A Comparative Study of Directed Testing vs UVM-Based Random Verification

Thesis report submitted in partial fulfilment of the requirement for the Award of the Degree of

MASTER OF TECHNOLOGY

in VLSI DESIGN

Submitted By

KONIKA GUPTA

6024620010

Under Supervision of

Dr. RAHUL UPADHYAY

(Associate Professor)

&

Dr. ASHU SHARMA

(Assistant Professor)

&

SAURABH SAXENA

(Engineering Manager)

Intel Technology India Pvt. Ltd.

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING & TECHNOLOGY

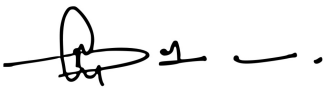
(A DEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

JANUARY – JUNE 2026

DECLARATION

I, **KONIKA GUPTA** hereby declare that the work presented in this Thesis Report entitled "**A Comparative Study of Directed Testing vs UVM-Based Random Verification"** in partial fulfilment of the requirement for the award of degree of Master of Technology (VLSI Design) submitted at **ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT**, Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under supervision of **Dr. RAHUL UPADHYAY** (Associate Professor) & **Dr. ASHU SHARMA** (Assistant Professor) and Industry Mentor **SAURABH SAXENA** from June, 2025 to June, 2026. The matter presented in this has not been submitted either in part or full to any other university or institute for the award of any other degree.

Konika Gupta
6024620010
Date:28/06/2026

 SAURABH SAXENA Engineering Manager Intel Technology India Pvt. Ltd. Date:28/06/2026	 Dr. RAHUL UPADHYAY Associate Professor Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date: 28/06/2026
	 Dr. ASHU SHARMA Assistant Professor Department of Electronics and Communication Engineering, Thapar Institute of Engineering & Technology, Patiala Date:28/06/2026

Tel: +91-80-6211 3000
Fax: +91-80-6280 9010

To Whomsoever It May Concern

WWID: 12307040

Employee Name: Konika Gupta

Internship Dates: 06/16/2025 to 06/12/2026

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,
For Intel Technology India Pvt. Ltd.

Harishkumar S
Authorized Signatory

Registered Office:
Intel Technology India Pvt Ltd.
#23-56P, Outer Ring Road, Devarabeesanahalli, Varthur Hobli, Bellandur Post
Bengaluru 560 103, Karnataka, India
CIN – U85110KA1997PTC021606

ACKNOWLEDGEMENT

This thesis wouldn't have been possible without the help and support of several people who played a key role in making it happen.

I want to give a heartfelt thanks to my supervisors **Dr. Rahul Upadhyay** and **Dr. Ashu Sharma**. Their support, encouragement, and extensive knowledge was very essential to me. They helped me all along during my research work and writing this dissertation, and for that matter, I am really thankful to them.

I would like to thank **Mr**. **Saurabh Saxena**, Manager of Team DDG at Intel Technology India Pvt. Ltd., for giving me the opportunity to work as an intern in his team. I would also like to thank him for his assistance, guidance, and understanding of my academic commitments. It was indeed a valuable and enjoyable experience because of him.

Mr. **Ajith Sudhindra**, SoC Verification Engineer, INTEL Technology India Pvt Ltd for being very patient and encouraging me throughout. The constructive criticisms offered by him at various stages of my work were truly thought provoking and enabled me to concentrate on my ideas. I am really thankful to him for all the discussions which helped me understand my work in detail.

Finally, my family and close friends. I wouldn’t be able to do this without the support of their love. And lastly, but not least, the one above all else, the omnipresent God, for giving me the willpower to push on, thank you so much everyone.

ABSTRACT

In this thesis, both directed testing and constrained random verification using the Universal Verification Methodology (UVM) approach are considered for verifying the functionality of an Advanced High-performance Bus Lite (AHB Lite) protocol design. With today's SoCs being very complex in nature, there is a great need to choose a proper verification methodology to ensure that all aspects comply with the protocol. This research provides a solution for this challenge by implementing both approaches on an AHB Lite slave memory design and then analyzing their performance based on certain quantitative metrics..

The directed test methodology validates basic AHB transactions such as single transactions, WRAP4 burst increments, and write then read transactions with manual stimulus with static address and known data patterns. The methodology based on UVM-Universal Verification Methodology uses a complete verification environment with a transaction level sequence item with protocol constrained, agent with driver and monitor, automation reference model scoreboard, and functional coverage collection from covergroups. Constrained random test stimuli generation covers the legitimate transaction space in different seeds, aiming at scenarios not anticipated in directed tests.

Both approaches are looked at using a few different factors like functional coverage percentage, code coverage, simulation throughput, test code volume, debugging complexity, and how well the testbench architecture scales. The results show that directed testing gives us a clear and easy way to debug known scenarios with just a bit of infrastructure. On the other hand, UVM-based constrained random verification manages to cover a lot more cases with fewer lines of test code, plus it brings to light some corner case behaviors that might get missed in tests that are written by hand. Overall, these findings are pretty useful for verification engineers when they're picking the right method for AHB-based digital IP verification in today's SoC development projects.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Background and Motivation

Demand for efficient and high performance and low power embedded system has led to the evolution of highly integrated System on Chip architecture where multiple functional blocks such as processor cores, memory controllers and peripheral interfaces interact via on chip bus based protocols. Amongst those AHB (Advanced High-performance Bus) plays an important role in providing high speed data transfers with features such as single cycle bus arbitration, pipelining of address/data phase and multiple master capability [1]. It is due to such features that AHB is widely used in microcontroller based and embedded computer systems.

With increasing integration complexity, the complexity of the bus based interaction grows significantly. AHB supports several types of data transfers such as single burst, incrementing and wrapping bursts and many different sizes of transfers and responses. Interactions between master and slave via control lines such as address, write control, transfer type and ready response lines should conform to the protocol rules very precisely. Otherwise it may result in incorrect data transfers, deadlocks or reduced performance.

In actual design processes, the verification engineers usually initiate their testing efforts by using the directed test methodology to verify basic operations like basic read and writes. While this methodology will help in exercising all the basic functions, this largely relies on the creative thinking of the designer, who can think of the many different cases which need to be checked. These complex cases including combinations of burst lengths and different arbitration schemes may get unnoticed if they are not specifically identified. This is even more critical in the case of a design which scales up and uses multiple configurations of the AHB interface.

The modern day verification process stresses on automation, reuse and coverage. UVM offers a modular structure in which the process of generation of the stimulus, the monitoring and checking of these are done separately[3]. The constrained random methodology will help in automatically generating a wide variety of transactions within the legal boundary of the AHB protocol.

The study will be guided by the goal of carrying out a comparison between the methods of directed testing and UVM-based random verification for AHB protocol. This will be done using the same AHB design in both cases with regard to issues like functional coverage, protocol compliance, and complexity in the verification process. The purpose of this research is to establish which method is more efficient and reliable.

1.2 Problem Statement

The increasing level of complexity in the design of SoCs has turned functional verification into an important yet very time-consuming stage in the development process. In case of a bus architecture, correct communication between master and slave devices is crucial for the reliability of the whole system. AMBA AHB protocol is distinguished by its pipeline implementation, burst capability, and possibility of using multiple masters; thus, it defines various functional scenarios, which should be verified in order to avoid any possible functional problems in silicon.

In traditional verification setup, directed testing is widely used to verify the correct operation of particular AHB transactions, such as single reads, single writes, and certain burst sequences. Although directed testing proves that the expected operation of some test scenario works properly, it does not ensure that all the possible combinations of transfer types, lengths, and boundaries are performed during verification. The success of the directed testing heavily depends on the manually written test cases, which could leave some unusual interactions between the protocol elements unconsidered.

This research compares two verification techniques utilized for the AHB protocol verification, namely, directed verification technique and randomized verification based on the UVM. The second approach involves constrained random stimulus creation and coverage-driven verification process. While this approach allows exploring a greater number of possible scenarios, it adds additional complexity in terms of environment creation, specification of constraints, and debugging process. Moreover, development of a reusable testbench based on the UVM approach might not be worthwhile for simpler design projects or projects with tight deadlines.

The main problem to be discussed in this thesis is related to the lack of the comparative analysis of the two approaches in terms of their use for the verification of the AHB protocol. It is important to conduct a comparison to understand how these approaches work with regards to coverage, detection of bugs, complexity of implementation, and scalability. Testing both approaches in the context of the AHB design under test and applying predefined verification criteria will help to define which technique is better for verifying AHB-based IPs in modern SoCs.

1.3 Introduction to AMBA AHB Protocol

The AMBA Advanced High-performance Bus (AHB) is a popular on-chip bus communication protocol used in embedded and microcontroller System-on-Chip architectures. It is designed to facilitate high-speed data transmission from multiple bus masters and slaves in the same system. The bus communication protocol offers a well-defined method for communication through a bus interface that includes address,

control, and data lines. This protocol is useful for the efficient interaction between components such as processors, memory blocks, and peripheral controllers.

The AMBA AHB protocol is implemented using a pipelining technique where the address phase and the data phase overlap to increase efficiency. It enables single and multiple data transfers using bursts, which can be either incremental or wrapping bursts. These different types of transfers and the associated responses and control fields ensure reliable transfer of data between the master and the slave devices. The protocol also has arbiters that control access when there are more than one masters trying to access the bus. Figure 1 below shows the AHB bus architecture with multiple masters accessing multiple slaves through an arbiter, multiplexer, and a decoder.

