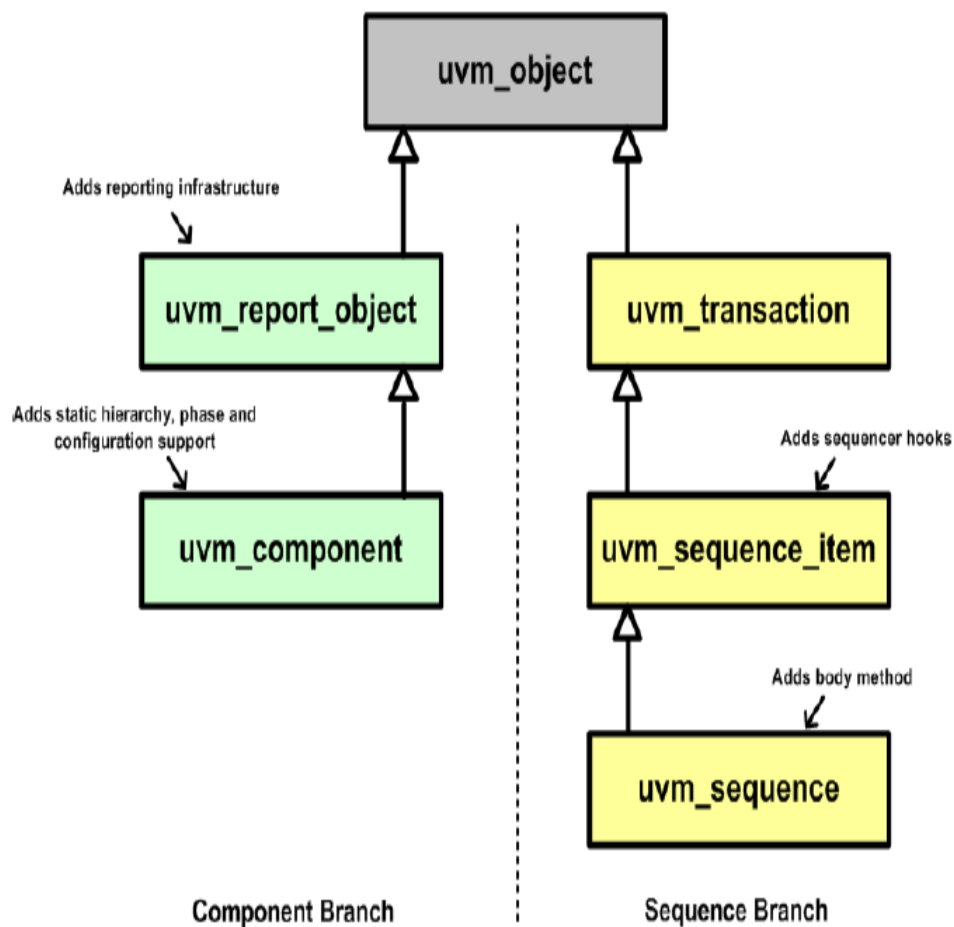


Figure 5 : UVM Inheritance Flow (Source: Accellera Cookbook)

3.1.3.2 UVM Factory

The purpose of the UVM factory is to enable an object of one type to be substituted with an object of a derived type without changing the testbench structure or even the testbench code. The mechanism used is referred to as an override, by either instance or type. This functionality is very handy for changing sequence behavior or replacing one version of a component by another. Any two components to be swapped must be polymorphically compatible. This includes the requirement that all the same TLM interface handles and TLM objects must be created by the replacement component.