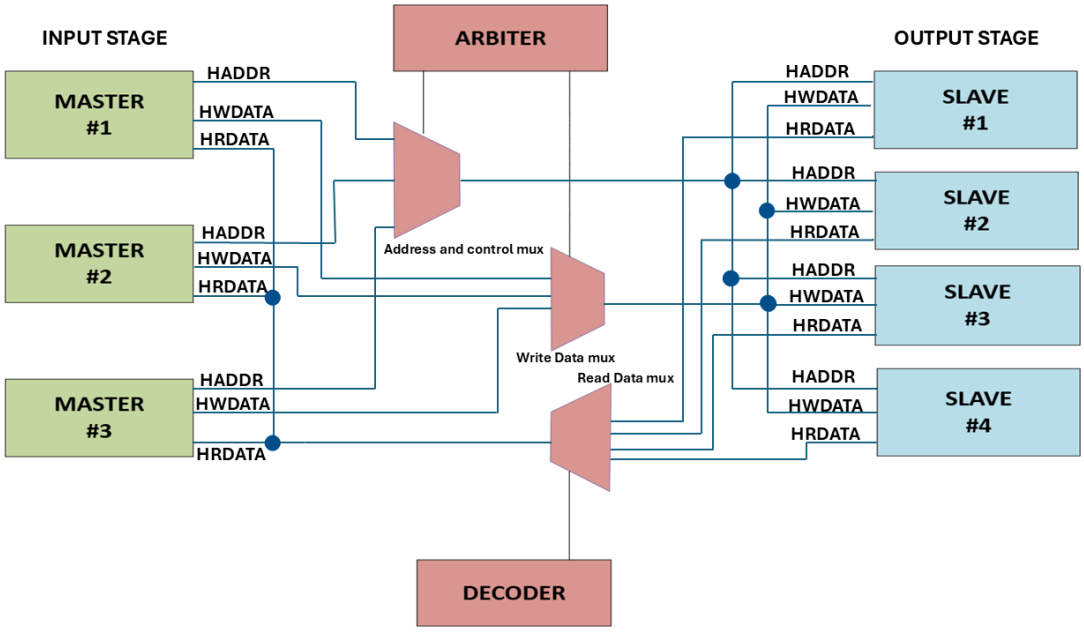


Figure 1.1: AHB BUS ARCHITECTURE

Further, the AHB protocol provides specific timing dependencies between the signals, including readiness and response signals, to maintain the synchronization of the bus. These features make the protocol suitable for applications where there is a need for the tradeoff between performance and complexity. However, the combination of several types of transactions, bursts, and arbitration makes the set of functional states to be verified much more extensive [2]. Thus, it is important to gain a profound insight into the architecture and functionality of the AHB protocol before developing the verification strategies in the thesis.

1.4 Verification Challenges in AHB Protocol validation

There are several verification challenges related to the AHB protocol verification which are caused by protocol complexity and a variety of transactions [6]. The AHB protocol has such features as: pipeline

address/data phase, different types of transactions, burst transaction, bus arbitration among several masters [6]. Every feature results in numerous interactions of signals that should correspond to the specification [6] precisely. In order to make sure that all sequences are functioning correctly, the verification process should be systematic.

Proper burst transactions is one of the most important challenges because of different burst sizes and their alignment. Address for increment bursts and wrap bursts should meet certain requirements not to go beyond boundaries.

In the case when ready signal is not asserted and wait state is started, special approach is needed because timing becomes very crucial here. The mistakes in control signals can cause the data corruption or failure of the whole transaction.

An additional problem comes in the form of arbitration situations within multi master configurations. In the situation where there are more than two masters that want to access the bus, it has to be ensured that the arbitration method ensures a deterministic way of granting bus access. Verification has to guarantee that there will be no deadlock or race around situation as well as maintaining data consistency when there is a switch of masters. This can be very complex and might not be entirely verified with directed testing only.

It is also difficult to have coverage of all the possible transfer types, lengths and responses. Identifying special cases including bursts aligned to the boundary, back-to-back transfers and errors requires a verification environment that will explore many different stimulus combinations. This is the real essence of why the need for looking into the verification approaches for the AMBA based AHB protocol becomes very important indeed.

1.5 Objectives of the Thesis

The principal objective of this thesis is to conduct a systematic quantitative analysis of directed testing and UVM based constrained random verification techniques as applied to an AHB-Lite bus protocol design. The purpose is to establish the superiority of one technique over another with respect to functional coverage, bug detection capabilities, and scalability in the context of current SoC verification methodologies.

To achieve this objective, the thesis tests an AHB slave module under test using the above methodologies. A directed test bench is employed to test critical AHB functionality such as single transfer, WRAP4 bursts and write followed by read consistency using hand coded stimulus with fixed addresses and data. The paper employs a UVM constrained random verification environment which utilizes transaction level modeling to generate randomized stimulus. This technique automatically verifies the results generated using scoreboard and aims for closure through coverage. Using this technique, the same functional area

can be explored using solver generated stimuli from random seed numbers, which is quite interesting! The two approaches are compared against several quantitative metrics including functional coverage percentage, code coverage, simulation time, number of lines in the test code, debugging effort and architectural scalability.

1.6 Scope and Limitations

This research has a scope that covers the functional verification of an AHB based digital design using two different techniques of verification, directed testing and UVM based random verification. The research will be restricted to AHB protocol only and will not include any other AMBA protocols like APB or AXI[2]. This research involves designing two different verification environments for the same AHB DUT and comparing the results of both the methodologies considering some defined verification metrics.

Implementation scope covers AHB transaction modeling, stimulus generation for single and burst transactions, bus activity observation and protocol checking. Functional and code coverage would be considered in order to measure the coverage achieved by each technique.[7] The comparison will take place under a simulation based verification environment using System Verilog. Hardware implementation or post-silicon validation is not included in this research scope.

Moreover, the study is concentrated on a particular AHB interface configuration despite the possibility of multiple masters and complicated arbitration mechanisms. While such implementation of the protocol in this thesis is possible, the number of masters and slaves could be limited to make the design more simple. Performance will be evaluated through simulations and coverage results, but not through real time hardware performance.

Certainly, there are several limitations connected to the work. Results obtained depend on the quality of written directed tests and efficiency of the constraint modeling in the UVM environment[8]. Insufficient or poorly performed constraint modeling can influence randomness and coverage. Similar situation could occur with the estimation of the development effort, which depends on the experience of the verification engineer.

Of course, the comparison takes place in an academic research field, which means that some challenges related to large regression suites and complex SoC subsystem integration are not covered. However, regardless of mentioned limitations, the purpose of this thesis is to give some insights regarding effectiveness of directed testing as opposed to random verification with UVM environment in case of AHB protocol verification.

CHAPTER 2: LITERATURE REVIEW

[1] ARM Limited's "AMBA 3 AHB-Lite Protocol Specification" [1] The paper presents a simplified AMBA 3 protocol called the AHB-Lite protocol. The protocol is intended to facilitate high bandwidth-low latency communication in embedded systems. In this paper, a discussion on the signals, transfer types, and timing of the protocol has been provided. This paper contains useful guidance on the design and verification of AHB-Lite bus connections that are commonly seen in SoC implementations.

[2] In "AMBA Specification Revision 2.0" [2], Introduction to AMBA bus standards is made by ARM limited through definition of the architecture and requirements of the standard. The standard gives the details about the functionality and behavior of the AHB, APB and ASB buses and has been instrumental in development of portable and reusable IPs..

[3] The "Universal Verification Methodology (UVM) 1.2 User's Guide" by Accellera [3] Introduction to UVM is a good introduction to UVM and serves as a guide to the UVM library. This guide helps verification engineers design testbenches that can be reused and made scalable using SystemVerilog. It briefly describes some of the UVM concepts such as sequence, component, configuration, and environment that make verification of complex digital designs easier.

[4] Accellera's "Universal Verification Methodology (UVM) 1.2 Reference Manual" [4] It gives a detailed view of the UVM class library. It covers all aspects including the syntax of UVM, class declaration, class variables, and methods that can be used in UVM. This documentation is a very valuable tool for engineers involved in constraint-random and coverage-based verification environments.

[5] In "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features" [5], SystemVerilog by C. Spear & G. Tumbush is a very detailed manual which will help you learn how to use SystemVerilog, including the verification part. You can find information on how to use such advanced features of the language as randomization, classes and assertions, as well as some techniques of building modern re-usable testbenches.

[6] J. Bergeron's "Writing Testbenches: Functional Verification of HDL Models" [6] is really an invaluable resource for those people who wish to learn how to create effective testbenches for their HDL models. The issues such as creation of stimuli, checking the output response, and usage of self-checking techniques are covered in this book. What I really like about Bergeron's work is that he combines theoretical information with practical examples.

[7] S. Rosenberg and K. A. Meade's "A Practical Guide to Adopting the Universal Verification Methodology (UVM)" [7] offers valuable information to companies and engineers that have decided to shift towards a methodology based on UVM. Topics covered range from planning to migration of

testbenches and best practices, always emphasizing on practical issues and maximization of reuse. The information provided is quite practical and explicit.

[8] "Comprehensive Functional Verification: The Complete Industry Cycle" by B. Wile, J. Goss, and W. Roesner [8] get an overview of the complete life cycle involved in functional verification of hardware design. It discusses some significant issues related to coverage measurement, bug tracking, verification planning, and integration of verification approaches, which make it quite useful as a reference guide for people wanting to get comprehensive validation.

[9] A. B. Mehta's "ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies" [9] provides a concise introduction to various verification methods such as simulation, emulation, and formal verification. It is primarily targeted at designs for ASIC and SoC applications where ensuring the quality of the product and improving efficiency of the design teams become highly significant. The book also describes ways to find and correct design errors in an early stage. The concepts are clearly explained and logically organized. The reader does not need to have a deep technical understanding of the topic.

[10] In "The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology" [10], UVM is presented clearly and with many examples by R. Salemi. This book explains the basic concepts of UVM in such a way that it becomes very simple for newbies who are migrating from the conventional verification techniques to the modern SystemVerilog based techniques. It's a very good guide for anyone willing to learn UVM.