3.1.3.3 APB Agent

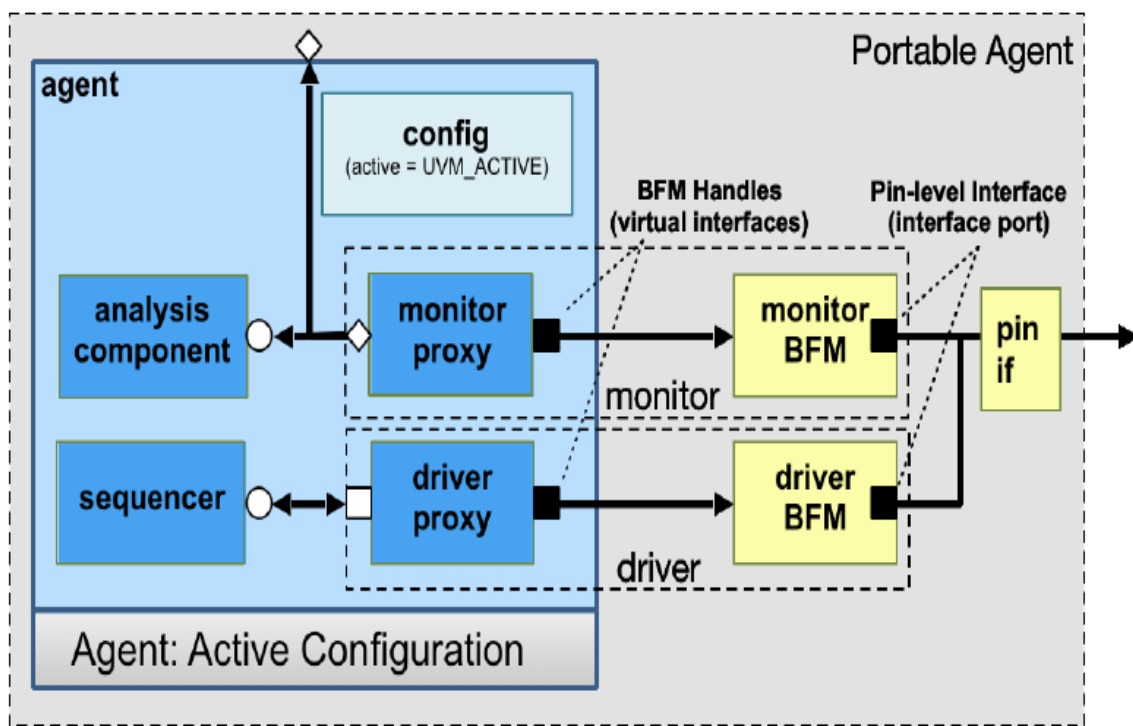


Figure 6 : APB Agent in Active Mode (Reference: UVM Cookbook)

- A UVM agent is a verification component "kit" for a given logical interface such as APB or USB. The agent includes a SystemVerilog interface encapsulating the corresponding set of interface signals, two SystemVerilog interfaces representing the monitor and driver BFM, and a SystemVerilog package including the various classes

that make up the overall agent class component.

- The agent class itself is a container class for a sequencer, a driver proxy and a monitor proxy, plus any other verification components deemed relevant such as a functional coverage collector or a scoreboard. Proxies are simply class objects providing the same API as "normal" class objects. The driver proxy and monitor proxy communicate with the rest of a UVM testbench in the usual way, while also accessing respectively the driver and monitor BFM interface via a virtual interface handle.
- A complete monitor is thus composed of the monitor proxy and monitor BFM working together as a pair, and the same is true for a driver. The agent also has an analysis port that is connected to the analysis port of the monitor, enabling a user to connect external analysis components to the agent without needing to know its internal structure.
- The agent is the lowest level hierarchical block in a testbench and its exact structure depends on its configuration, which can vary from one test to another per the agent configuration object. The classes and interfaces together constitute a portable, or reusable Agent.

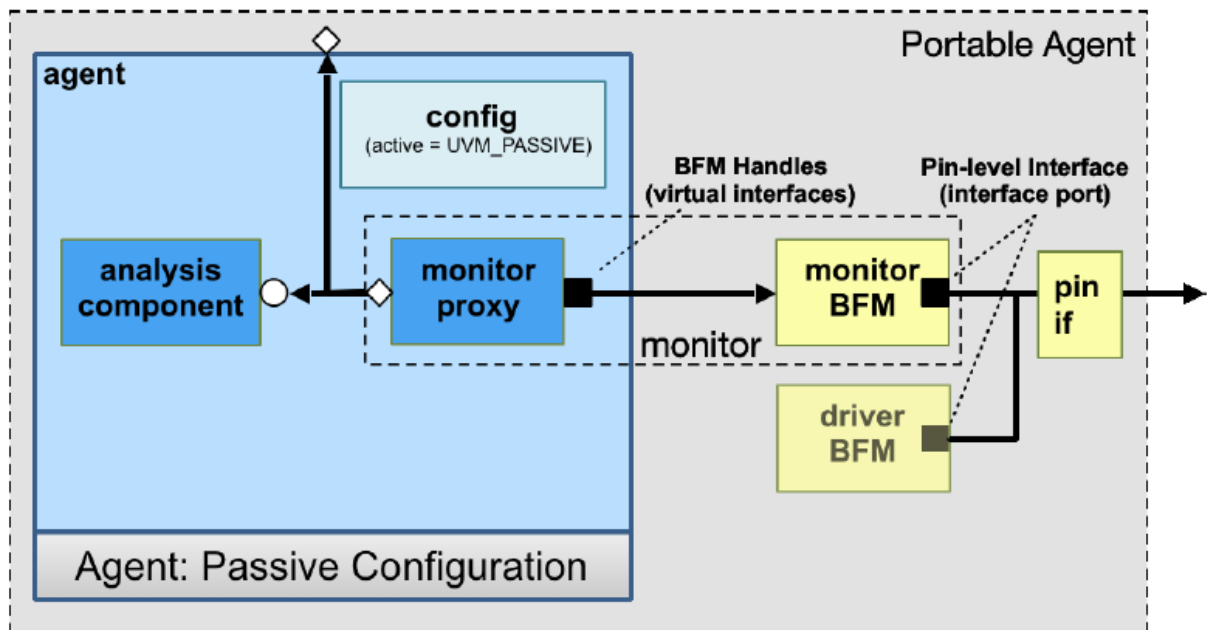


Figure 7 : APB agent in Passive configuration (Source: UVM Cookbook)

Let's examine how an APB agent is composed, configured, built, and connected. The BFM's define tasks and functions to interact with the signals in the apb_if pin interface. The driver and monitor proxies do not directly access the pins, which is to be kept local to the BFM's.

By convention, UVM agent configuration classes have a data member of type `uvm_active_passive_enum` that defines whether the agent is active (`UVM_ACTIVE`) with the sequencer and driver proxy constructed, or passive (`UVM_PASSIVE`) with neither the driver proxy nor sequencer constructed. This parameter is called `active` and by default set to `UVM_ACTIVE`.