[11] The "VCS User Guide" from Synopsys Inc. [11] is to assist you in making the most of the VCS simulator, providing you with useful information and hints on how to do things properly. The book covers all major aspects, starting from such fundamental activities as compilation of your source code, simulation process itself, debug of possible problems, and optimization of your SystemVerilog-based designs. You will definitely benefit from its reading and find it very helpful, particularly in complicated verification environment.

[12] Synopsys Inc.'s "Verdi User Guide and Tutorial" [12] This book shows you how to make use of a popular debugging software application when verifying digital circuits. The process of viewing waveforms, signal tracing, and examining source code are explained here and these are useful in solving problems. This easy to understand and practical guide to debugging is invaluable to engineers who want to learn how to debug their projects effectively.

[13] The "IEEE Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language" [13] is a formal description of SystemVerilog language including design and verification parts. Such a standard acts as a crucial document in order to ensure the interoperability,

portability and consistency of the methodology and tool set applied in the semiconductor industry. This standard plays a very important role in establishing a common framework to operate with.

[14] IEEE's "IEEE Standard 1850-2010: IEEE Standard for Property Specification Language (PSL)" [14] developed by IEEE, defines the formal language that is being used to define temporal properties of hardware designs which may be utilized for assertion checks through simulations and formal verification. This theory forms the basis of concurrent assertion evaluation technique used in the checking of AHB protocols in this thesis, particularly properties across multiple clock cycles.

[15] Bergeron, Cerny, Hunter and Nightingale's "Verification Methodology Manual for SystemVerilog" [15] provides easy-to-follow instructions on how to create hierarchies for the testbenches and manage the verification process in highly complicated SoCs. The author presents four steps that include test creation, protocol execution, output observation, and result validation. This method will help in constructing the UVM testbench for AHB verification.

[16] Vijayaraghavan and Ramanathan's "A Practical Guide to SystemVerilog Assertions" [16] considers assertion styles that can be applied both immediately and concurrently, property specifications, and incorporating SVA into simulation and formal verification workflows. Also considered are AMBA related temporal properties and assertion binding methodologies that were the basis for the assertions and their binding to interfaces used in this thesis' verification environment.

[17] Piziali's "Functional Verification Coverage Measurement and Analysis" [17] covers in detail the subject of coverage metrics by providing formal definitions of coverpoints, bins, cross coverage, and coverage closure requirements that are now part of standard industrial practices. This paper also supplies the theoretical background for the functional coverage model used in the present thesis where the covergroup definitions for burst type, transfer direction, transfer size, and address range are defined.

[18] Keating and Bricaud's "Reuse Methodology Manual for System-on-a-Chip Designs" [18] provides guidelines to design reusable verification components with emphasis on portability and reusability with detailed explanation of abstraction of interfaces, parameterization and packaging of components. This paper provides direct support to the thesis argument presented in this paper regarding reuse in verification using UVM being greater than that of project specific directed testbenches.

[19] Foster, Krolnik and Lacey's "Assertion-Based Design" [19] covers the topic of assertion-based verification in full by showing how assertion properties can be used to monitor the protocol compliance continuously along with the simulation checking. This work presents a method for classification of assertions and their inclusion in the verification process along with the assertion coverage. This can be seen from the multi-layered verification methodology that uses scoreboards, functional coverage, and assertion checking in this thesis.

[20] Mehta's "ASIC Verification: A Guide to Functional Verification of Complex SoC Using OVM/UVM" [20] provides a practical approach on how to carry out the entire verification process starting from the test bench structure all the way to cover age and regression management for the complicated SoCs designed through the OVM and UVM methodologies. This was used as a practical source for carrying out the UVM testbench design that was done in this research.

[21] Blackmore's "A Practical Approach to UVM Verification of AMBA Bus Protocols" [21] shows how the architecture of agents, scoreboards, and coverage collection can be designed explicitly for the verification of AHB and AXI protocols. It is one of the best examples available for the design of AHB UVM testbench described in this thesis and confirms that the agent-based methodology is viable for AMBA protocol verification.

[22] Lacey's "Systematic Coverage Closure Using Directed and Random Techniques" [22] concludes that relying solely on directed testing or random verification is not sufficient for achieving the most efficient closure of the design. Rather, a combination of the two provides a superior methodology for achieving improved coverage, increased bug detection capability, and more efficient simulation budgets. The findings will be used to provide a direct comparison of directed testing and constrained random verification in this thesis.

2.1 Evolution of Functional Verification

The process of functional verification has undergone tremendous changes with the rising complexity of digital integrated circuits. In simple digital systems, the process of verification involved application of simple test vectors to the inputs of the system[9]. As the scale of integration has become high and the concept of System on Chip has started emerging, the total number of combinations of possible inputs has risen exponentially. It became necessary to adopt an effective verification methodology.

With the development of hardware description language, simulation-based verification has emerged as the de facto method of verification. Stimulus generators have been used to generate stimulus and capture the output[8]. The methodology of verification gradually changed from applying stimulus to coverage-based verification, which is not just about verification but is also about measuring the level of coverage of the behavior of design[10].

2.2 Directed Testing Methodology

Directed testing is among the first and most obvious methods used in functional verification. Directed tests are manually developed to verify functional requirements by providing specific input scenarios. Each test is created to perform a certain functionality within the design such as a certain type of transfer or control.

Directed tests for bus protocols like AHB usually consist of simple read and write transactions, burst transactions and responses [6].

Directed tests can be praised for their simplicity and clearness. The engineer has complete control over the stimuli applied to the design, which simplifies the process of debugging. Nevertheless, the approach is highly dependent on the skill of the engineer to predict all scenarios. When the number of transaction variants increases, it becomes complicated and time-consuming to obtain full coverage using only directed tests.

2.3 Constrained Random Verification Concepts

Constrained random verification was introduced to address the shortcomings of manual testing. Rather than specifying all the inputs, you can define constrains that ensure that all the randomly selected inputs fall within an acceptable range. The testing process can then automatically test different scenarios within those constraints.

With respect to the AHB protocol, various constrained random approaches can be employed to randomly select combinations of various transfer types, burst lengths, address alignment and responses. Coverage metrics can be used to determine what combination is covered by testing [7]. When required coverage targets are not reached, constraints can be modified to direct stimuli generation towards uncovered areas of the design state space.

2.4 SystemVerilog in Verification

The addition of verification methodologies makes SystemVerilog more suitable for describing hardware. Some of the added methodologies include object oriented programming, random testing, functional coverage, and assertions. This makes the development of reusable verification environments easier.

In relation to AHB verification, SystemVerilog has the capability of conducting transaction level modeling in which the transfers on the bus are modeled using high level objects. Assertions are used to check whether protocol is adhered to in terms of the validity of transfer and response types [13]. Functional coverage mechanisms are used to track the transferred data types, bursts and responses.

2.5 Universal Verification Methodology Overview

Universal Verification Methodology is an object-oriented methodology that relies on SystemVerilog as a basis in order to develop a reusable and scalable environment for verification. Universal Verification Methodology describes a certain architecture including such components as drivers, monitors, sequencers, agents, and scoreboards[10]. Such organization of the code provides clear division between stimulus generation, driving signals and checking results into specific layers. Figure 2 is a hierarchical view of all

UVM base classes. This figure shows the inheritance principle used in UVM. This hierarchy should be understood because all components, sequences and testbench objects inherit from the UVM base classes.