3.1.3.4 APB Driver

- The UVM driver is responsible for communicating at the transaction level with the sequence via TLM communication with the sequencer and converting between the `sequence_item` on the transaction side and pin-level activity in communicating with the DUT via a virtual interface.
- As the name implies, drivers typically get a `sequence_item` and use that information to drive signals to the DUT and may, in certain applications, also receive a pin-level response from the DUT and convert it back into a `sequence_item` for the sequence to complete the transaction.
- A driver may also function as a "responder" (i.e. in "slave mode") in which the driver reacts to pin-level activity in the interface to communicate with a sequence that then sends a response transaction back to the driver to complete the protocol transaction. When its agent is configured to be in passive mode, the driver is, by definition, not instantiated in the agent.

3.1.3.5 APB Monitor

- The first task of the analysis portion of the testbench is to monitor activity on the DUT.
- A Monitor, like a Driver, is a constituent of an agent. A monitor component is similar to a driver component in that they both perform a translation between actual signal activity and an abstract representation of that activity.
- The key difference between a Monitor and a Driver is that a Monitor is always passive. It does not drive any signals on the interface.

- When an agent is placed in passive mode, the Monitor continues to execute.
- A Monitor communicates with DUT signals through a virtual interface, and contains code that recognizes protocol patterns in the signal activity. Once a protocol pattern is recognized, a Monitor builds an abstract transaction model representing that activity, and broadcasts the transaction to any interested components.

3.1.3.6 APB Sequences

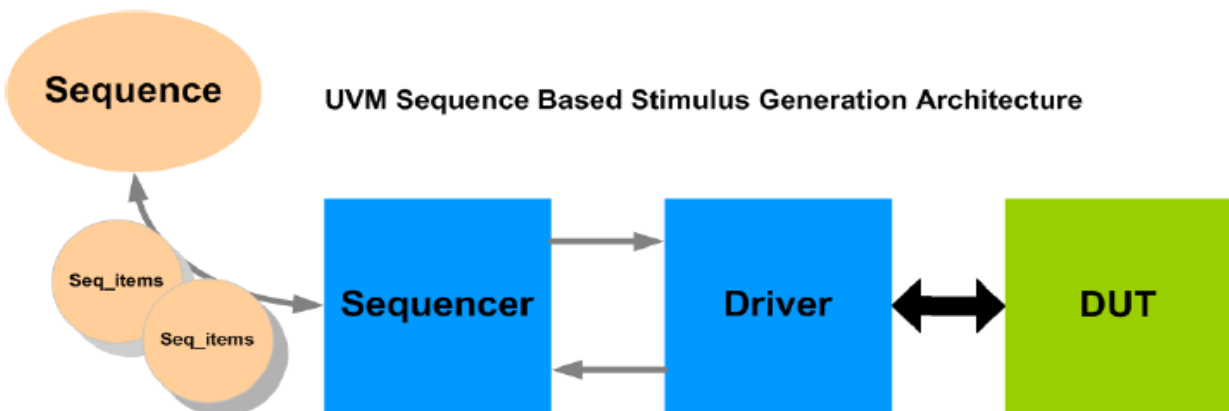


Figure 8 : UVM Sequence (Mentor's Verification Academy)

- A sequence is an example of what software engineers call a 'functor', in other words it is an object that is used as a method. An UVM sequence contains a task called body. When a sequence is used, it is created, then the body method is executed, and then the sequence can be discarded.
- Unlike an uvm_component, a sequence has a limited simulation life-time and can therefore can be described as a transient object. The sequence body method can be used to create and execute other sequences, or it can be used to generate sequence_item objects which are sent to a driver component, via a sequencer component, for conversion into pin-level activity or it can be used to do some combination of the two.
- The sequence_item objects are also transient objects, and they contain the information that a driver needs in order to carry out a pin level interaction with a DUT. When a response is generated by the DUT, then a sequence_item is used by the driver to pass the response information back to the originating sequence, again via the sequencer. Creating and executing other sequences is effectively the same as being able to call conventional subroutines, so complex functions can be built up by chaining together

simple sequences.

3.1.3.7 UVM Config DB

- The `uvm_config_db` class is the recommended way to access the resource database. A resource is any piece of information that is shared between two or more components or objects. Use `uvm_config_db::set` to put information into the database and use `uvm_config_db::get` to retrieve information from the database. The `uvm_config_db` class is a type-parameterized class, and consequently the database behaves as if it were partitioned into many type-specific "mini databases."
- There are no limitations on the type parameter, which can be a class, a `uvm_object`, a built-in type like a bit, byte, or a virtual interface, etc.
- There are two typical uses of the `uvm_config_db`. The first is to pass virtual interfaces from the HDL/DUT domain to the test, and the second is to pass configuration objects down through the testbench hierarchy.