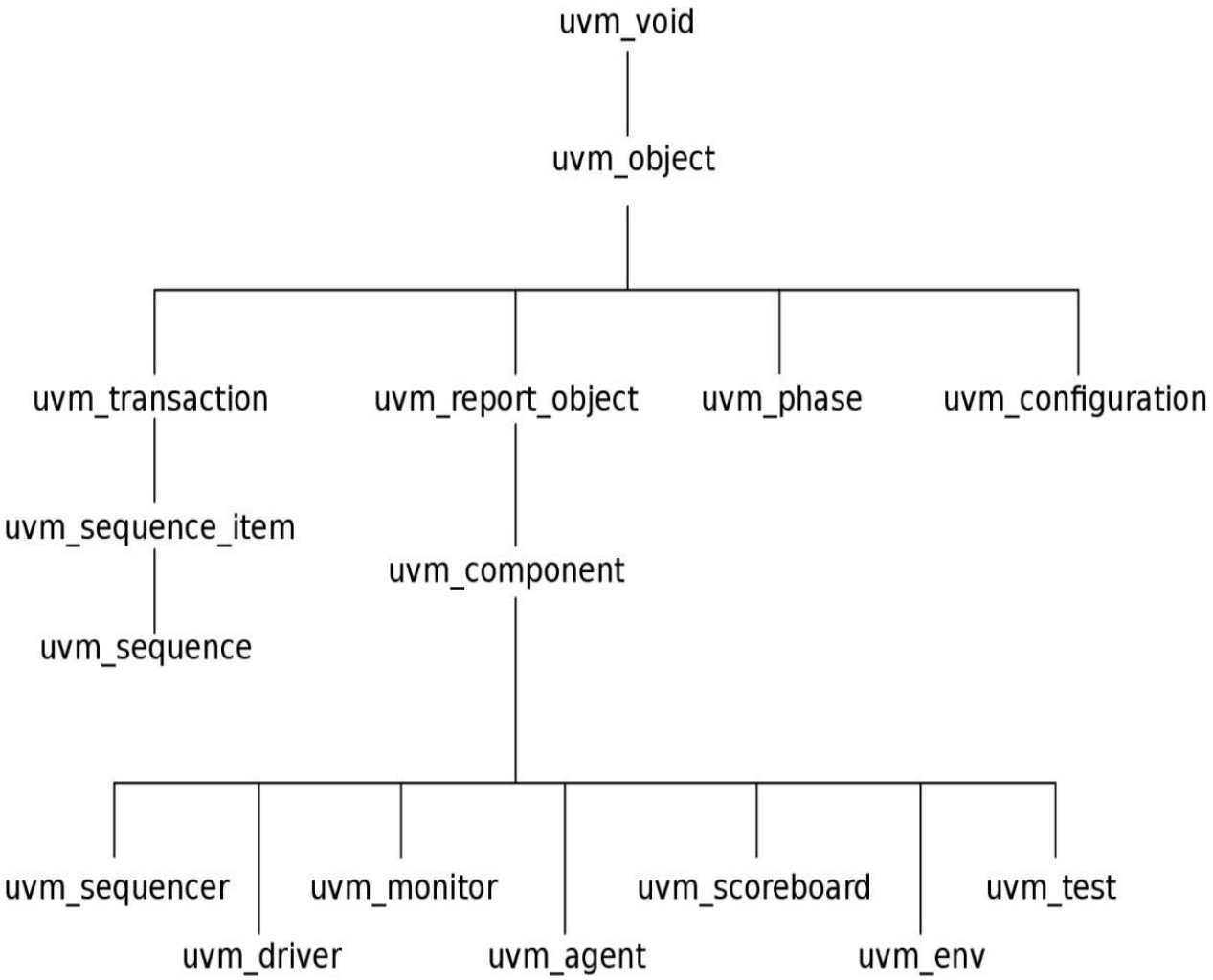


Figure 2.1: UVM CLASS HIREACHY

In AHB verification, a UVM-based environment may contain agents for the master and slave that would perform bus transactions. Sequences could create constrained random transactions, and scoreboards may compare the expected and real behavior[12]. The possibility to configure the environment and reuse its components makes it possible to adapt this environment to other AHB configurations.

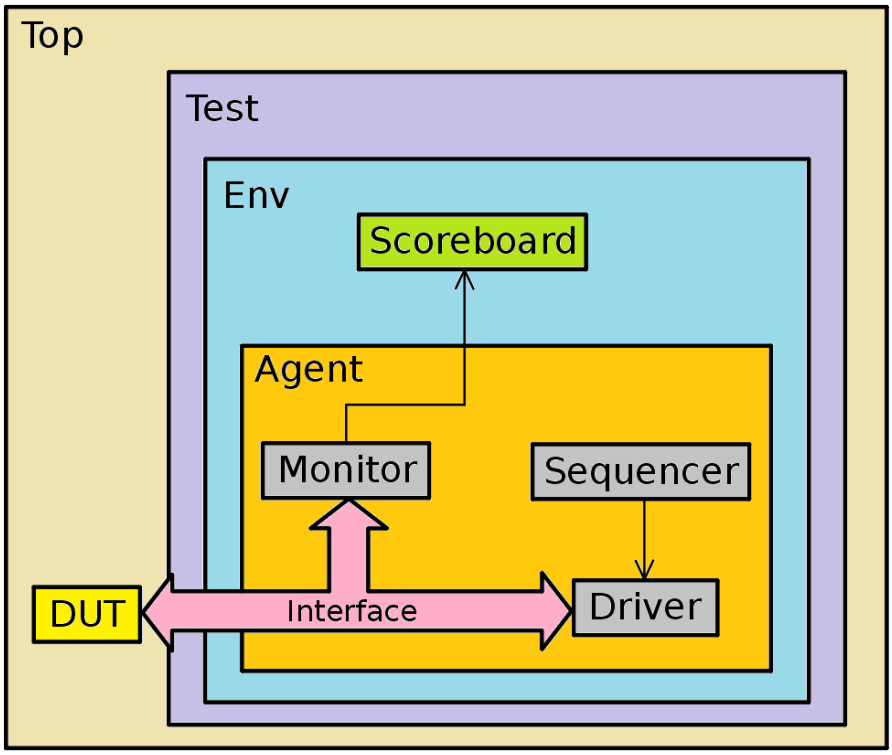


Figure 2.2: BASIC UVM TESTBENCH ARCHITECTURE

The figure below depicts an illustration of the typical UVM testbench structure and describes how the different verification components interrelate within the testbench. The top module houses both the design under test and the interface that connects the design under test to the UVM environment. In the testbench, which inherits the test class in UVM, the design environment is configured, sequences are selected, and the verification process starts[7]. In the test environment, the verification environment acts as the principal component where all other verification components reside such as the agent and the scoreboard. The agent, on the other hand, encompasses the principal verification components needed to create and observe the stimuli. The verification components that make up the agent are the sequencer, the driver, and the monitor. The sequencer generates stimulus in terms of transactions, while the driver transforms those transactions into signals that are then driven to the interface of the design under test. The monitor passively monitors the activities in the interface in terms of signals, transforms them to transactions and delivers the transactions. The scoreboard uses the monitored transactions to perform result checking where it compares the actual output behavior of the design with the expected behavior.

2.6 AHB Protocol Verification Techniques

Usually, the verification of the AHB protocol is carried out based on compliance with timing relations, sequence of operations on control signals, and bursting rules. In the literature, one can find that both

directed testing and constrained random testing are actually applied. Directed testing is often used for bring-up and sanity testing, whereas random testing is used to increase coverage [13].

There are various approaches to verification used. One of them is assertion-based verification – it verifies that rules of protocol work correctly. The second one is coverage models which take into account all possible combinations of transfers tested. Scoreboard is the mechanism that allows verifying the correctness of the data.

According to some researchers, the transaction-level modeling is useful for creating testbenches and increasing understandability of this procedure. It depends on the size of your project what approach will be more appropriate..

2.7 Research Gap and Motivation for Comparative Study

While the two verification approaches mentioned above are discussed in verification literature, there seems to be no direct comparison between them in relation to the AHB protocol. Although numerous pieces of research discuss the general advantages of random verification methodology, there is no sufficient analysis comparing the aspects of coverage efficiency, fault detection capability, and implementation costs for the AHB protocol.

The need for comparison described above leads to the necessity of conducting a comparative study of the two methods under discussion. Thus, the current thesis will examine the two approaches by implementing them in the context of the same AHB protocol under identical conditions. The results of this analysis will be obtained by evaluating various coverage metrics and simulation/bug hunting effort.

CHAPTER 3: AMBA AHB Protocol Architecture

3.1 Overview of AMBA Architecture

Any System-on-Chip design that mixes fast processing cores with slower peripherals needs a common language for those blocks to talk to each other. The Advanced Microcontroller Bus Architecture (AMBA) is ARM's answer to that problem: a family of bus protocols that standardizes how masters (processors, DMA controllers) and slaves (memory, peripheral interfaces) exchange address, control, and data information [2]. Figure 4 shows a representative ARM-based SoC in which high-speed blocks sit on one bus tier and low-speed peripherals sit on another, bridged together rather than sharing a single uniform bus.

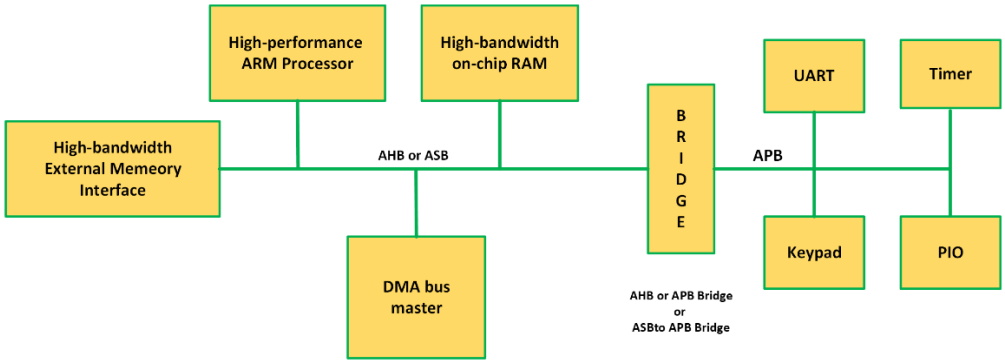


Figure 3.1: AMBA BASED MICROCONTROLLER ARCHITECTURE

That two-tier split is the central design idea behind AMBA and the reason it is relevant to the verification work in this thesis. Putting a fast CPU and a slow UART on identical bus logic would either throttle the CPU to the UART's pace or waste silicon giving the UART hardware it doesn't need. AMBA avoids both outcomes by letting the high-performance tier (AHB, or its predecessor ASB) run a pipelined protocol tuned for throughput, while the low-power tier (APB) runs a simpler protocol tuned for low gate count. Bridges between tiers absorb the protocol translation, so a peripheral block designed against APB never needs to know anything about AHB's timing.

It has another advantage that is not immediately obvious. It allows IP blocks designed by different teams or supplied by different vendors to work together without having to negotiate names or conventions about signalling every time because they have been defined in advance by AMBA. This property is exactly what this thesis builds on in creating a verification environment that is based on the AHB specification and can be used repeatedly for different slaves. The rest of this chapter discusses in detail the specific level of the AHB tier as it is the protocol being tested [1].

3.2 AHB Bus Structure and Components

AHB follows the shared bus architecture, where each master and slave device is connected to a common bus for the transfer of addresses, controls, and data, rather than through individual point-to-point connections. This makes the interconnection quite simple and scalable as well; all that changes when it transitions from a single master to multiple masters is the addition of arbitration functionality.

Four roles make up a typical AHB system. Bus masters processors, DMA engines initiate transfers. Bus slaves memory blocks, peripheral interfaces — respond to whichever master currently holds the bus, each one mapped to a fixed range of the system address space [1]. Figure 5 shows the input/output signal set for one such slave: it accepts control, address, and data signals from the bus, processes them, and drives back read data together with a response code. Sitting above the masters and slaves are the two pieces of "traffic control" logic: an arbiter, which decides which master gets the bus when more than one wants it at the same time, and a decoder, which reads the address on the bus and asserts the select line for the one slave that owns that address range.