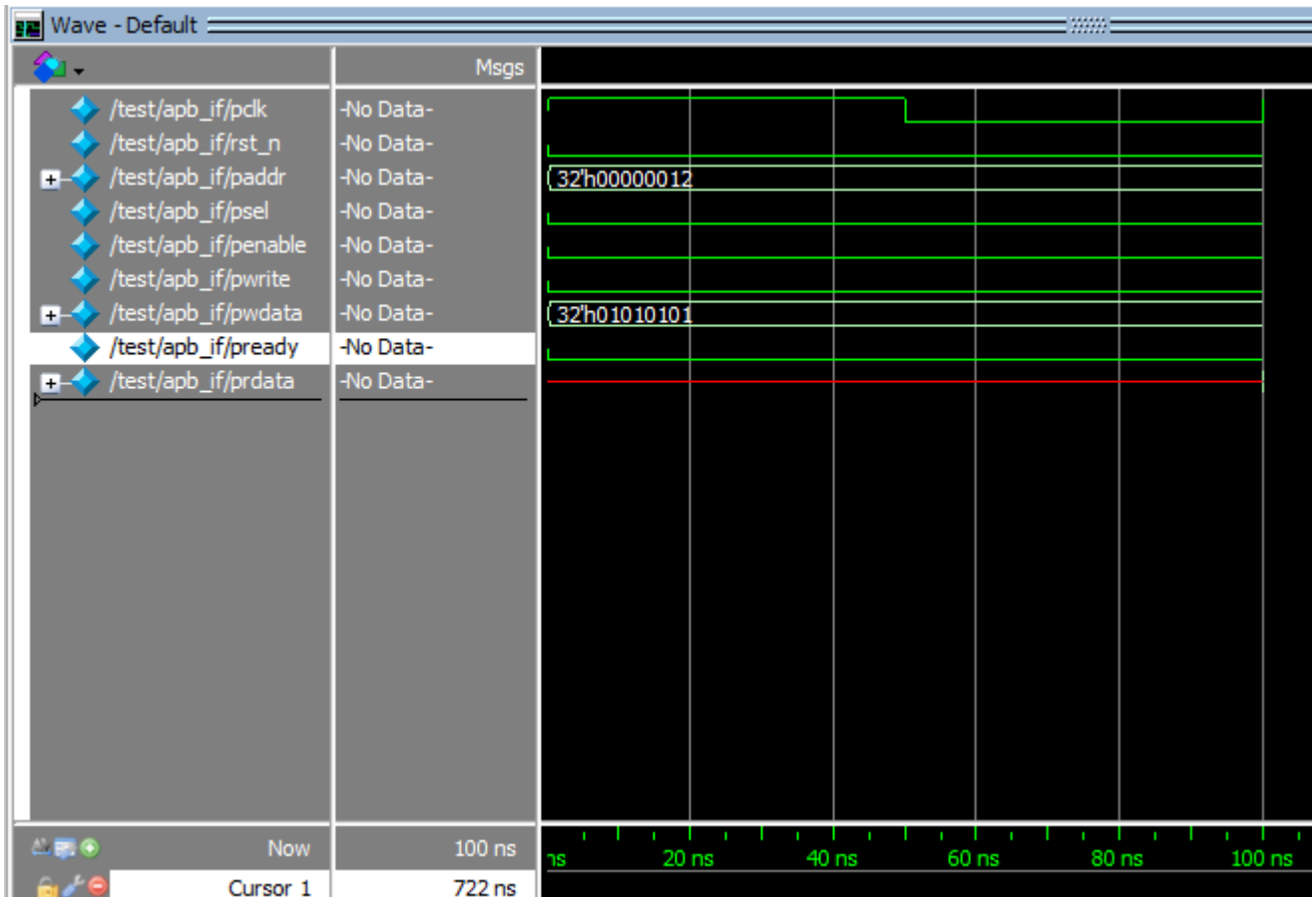
Chapter 4

IMPLEMENTATION RESULTS

This chapter gives the result analysis of implemented APB protocol design through its Verification Environment by using coverage driven functional verification methodology.