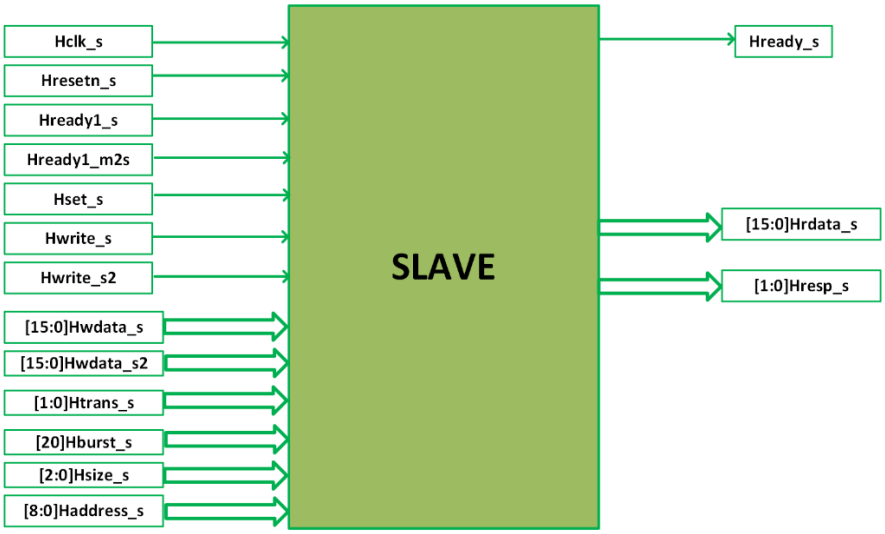


Figure 3.2: AHB SLAVE SIGNALS

Because address and data are handled in separate phases and because those phases are pipelined so that one transfer's data phase overlaps the next transfer's address phase a single AHB transaction unfolds as: master requests, arbiter grants, master drives address and control signals in the address phase, then the actual data crosses the bus in the following data phase, with the transfer's size, direction, and burst behaviour all encoded in that initial control information [1].

Figure 6 illustrates the complete process of arbitration, addressing stage, and data stage of multiple masters in a system. The aspect that allows the protocol to be independent of the speed of individual components within the system is the two signals that are superimposed over this structure: the ready signal which a

slower slave can deassert to add wait states and the response signal that indicates whether the transaction was successful or if it was erroneous. It is these four components: masters, slaves, arbitration, and decoding, that endow AHB with its bandwidth and where verification environment depends on [1].

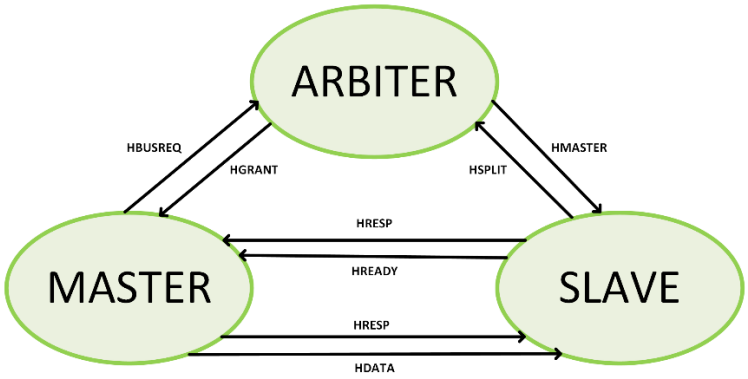


Figure 3.3: AMBA BASED CONNECTION IN AHB BUS ARCHITECTURE

The signaling and interactions as illustrated in Figure 6 indicate how bus ownership is managed through the arbiter, how the master initiates a transaction, and how the slave responds with data/status. This coordination ensures that the AHB protocol works smoothly and in an orderly way, even in a scenario where there are many masters on the bus.

3.3 AHB Signal Description

AHB communication reduces to four functional signal groups, all referenced to a common clock. Verifying the protocol means verifying that every one of these groups behaves correctly, both individually and in relation to each other [1].

3.3.1 Global Signals

The clock and reset lines are shared by every block on the bus. The clock is what makes bus behavior deterministic nothing changes except at a clock edge and reset brings every connected module to a known state before any transaction can begin. Every other signal group in this section is only meaningful relative to these two.

3.3.2 Address and Control Signals

During the address phase, the master asserts the target address together with a set of control fields that describe the transfer: a transfer-type field marking the cycle as idle, busy, non-sequential, or sequential; a read/write line; a size field selecting byte, half-word, or word granularity; and a burst field specifying whether this is a standalone transfer or one beat of an incremental or wrapping burst [1]. An optional

protection field can also flag privilege level or access type where the system needs it. Taken together, these fields are what let a decoder and a slave know, before any data moves, exactly what kind of transfer is about to happen.

3.3.3 Data Signals

The data phase is where the payload actually crosses the bus write data flows from master to slave, read data flows from slave to master over a bus width that is fixed for a given implementation but can differ between systems [1]. Data is only considered valid once the ready signal confirms the transfer has completed, which is the link between this signal group and the handshake mechanism described next.

3.3.4 Response and Handshake Signals

Two signals govern the pace of every transfer. Ready indicates whether the current transfer can close on this cycle; a slave that isn't finished processing deasserts it and forces a wait state. Response reports the outcome success or error: once the transfer does close. Together they are the mechanism that keeps a fast master and a slow slave synchronized without either one guessing at the other's timing, and they are consequently one of the higher value targets for verification, since almost every protocol violation eventually surfaces as an incorrect ready/response combination [1].

3.4 AHB Transfer Types and Transaction Flow

Every AHB transfer type tells a decoder and slave what to expect before the data phase begins, and the transaction flow that connects them is what a verification environment has to reconstruct and check cycle by cycle.

3.4.1 Transfer Types

Four transfer-type encodings exist. Idle means no meaningful transfer is happening on this cycle. Busy means a master mid-burst needs a cycle before its next address is ready, but is holding the bus rather than releasing it. Non-sequential opens a new transaction either a standalone transfer or the first beat of a burst with an address unrelated to whatever came before. Sequential follows a non-sequential transfer inside a burst, with its address derived from the previous one according to the burst's increment rule.

3.4.2 Single and Burst Transfers

A single transfer is the simplest case: one address phase, one data phase, done. Bursts extend this by chaining several data transfers under a single bus grant, which amortizes arbitration overhead across the whole sequence rather than paying it per transfer [1]. AHB defines both fixed-length bursts, which move a set number of beats and stop, and undefined-length incremental bursts, which continue until the master

itself ends the sequence. A third variant, wrapping bursts, keeps the address inside a fixed-size window by wrapping back to the base address once it reaches the boundary the mechanism that makes cache-line fills and circular-buffer access patterns efficient. Table 1 lists the eight encodings of the 3-bit HBURST field that select among these behaviors.

Table 3.1: BURST TRANSFER TYPES IN AHB

HBURST [2:0]	TYPE	DESCRIPTION
000	SINGLE	Single Transfer
001	INCR	Incrementing burst of unspecified length
010	WRAP4	4-Beat wrapping burst
011	INCR4	4-Beat incrementing burst
100	WRAP8	8-Beat wrapping burst
101	INCR8	8-Beat wrapping burst
110	WRAP16	16-Beat wrapping burst
111	INCR16	16-Beat incrementing burst

3.4.3 Transaction Flow in AHB

A transaction starts with a master requesting the bus. Once the arbiter grants it, the master opens the address phase by driving address and control signals, and the decoder identifies the target slave from that address. The data phase follows on the next cycle: the master supplies data for a write, or the slave returns data for a read, while pipelining lets the next transfer's address phase begin in parallel with the current transfer's data phase.

Figure 7's waveform illustrates this for a basic write: in cycle one the master drives address A and asserts the write line; in cycle two, with ready high, the master's write data for A appears on HWDATA while address B for the following transfer is already visible on HADDR. This is the throughput payoff of pipelining one transfer completing per cycle as long as ready stays asserted [2]. That may be done in time or delayed due to waiting states, and all depends on the ready signal, whereas the response signal will tell if the transfer went without problems or not.

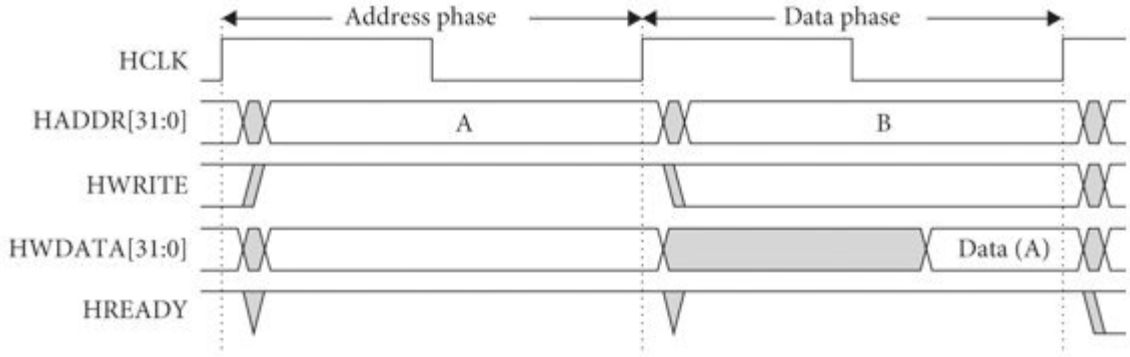


Figure 3.4: AHB DATA PHASE AND ADDRESS PHASE IN WRITE TRANSFER

3.4.4 Pipelined Operation and Overlapping Phases

It is this overlap in which one transfer's data phase overlaps the address phase of the following transfer that provides the AHB's throughput, but it also makes verification of the AHB more difficult to verify than a simple non-pipelined bus because of the dependency of the correctness of the bus on the interactions between consecutive transfers rather than any one transfer by itself.