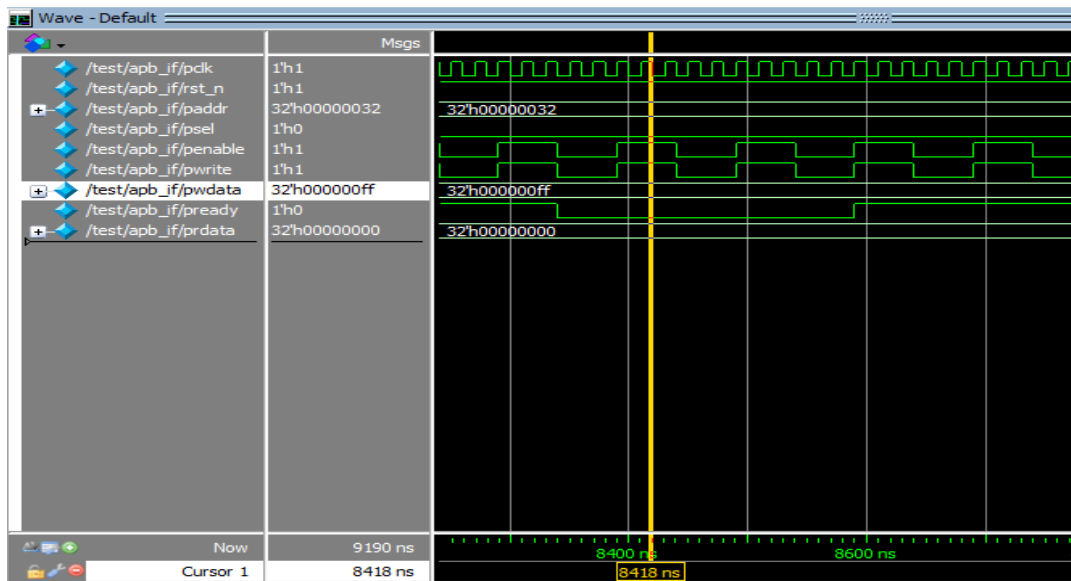
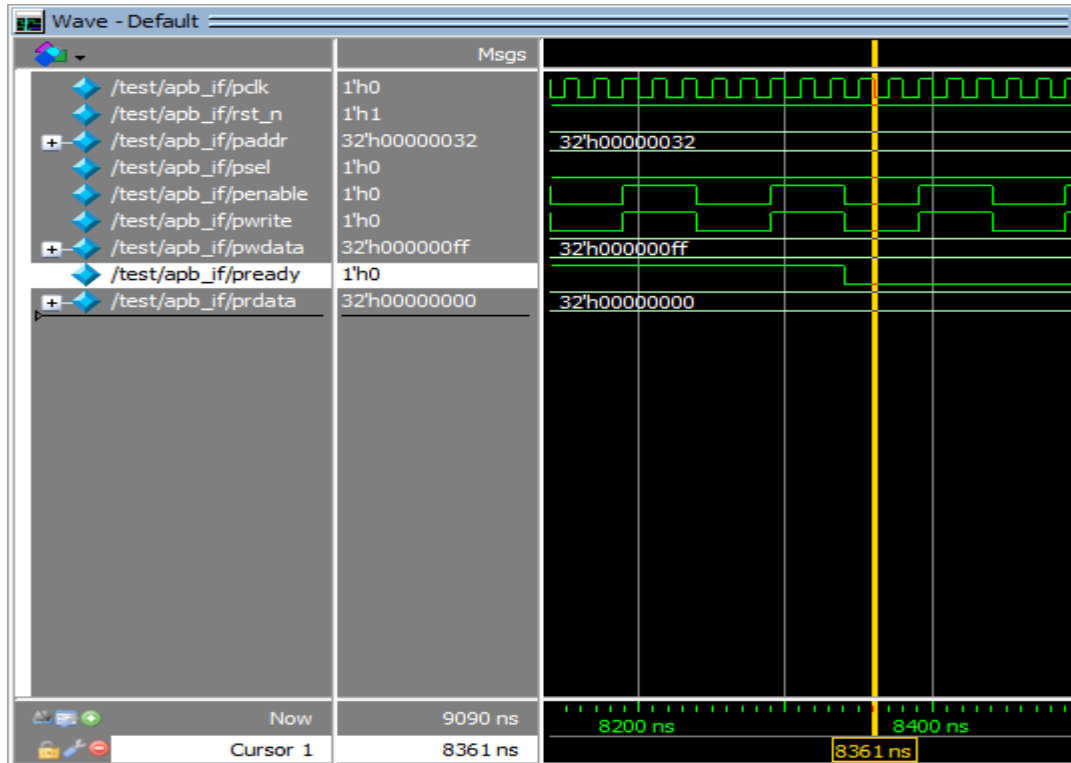
4.1 APB Protocol Transfers (Initial Conditions)

APB Protocol initial conditions simulation log is given below. PSEL,PENABLE,PWRITE, PWDATA are all low. We can clearly see garbage value before reset.

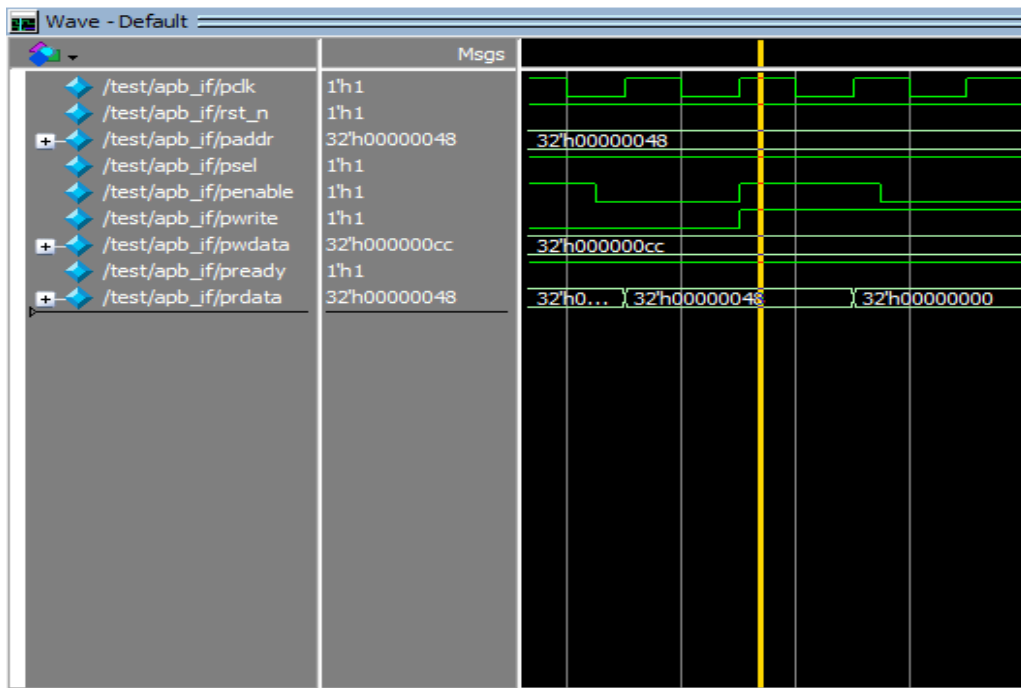


4.2 APB Protocol Transfers (after resetting)

We can clearly see no data being written on slave after resetting.