Figure 8 illustrates this in a write-then-read operation where cycle one contains the write address A, cycle two holds the write data A along with the read address B, and cycle three holds the read data B [2]. It is the interaction between these three cycles that need to be verified for correct addressing, data and response timing and not just individual transfer verification.

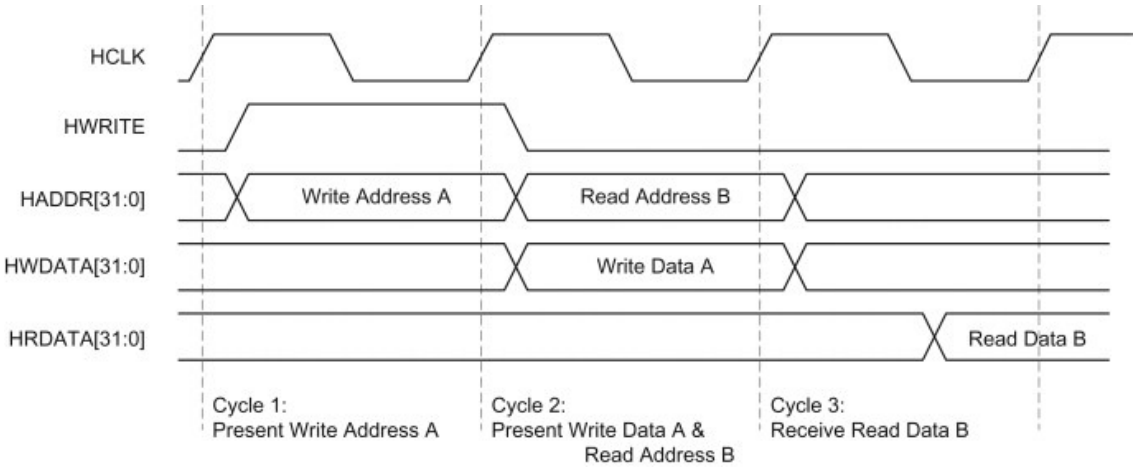


Figure 3.5: AHB PIPELINED OPERATION

3.5 Burst Transfers and Pipelining

Burst transfers are one of the key components in the AHB protocol that help make data transfer over the bus more effective. In sections 3.4.2 and 3.4.4 we discussed the concept of bursts and pipelining; in this section, we'll see why they are important and what verification entails.

3.5.1 Types of Burst Transfers

The AHB supports two modes of bursts: increment and wrap. Incrementing bursts provide addresses in an increasing fashion after each data transfer. They may have fixed or variable lengths according to system requirements[2]. In fixed-length bursts, a set number of data items are transferred, but undefined bursts go on till the master stops them.

Wrap bursts are used when the address range must not exceed the defined limits. Once the address reaches the limit, it goes back to the initial boundary address. This method is highly beneficial for cache line filling and circular buffering. The correct operation of address generation needs to be verified properly.

3.5.2 Pipelined Operation

AHB protocol uses pipelining for separation of address phase and data phase. In this way, while one transaction is being completed with data phase, the next one begins its address phase. Such an overlapping guarantees that there will always be work for the bus and prevents unnecessary idle cycles.

While pipelining contributes to increase efficiency, it complicates verification process. Connection between current and previous transactions should be taken into account, especially in the case of bursts. Verification should guarantee proper alignment of address generation, data transaction, and response signaling from cycle to cycle.

3.6 Arbitration and Bus Master Control

In those cases where multiple masters exist, there should be an arbitration mechanism to regulate the access to the shared bus. Arbitration is responsible for allocating bus access to each master in accordance with a pre-specified priority scheme.

3.6.1 Arbitration Process

There needs to be an entity responsible for settling concurrent requests in multi-master systems, and that is what the arbiter does: It analyzes outstanding requests and gives access to the bus based on the policy the system uses, which could be fixed priority, round-robin, or anything else [2]. The bus is held by a master until its transfer or burst completes, without any preemption in between.

From the verification point of view, there are two main properties of the arbitration that have to be verified explicitly: mutual exclusion (two masters never control the bus simultaneously), and fairness,i.e., no

master will never be starved infinitely under the used policy. Both properties cannot be easily seen from one trace, as they involve observing behavior in multiple arbitrations.

3.6.2 Bus Master Responsibilities

Whether it be the current master that owns the bus, this particular entity possesses the addresses, controls, and data lines and is supposed to abide by the protocol specifications in relation to the sequence of transfers, the bursting nature, and responses. However, in reality, once a master gains control, it retains such until the arbiter changes the ownership, which makes master control logic another potential source of problems with the protocol.

3.7 Timing and Handshake Mechanism

AHB is purely synchronous: all transitions are synchronized with one common clock edge, and this is what guarantees transfer timing synchronization throughout the entire system rather than depending on the propagation delays of particular devices.

3.7.1 Address and Data Phase Timing

The address phase takes up one clock period when the master sends out its address and control signals; the data phase comes in the following clock period [2]. Due to pipelining, the address phase of one transfer and the data phase of another can coincide in time, and therefore, testing will have to prove two aspects: that control signals are kept stable for their valid time window, and that data is read only after the transfer has been successfully completed.

3.7.2 Ready Signal Based Handshake

Ready is the signal component through which an efficient master is matched to an inefficient slave. If the slave is not yet finished, then ready will stretch the data phase by holding off the assertion of ready, thereby forcing the master to keep its address and control signals in their current state until the slave completes its work [2]. Checking the validity of this protocol involves checking whether the wait states occur exactly when they should, and whether address and control signals really remain frozen during that time.

3.8 Error Handling and Response Mechanism

3.8.1 Response Types

The response signal is a binary status for each transfer either of normal completion or an error signifying such things as invalid address or protection violation. Based on system architecture, the master can choose to cancel the remaining transfers of a burst operation in process or do something else in response to an error response received. The verification procedure is made up of two separate parts that verify that the

slave generates error responses only when it encounters faults and that the master behaves accordingly to an error response.

3.8.2 Handling of Fault Conditions

Examples of fault scenarios would be address decode fault, permission fault, or some other kind of timing violation within the protocol [8]. Any of these scenarios should not be able to corrupt the data or affect the bus in any way for future transactions. The confirmation of fault response accuracy (correct response only when there is an actual fault condition) and containment (not affecting other transactions following the faulty transaction) is what completes the AHB verification regarding fault handling, which in turn concludes our signal and timing model presented in this chapter of masters, slaves, arbitration, decode, and the ready/response exchange system working in concert to ensure bus stability in a multi-master environment [1][8].

CHAPTER 4: Directed Testing Based Verification Of AHB

4.1 Concept of Directed Testing

Directed Testing approach can be defined as the form of verification where input sequences are determined beforehand for the testing of certain functional requirements of the design[13]. Each test case of the testing is done on a particular function of the requirements of the protocol.

For example, in the case of AHB protocol, the directed testing approach involves testing of AHB protocols' functionalities like single read operation, single write operation, incremental burst operation and wrapping burst operation[8]. In directed testing, verification engineer completely controls all the address, control and data signals of the design.

Directed Testing approach can effectively be applied in the initial phase of AHB protocol implementation.

4.2 Testbench Architecture for AHB Using Directed Approach

This testbench architecture, designed for this thesis, validates an AHB Lite slave memory through manually crafted stimulus wherein every transaction parameter, such as address, data, burst type, and direction is explicitly specified by the verification engineer. This architecture maintains a layered approach to maintain separation between stimulus creation, bus driver, and checker; however, the difference between a randomized approach is that there is no constraint solver and randomization involved in generating test vectors [11]. This section will briefly discuss each layer of this architecture.

Design Under Test

In this case, the design under test is an AHB Lite slave memory whose parameterization includes a 32-bit address bus width, 32-bit data width and memory depth of 1024 words, resulting in an addressable memory of 4 KB. It implements a standard AHB two-phase pipelining with the current transaction's address phase overlapping with the last transaction's data phase.

It implements all of the burst types as specified by the AHB Lite specification, namely SINGLE, INCR, WRAP4, INCR4, WRAP8, INCR8, WRAP16, and INCR16 [1]. The design implements a one-cycle response protocol with HREADY signal always asserted and HRESP set to OKAY value. This makes it easier to model latency while testing the full address and data path. Upon reset of the design, the address pipeline registers are reset and HRDATA outputs are set to 0.

AHB Interface

An AHB interface is utilized to establish a connection between the testbench and the device under test [12]. In System Verilog (ahb_if), an interface is used to connect the testbench to design under test. The interface acts as a connection mechanism for both testbench and the device under test where all the AHB-

Lite signals are arranged in one block. The AHB-Lite signals include the signals driven by master (HADDR, HBURST, HSIZE, HTRANS, HWRITE, and HWDATA) and the slave signals which provide response signals (HRDATA, HREADY and HRESP). This helps to increase the readability of the bus in the testbench and avoids the need to connect each signal separately.

Not only does the interface help to simplify the connectivity problem, but it also helps to improve the maintainability and reusability of the testbench code. Since the bus level details are abstracted in the interface, it becomes easy for the testbench to operate at transaction level. This means that it will be easy to incorporate more verification aspects like clocking blocks, modports and protocol tasks in the testbench. Therefore, an AHB interface (ahb if) is a very useful tool regarding connectivity and architecture.nvironment and the DUT by organizing all the AHB-Lite signals in one structured block. The AHB-Lite signals include the signals driven by the master, including HADDR, HBURST, HSIZE, HTRANS, HWRITE, and HWDATA, as well as the slave signals providing the response, such as HRDATA, HREADY, and HRESP. This bus organization increases the readability in the testbench and reduces the difficulty of connecting each signal separately.