4.3 APB Protocol Implementation (After enabling Slave)



Above figure shows the working APB slave after enabling all the required signals through simulation and transcript log report.

4.4 UVM Implementation

--- UVM Report Summary ---

** Report counts by severity

UVM_INFO : 485

UVM_WARNING : 0

UVM_ERROR : 0

UVM_FATAL : 0

** Report counts by id

[] 160

[APB_AGENT] 1

[APB_DRIVER] 80

[APB_MONITOR] 80

[APB_SCOREBOARD] 80

[APB_SUBSCRIBER] 80

[Questa UVM] 2

[RNTST] 1

[TEST_DONE] 1

** Note: \$finish : C:/questasim_10.4e/win32/./verilog_src/uvm-1.1d/src/base/uvm_root.svh(430)

Time: 6490 ns Iteration: 54 Instance: /test

```
UVM_INFO @ 6190: uvm_test_top.env.agt.drv [APB_DRIVER ] Got Transaction pwrite=READ paddr=61 data=6b
UVM_INFO @ 6230: uvm_test_top.env.agt.mon [APB_MONITOR] Got Transaction pwrite=READ paddr=61 data=61
UVM_INFO apb_subscriber.sv(23) @ 6230: uvm_test_top.env.apn_subscriber_h [APB_SUBSCRIBER] Subscriber received tx pwrite=READ paddr=61 data=61
UVM_INFO apb_scoreboard.sv(43) @ 6230: uvm_test_top.env.scb [APB_SCOREBOARD] ----- :: READ DATA Match :: -----
UVM_INFO apb_scoreboard.sv(44) @ 6230: uvm_test_top.env.scb [ ] Addr: 61
UVM_INFO apb_scoreboard.sv(45) @ 6230: uvm_test_top.env.scb [ ] Expected Data: 61 Actual Data: 61
UVM_INFO @ 6270: uvm_test_top.env.agt.drv [APB_DRIVER ] Got Transaction pwrite=WRITE paddr=93 data=ff
UVM_INFO @ 6310: uvm_test_top.env.agt.mon [APB_MONITOR] Got Transaction pwrite=WRITE paddr=93 data=ff
UVM_INFO apb_subscriber.sv(23) @ 6310: uvm_test_top.env.apn_subscriber_h [APB_SUBSCRIBER] Subscriber received tx pwrite=WRITE paddr=93 data=ff
UVM_INFO apb_scoreboard.sv(37) @ 6310: uvm_test_top.env.scb [APB_SCOREBOARD] ----- :: WRITE DATA :: -----
UVM_INFO apb_scoreboard.sv(38) @ 6310: uvm_test_top.env.scb [ ] Addr: 93
UVM_INFO apb_scoreboard.sv(39) @ 6310: uvm_test_top.env.scb [ ] Data: ff
UVM_INFO @ 6350: uvm_test_top.env.agt.drv [APB_DRIVER ] Got Transaction pwrite=WRITE paddr=32 data=ff
UVM_INFO @ 6390: uvm_test_top.env.agt.mon [APB_MONITOR] Got Transaction pwrite=WRITE paddr=32 data=ff
UVM_INFO apb_subscriber.sv(23) @ 6390: uvm_test_top.env.apn_subscriber_h [APB_SUBSCRIBER] Subscriber received tx pwrite=WRITE paddr=32 data=ff
UVM_INFO apb_scoreboard.sv(37) @ 6390: uvm_test_top.env.scb [APB_SCOREBOARD] ----- :: WRITE DATA :: -----
UVM_INFO apb_scoreboard.sv(38) @ 6390: uvm_test_top.env.scb [ ] Addr: 32
UVM_INFO apb_scoreboard.sv(39) @ 6390: uvm_test_top.env.scb [ ] Data: ff
UVM_INFO verilog_src/uvm-1.1d/src/base/uvm_objection.svh(1268) @ 6490: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
```

UVM analysis was first performed using open source tool i.e. EDA PLAYGROUND and later, it was simulated using Mentor's Questasim 10.4e




































```
# KERNEL: UVM_INFO /home/build/vlib1/vlib/uvm-1.2/src/base/uvm_report_server.svh(856) @ 6490: reporter [UVM/REPORT/SERVER]
# KERNEL: --- UVM Report Summary ---
# KERNEL:
# KERNEL: ** Report counts by severity
# KERNEL: UVM_INFO : 484
# KERNEL: UVM_WARNING : 0
# KERNEL: UVM_ERROR : 0
# KERNEL: UVM_FATAL : 0
# KERNEL: ** Report counts by id
# KERNEL: [] 160
# KERNEL: [APB_AGENT] 1
# KERNEL: [APB_DRIVER ] 80
# KERNEL: [APB_MONITOR] 80
# KERNEL: [APB_SCOREBOARD] 80
# KERNEL: [APB_SUBSCRIBER] 80
# KERNEL: [RNTST] 1
# KERNEL: [TEST_DONE] 1
# KERNEL: [UVM/RELNOTES] 1
# KERNEL:
# RUNTIME: Info: RUNTIME_0068 uvm_root.svh ($21): $finish called.
# KERNEL: Time: 6490 ns, Iteration: 56, Instance: /test, Process: @INITIAL#50_3@.
# KERNEL: stopped at time: 6490 ns
# VSIM: Simulation has finished. There are no more test vectors to simulate.
..-----
```

Figure 4.9:- Logs of Baud Rate test

4.5 Coverage Report

Coverage is used to measure tested and untested portions of the design. It defines the percentage of verification done.