Not only does the interface simplify the connectivity problem, but it also makes the testbench code maintainability and reusability much easier. The abstraction of the bus level details enables the testbench to focus on transaction behavior rather than bus level wiring. This makes it easier to add extra verification features such as clocking blocks, modports, and protocol tasks to the testbench. Thus, the AHB interface (ahb if) is a very useful tool regarding connectivity and architecture.

Top Level Testbench Module

The top level testbench module referred to as "tb_top" is the central coordination entity in the simulation environment. It generates the system clock required for the synchronous operation and resets the design under test (DUT). Afterward, it removes the reset in order to start the regular operation. This module generates an instance of the DUT and connects it with the appropriate verification interface in order to ensure efficient interaction between the testbench and DUT. It also initiates waveform dumping to allow observation of the signals after the simulation process as it is useful for the inspection, debugging and analysis purposes. All these operations are gathered together in order to provide well-organized and maintainable foundation for the verification environment.

Driver

AHB driver created as uvm_driver is the active interface between the sequencer and the bus. Each transaction object generated by the sequencer is processed by applying the corresponding signal values on the AHB interface with the use of master clocking block [10]. It means that the driver translates the abstract transaction information into concrete actions of the bus at the pin level and allows stimulation of the DUT

in a time-controlled way. Thus, the driver maintains the correct order of the signal updates in accordance with the clock and ensures the proper representation of the protocol on the bus level.

Monitor

The Monitor is the passive part of the verification process, as it does not play an active role but is absolutely vital for the purpose of displaying information. The monitor clocking block in the verification environment observes the activity on the AHB bus without affecting the signal behaviour on the interface [10]. It monitors the transaction time address, control and data behavior and reconstructs this information for further checking and analysis. The monitor itself does not provide any stimuli, so its design evaluation is non-invasive. This factor is very important for getting exact results of verification. The information acquired by the monitor can be additionally distributed to the components like scoreboard and coverage collector in an effort to check the correctness of work and function analysis.

Scoreboard

The scoreboard is an associative array indexed by address and is a reference memory model which reflects the expected contents of the slave. When the monitor receives a write transaction, the scoreboard walks through the address sequence storing each beat data value at the corresponding address in the reference array[7]. The scoreboard fetches the expected value from the reference array and compares it with the actual data seen on the bus when a read transaction is received. At the end of the simulation, the scoreboard prints a summary with the total write count, read count, pass count and fail count. The test is reported as passed only if the scoreboard saw zero mismatches.

Directed Sequence

The directed sequence is the defining component that distinguishes this architecture from the randomized approach[5]. The class ahb_wrap4_directed_seq extends uvm_sequence and constructs each transaction by explicitly assigning every field without invoking the randomize function at any point. Figure 9 illustrates the standard UVM stimulus generation mechanism where a uvm_sequence generates data items (seq_item) and passes them through a UVM_SEQUENCER to be executed by the UVM_DRIVER.

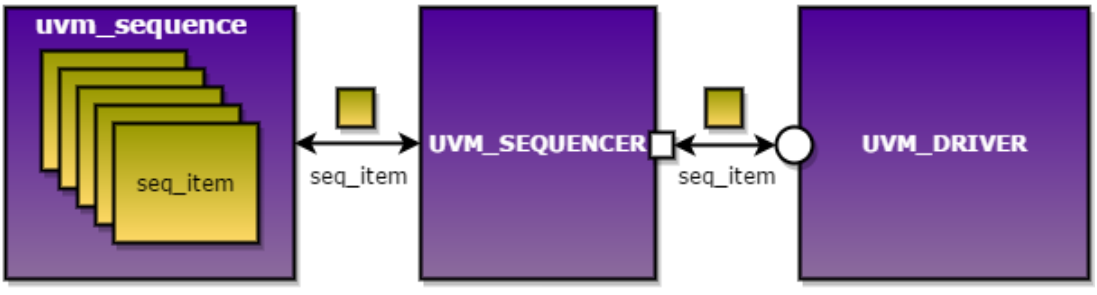


Figure 4.1: UVM Sequence Flow

4.3 Stimulus Generation Strategy

In directed testing stimulus generation is scenario driven. The engineer derives the functional requirements from the AHB protocol specification and separately writes test cases for the relevant scenarios[12].
The stimulus is designed to guarantee the legality of the transitions among the different transfer types. We take a detailed look at things like address alignment and burst termination.

4.4 Directed Test Case Development for AHB Transfers

Directed test case development Directed test case development is concerned with generating specific scenarios within the sequence class `ahb_wrap4_directed_seq. In this process you explicitly set each address, data value, burst type, transfer direction. No randomization. Usually, each test case consists of writing data first and then reading it back to verify data integrity.

4.5 Functional Coverage in Directed Testing

Functional coverage in directed testing is measured using coverage constructs that track exercised transfer types, burst modes and response conditions[4].

Example coverage model.

```
covergroup ahb_cg @(posedge HCLK);
  coverpoint HTRANS;
  coverpoint HBURST;
  coverpoint HRESP;
endgroup
```

Coverage results depend entirely on written test cases. If a particular combination is not coded, it will not be exercised. Therefore achieving high coverage requires careful enumeration of protocol scenarios.

4.6 Debug and Result Analysis

Debugging is easy in directed testing, because stimulus is deterministic. If it crashes, you will know exactly what sequence caused the error.
Waveform analysis is often used to verify the correct sequencing of HADDR, HTRANS, HWRITE and HREADY signals. Result analysis is to validate data integrity and to ensure response values are as expected. Coverage reports are analyzed to find untested scenarios.

4.7 Advantages and Limitations of Directed Testing

The method of directed testing allows for a simple and explicit control of the stimuli. It's very good for testing basic functionality, and making sure protocols are followed early in the design process. Plus, it helps to reduce debugging efforts, because the behavior is usually pretty predictable.

However, the limitations are limited scalability and the reduced corner case exploration. As the number of possible AHB transaction combinations increases, exhaustive directed tests become time consuming to write and maintain[1]. The methodology does not have inherent randomness and reuse ability as compared to UVM based verification.

CHAPTER 5: UVM Constraint Based Random Verification of AHB

5.1 Overview of UVM Based Random Verification

With the establishment of directed testing in the previous chapter using the same UVM testbench framework, the next phase is the verification of the design by switching the deterministic sequence block into a constrained-random sequence block while keeping the rest of the environment the same. The driver, monitor, scoreboard, interface, top-level module, and the DUT described in Chapter 4 are still the same in this phase of verification. The sole difference lies in the sequence layer where the explicit field assignments have been replaced with SystemVerilog constraints blocks and randomize calls to let the simulator generate different, but still legal AHB transactions[14]. A UVM verification environment designed to conduct tests of a directed vs random approach on AHB slave DUT is shown in Figure 10 below. It shows how the top testbench (tb_top) instantiates the interface, initializes the configuration database, and drives the stimulus via AHB agent using both directed and random sequence blocks.

This architectural consistency has been purposely designed for a reason. It shows one of the inherent strengths of the UVM approach, and it lies in the ability to change the methodology of the stimulus generation independent of the architecture of checks and observations.[15] The scoreboard reference model continues to check all transactions irrespective of whether the data was written by the engineer himself or the constraint solver. The monitor also continues to passively observe and decode bursts from the bus signals. Comparison of the two methods of verification is thus an authentic test of the stimulus methodology strength.

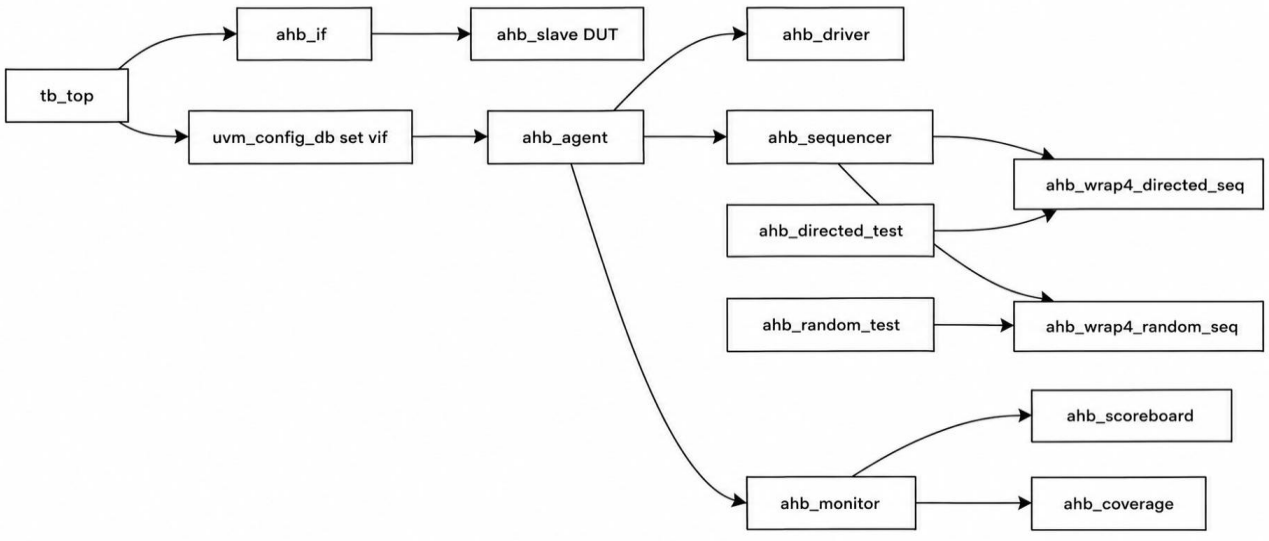


Figure 5.1:UVM Architecture diagram

5.2 Transaction Class and Constraint Definition

The transaction class ahb_seq_item employed for the chapter on directed testing is extended with constraint blocks to allow for constrained random generation. The basic declaration of the fields for address, burst mode, size, write mode, data array, and burst length is identical to the one used in the directed test. What is added is the use of constraint blocks defining the legal range of values for each field.