4.5.1 Coverage with data & address bins set in limits

Covergroups							
Name	Class Type	Coverage	Goal	% of Goal	Status	Indu	
/testbench_sv_uni...							
TYPE cover_b...	apb_subscriber	100.0%	100	100.0%		✓	
CVP cover...	apb_subscriber	100.0%	100	100.0%		✓	
bin a[0]		6	1	100.0%		✓	
bin a[1]		2	1	100.0%		✓	
bin a[2]		5	1	100.0%		✓	
bin a[3]		5	1	100.0%		✓	
bin a[4]		3	1	100.0%		✓	
bin a[5]		5	1	100.0%		✓	
bin a[6]		7	1	100.0%		✓	
bin a[7]		4	1	100.0%		✓	
bin a[8]		7	1	100.0%		✓	
bin a[9]		7	1	100.0%		✓	
bin a[10...]		4	1	100.0%		✓	
bin a[11...]		4	1	100.0%		✓	
bin a[12...]		7	1	100.0%		✓	
bin a[13...]		6	1	100.0%		✓	
bin a[14...]		4	1	100.0%		✓	
bin a[15...]		4	1	100.0%		✓	
CVP cover...	apb_subscriber	100.0%	100	100.0%		✓	
bin d[0]		4	1	100.0%		✓	
bin d[1]		4	1	100.0%		✓	
bin d[2]		1	1	100.0%		✓	
bin d[3]		3	1	100.0%		✓	
bin d[4]		3	1	100.0%		✓	
bin d[5]		8	1	100.0%		✓	
bin d[6]		7	1	100.0%		✓	
bin d[7]		5	1	100.0%		✓	
bin d[8]		9	1	100.0%		✓	
bin d[9]		6	1	100.0%		✓	
bin d[10...]		5	1	100.0%		✓	
bin d[11...]		3	1	100.0%		✓	
bin d[12...]		5	1	100.0%		✓	
bin d[13...]		6	1	100.0%		✓	
bin d[14...]		5	1	100.0%		✓	
bin d[15...]		6	1	100.0%		✓	

4.5.2 Not covering entire range (out of limits)

/testbench_sv_unit/apb_subscriber							
-	TYPE cover_bus3	apb_subscriber	44.4%	100	44.4%	<div><div></div></div>	✓
+	CVP cover_bus3::data	apb_subscriber	33.3%	100	33.3%	<div><div></div></div>	✓
+	CVP cover_bus3::addr	apb_subscriber	55.5%	100	55.5%	<div><div></div></div>	✓
-	TYPE cover_bus	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus::data	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
-	TYPE cover_bus2	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus2::data	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus2::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓

4.5.3 Improvement in Coverage by increasing bins

Covergroups							
Name	Class Type	Coverage	Goal	% of Goal	Status	Included	
/testbench_sv_unit/apb_subscriber							
-	TYPE cover_bus3	apb_subscriber	61.1%	100	61.1%	<div><div></div></div>	✓
+	CVP cover_bus3::data	apb_subscriber	66.6%	100	66.6%	<div><div></div></div>	✓
+	CVP cover_bus3::addr	apb_subscriber	55.5%	100	55.5%	<div><div></div></div>	✓
-	TYPE cover_bus	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus::data	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
-	TYPE cover_bus2	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus2::data	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
+	CVP cover_bus2::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓

4.5.3 Adding ignore bins

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/testbench_sv_unit/apb_subscriber						
TYPE cover_bus3	apb_subscriber	61.1%	100	61.1%	<div><div></div></div>	✓
CVP cover_bus3::data	apb_subscriber	66.6%	100	66.6%	<div><div></div></div>	✓
CVP cover_bus3::addr	apb_subscriber	55.5%	100	55.5%	<div><div></div></div>	✓
TYPE cover_bus	apb_subscriber	86.4%	100	86.4%	<div><div></div></div>	✓
CVP cover_bus::data	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
CROSS cover_bus::CC	apb_subscriber	59.2%	100	59.2%	<div><div></div></div>	✓
CVP cover_bus::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
bin a1		7	1	100.0%	<div><div></div></div>	✓
bin a2		8	1	100.0%	<div><div></div></div>	✓
bin a3		21	1	100.0%	<div><div></div></div>	✓
bin a4		32	1	100.0%	<div><div></div></div>	✓
ignore_bin a5		4	-	-	<div><div></div></div>	✓
TYPE cover_bus2	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
CVP cover_bus2::data	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
CVP cover_bus2::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓

4.5.4 Goal Test (deliberately kept under 90 for cover_bus3)

Name	Class Type	Coverage	Goal	% of Goal	Status	Included
/testbench_sv_unit/apb_subscriber						
TYPE cover_bus3	apb_subscriber	61.1%	100	61.1%	<div><div></div></div>	✓
CVP cover_bus3::data	apb_subscriber	66.6%	100	66.6%	<div><div></div></div>	✓
CVP cover_bus3::addr	apb_subscriber	55.5%	100	55.5%	<div><div></div></div>	✓
TYPE cover_bus	apb_subscriber	86.4%	100	86.4%	<div><div></div></div>	✓
CVP cover_bus::data	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
bin d1[0]		4	1	100.0%	<div><div></div></div>	✓
bin d1[1]		4	1	100.0%	<div><div></div></div>	✓
bin d1[2]		1	1	100.0%	<div><div></div></div>	✓
bin d1[3]		3	1	100.0%	<div><div></div></div>	✓
bin d1[4]		3	1	100.0%	<div><div></div></div>	✓
bin d1[5]		8	1	100.0%	<div><div></div></div>	✓
bin d1[6]		7	1	100.0%	<div><div></div></div>	✓
bin d1[7]		5	1	100.0%	<div><div></div></div>	✓
bin d1[8]		9	1	100.0%	<div><div></div></div>	✓
bin d1[9]		6	1	100.0%	<div><div></div></div>	✓
bin d1[10]		5	1	100.0%	<div><div></div></div>	✓
bin d1[11]		3	1	100.0%	<div><div></div></div>	✓
bin d1[12]		5	1	100.0%	<div><div></div></div>	✓
bin d1[13]		6	1	100.0%	<div><div></div></div>	✓
bin d1[14]		5	1	100.0%	<div><div></div></div>	✓
bin d1[15]		6	1	100.0%	<div><div></div></div>	✓
bin d2		5	1	100.0%	<div><div></div></div>	✓
bin d3		21	1	100.0%	<div><div></div></div>	✓
bin a4		34	1	100.0%	<div><div></div></div>	✓
CROSS cover_bus::CC	apb_subscriber	59.2%	100	59.2%	<div><div></div></div>	✓
CVP cover_bus::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
bin a1		7	1	100.0%	<div><div></div></div>	✓
bin a2		8	1	100.0%	<div><div></div></div>	✓
bin a3		21	1	100.0%	<div><div></div></div>	✓
bin a4		32	1	100.0%	<div><div></div></div>	✓
ignore_bin a5		4	-	-	<div><div></div></div>	✓
TYPE cover_bus2	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓
CVP cover_bus2::addr	apb_subscriber	100.0%	100	100.0%	<div><div></div></div>	✓

4.6 Summary

This chapter mentions the simulation, transcript log file, and coverage analysis results of various parameters. Apart from that, various other parameters like Code Coverage, Line Coverage, FSM Coverage, Toggle Coverage can be calculated.

Chapter 5

CONCLUSIONS AND SCOPE FOR FUTURE WORK

5.1 Conclusion

- In this dissertation, AMBA APB Protocol is implemented using Verilog.
- Verification Environment is implemented using SV & UVM methodology. Functional and coverage driven verification of **AMBA APB features** is implemented, as mentioned in Arm Inc. datasheet
- Various testcases are written & verified to achieve desire functional verification. Different options are exercised for coverage collection. This is exactly like mimicking the normal industry methods which are used to test complex DUTs. Verification environment is coded in system Verilog and UVM methodology by using Mentor Graphics tool Questasim-10.4e.
- All stages are tested using EDA Playground, an open source tool.

5.2 Scope for Future Work

- AMBA APB VIP can be stored as an independent VIP for interfacing processors with low bandwidth applications.
- It can also be bridged with several communication protocols like AHB, ASB, ACE, AXI, SPI, I2C for achieving a better communication.
- It can be used as a Verification IP for smaller projects across vendors in the industry.

REFERENCES

- [1] Datasheet. *APB Protocol Specification*. ARM Inc., IHI0024C, August 2008.
- [2] Accellera. *Universal Verification Methodology (UVM) 1.2 User's Guide*. December 8, 2018.
- [3] Litterick, Mark, and Marcus Harnisch. "Advanced UVM Register Modeling." Proc. Design and Verification Conference & Exhibition Europe (DVCon Europe). 2014.
- [4] Graphics, Mentor. *UVM cookbook*, Online Methodology Documentation from the Verification Methodology Team. (2014).
- [5] Sutherland, Stuart, Simon Davidmann, and Peter Flake. *System Verilog for Design Second Edition: A Guide to Using System Verilog for Hardware Design and Modeling*. Springer Science & Business Media, 2006.
- [6] Spear, Chris, Tombush. *System Verilog for Verification: a guide to learning the testbench Language Features*. Springer Science & Business Media, 2012.
- [7] Mehta, Ashok B. *System Verilog Assertions and Functional Coverage*. Springer New York, NY, 2014. 9-28.

APPENDIX A

Plagiarism Report

Report_1

ORIGINALITY REPORT

19%

SIMILARITY INDEX

17%

INTERNET SOURCES

4%

PUBLICATIONS

16%

STUDENT PAPERS

PRIMARY SOURCES

1

www.quicklogic.com

Internet Source

8%

2

Submitted to Visvesvaraya Technological University

Student Paper

2%

3

Submitted to Institute of Technology, Nirma University

Student Paper

1%

4

Myoung-Keun You, Gi-Yong Song.
"Implementation of a simple emulator platform
with limited resources", TENCON 2007 - 2007
IEEE Region 10 Conference, 2007

Publication

1%

5

tutorial.cytron.com.my

Internet Source

1%

6

-the-gender-pay-aauw.org

Internet Source

1%

7

Submitted to Engineers Australia

Student Paper

1%