The address constraint ensures that any addresses generated are within the range of the 4 kilobyte address space of the slave memory, such that out-of-range access is not created during the random stimulus phase. The alignment constraint ensures that the generated addresses conform to the word alignment requirement of 32 bits, thus avoiding the generation of misalignment accesses that are not supposed to be handled by the current DUT[16]. We have made sure that the burst type constraint in the base random sequence is WRAP4 to ensure consistency with the guided test case during the first comparison phase. The burst type constraint will however be relaxed in the longer sequences. This will allow us to study different burst types. The data array size constraint ensures that the dynamically allocated data array size conforms to the burst length based on the burst type.

Solve before directive may be inserted between burst and burst_len in order to guarantee that the solver will determine the burst type prior to determining the size of the dependent array. A weighted distribution may be used on the write field to skew the ratio of writes over reads, for instance, with a weight of 60% for the former and 40% for the latter, which mimics the behavior of actual bus traffic. In this way, we can direct the random stimulus towards our needs while preserving its randomness[17].

5.3 Constrained Random Sequence

In the restricted random sequence ahb_wrap4_rand_seq, the randomization of the fields specified in the directed sequence is replaced by just one randomize operation per transaction. This sequence is looped for a configurable number of times, where on each iteration a new random transaction is created. The start address, data value, and direction vary from iteration to iteration under the constraints, creating different and varied sequences of AHB WRAP4 transactions, which test more varied addresses and data than the fixed sequence.

Code:

Constrained random WRAP4 sequence

Source: ahb_wrap4_random_seq.sv

```
// ============================================
// Randomized WRAP4 Write-then-Read Sequence
// ------------------------------------------------------------------
// This sequence demonstrates CONSTRAINED-RANDOM VERIFICATION (CRV):
```

```
//   - Addresses and data are randomized within legal constraints defined
//     in ahb_seq_item.
//   - The burst type is constrained to WRAP4, but address and data are free
//     to be any legal value → explores many more scenarios than directed.
//   - Multiple write-read pairs are generated, each with different random
//     values. The number of iterations is configurable.
// KEY DIFFERENCE vs. DIRECTED:
//   Directed = you pick exact values → covers 1 scenario.
//   Random   = solver picks values  → covers N scenarios automatically.
// ============================================
class ahb_wrap4_random_seq extends uvm_sequence #(ahb_seq_item);
  `uvm_object_utils(ahb_wrap4_random_seq)
  // Number of write-read pairs to generate (can be set from test)
  rand int unsigned num_txns;
  constraint c_num_txns { num_txns inside {[4:10]}; }
  function new(string name = "ahb_wrap4_random_seq");
    super.new(name);
  endfunction
  task body();
    ahb_seq_item wr_txn, rd_txn;
    `uvm_info("RND_SEQ", $sformatf("Starting %0d randomized WRAP4 write-read pairs", num_txns),
UVM_LOW)
    for (int i = 0; i < num_txns; i++) begin
      // ---- Random WRITE ----
      wr_txn = ahb_seq_item::type_id::create($sformatf("wr_txn_%0d", i));
      start_item(wr_txn);
      if (!wr_txn.randomize() with {
        burst == 3'b010;              // force WRAP4
        write == 1;                   // force write
        // all other fields (addr, data) are randomized by the solver
        // subject to the constraints in ahb_seq_item
      }) `uvm_fatal("RND_SEQ", "Randomization failed for write txn")
      finish_item(wr_txn);
```

```
    `uvm_info("RND_SEQ", $sformatf("[%0d/%0d] WRAP4 WR addr=0x%08h",
            i+1, num_txns, wr_txn.addr), UVM_MEDIUM)
    // ---- Read-back at the SAME random address ----
    rd_txn = ahb_seq_item::type_id::create($sformatf("rd_txn_%0d", i));
    start_item(rd_txn);
    if (!rd_txn.randomize() with {
      addr  == wr_txn.addr;          // same address as the write
      burst == 3'b010;               // WRAP4
      write == 0;                    // read
      size  == wr_txn.size;
    }) `uvm_fatal("RND_SEQ", "Randomization failed for read txn")
    finish_item(rd_txn);
    `uvm_info("RND_SEQ", $sformatf("[%0d/%0d] WRAP4 RD addr=0x%08h",
            i+1, num_txns, rd_txn.addr), UVM_MEDIUM)
  end
  `uvm_info("RND_SEQ", $sformatf("All %0d randomized WRAP4 pairs complete", num_txns), UVM_LOW)
 endtask
endclass : ahb_wrap4_random_seq
```

Every time randomize() is invoked, a new set of values will be produced for all rand variables provided that the constraint solver resolves that the active constraints are met before sending the transaction to the driver. If the randomization process fails to meet the constraints, then a UVM_FATAL signal will be raised instantly so that a non-valid transaction is not applied in the test [18].

5.4 Sequence Library and Test Classes

The sequence library is designed such that there will be multiple options in terms of varying complexity of the stimuli presented. The WRAP4 constrained random transaction sequence is provided in the form of the above-mentioned basic random sequence for preliminary comparison against the guided sequence group. The addition of address alignment and size constraints for every family of bursts as well as lifting of the constraint on burst types such that all burst types can be included results in a random sequence covering the entire space of burst types. A stress sequence tests the pipelining overlap characteristics of the AHB Slave under heavy workload conditions.

There are two test classes that will be employed in the execution of the above sequences. The first test class is called the base random test class, which executes the WRAP4 constrained random sequence, and will be used for the main analysis of directed versus random testing. The second test class is called the extended random test class, which executes the burst type sequence. This test class will be used for coverage analysis[19].

Code:

Randomized test class

Source: ahb_random_test.sv

```
// ============================================================
// Randomized WRAP4 Test
// Runs the constrained-random sequence – solver picks address & data.
// ============================================================
class ahb_random_test extends ahb_base_test;
  `uvm_component_utils(ahb_random_test)
  function new(string name = "ahb_random_test", uvm_component parent = null);
    super.new(name, parent);
  endfunction
  task run_phase(uvm_phase phase);
    ahb_wrap4_random_seq seq;
    phase.raise_objection(this);
    // Wait for reset de-assertion
    #200;
    seq = ahb_wrap4_random_seq::type_id::create("seq");
    // Let solver choose number of transactions (4–10)
    if (!seq.randomize())
      `uvm_fatal("TEST", "Sequence-level randomization failed")
    seq.start(env.agt.sqr);
    // Drain time for final data phase
    #200;
    phase.drop_objection(this);
  endtask
endclass : ahb_random_test
```

5.5 Scoreboard Behavior Under Random Stimulus

The scoreboard behaves in exactly the same way when stimulated randomly as when stimulated with directed stimulus, and this is due to the similarity of infrastructure designs. All the transactions generated by the monitor are analyzed by the scoreboard, regardless of their source. In the case of writes, the reference memory array is written in beat address with corresponding values. In the case of reads, the expected values are taken from the reference array and compared to the HRDATA values beat by beat [20].

What is important about the work of the scoreboard under the random stimulus is the amount and variety of transactions it has to analyze. While in the case of directed stimulus there is always a fixed number of predefined transactions generated at fixed addresses, in the case of random stimulus there is the variety of addresses chosen in the whole constrained range, leading to more diverse transactions written into the reference array[20].

Scoreboard with reference memory model

Source: ahb_scoreboard.sv

```
// ================================================
// AHB Scoreboard
// Maintains a reference memory model and checks write-then-read consistency.
// ================================================
class ahb_scoreboard extends uvm_scoreboard;
  `uvm_component_utils(ahb_scoreboard)
  uvm_analysis_imp #(ahb_seq_item, ahb_scoreboard) ap;
  // Reference memory mirror
  bit [31:0] ref_mem [bit [31:0]];
  int unsigned write_count;
  int unsigned read_count;
  int unsigned pass_count;
  int unsigned fail_count;
  function new(string name = "ahb_scoreboard", uvm_component parent = null);
   super.new(name, parent);
  endfunction
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
```

```
    ap = new("ap", this);
  endfunction
  // Called by the monitor's analysis port
  function void write(ahb_seq_item txn);
    bit [31:0] addrs[];
    txn.get_addr_sequence(addrs);
    if (txn.write) begin
      // ---- WRITE: store each beat into reference memory ----
      write_count++;
      for (int i = 0; i < txn.burst_len; i++) begin
        ref_mem[addrs[i]] = txn.data[i];
        `uvm_info("SCB", $sformatf("WR ref_mem[0x%08h] = 0x%08h", addrs[i], txn.data[i]),
UVM_HIGH)
      end
    end else begin
      // ---- READ: compare each beat with reference memory ----
      read_count++;
      for (int i = 0; i < txn.burst_len; i++) begin
        if (ref_mem.exists(addrs[i])) begin
          if (txn.data[i] === ref_mem[addrs[i]]) begin
            pass_count++;
            `uvm_info("SCB", $sformatf("PASS RD addr=0x%08h exp=0x%08h got=0x%08h",
                  addrs[i], ref_mem[addrs[i]], txn.data[i]), UVM_MEDIUM)
          end else begin
            fail_count++;
            `uvm_error("SCB", $sformatf("FAIL RD addr=0x%08h exp=0x%08h got=0x%08h",
                   addrs[i], ref_mem[addrs[i]], txn.data[i]))
          end
        end else begin
          `uvm_info("SCB", $sformatf("RD addr=0x%08h not yet written (exp 0x0, got 0x%08h)",
                addrs[i], txn.data[i]), UVM_MEDIUM)
        end
      end
    end
```

```
  endfunction
  function void report_phase(uvm_phase phase);
    super.report_phase(phase);
    `uvm_info("SCB", $sformatf(
      "\n=== Scoreboard Summary ===\n Writes: %0d\n Reads: %0d\n Pass: %0d\n Fail: %0d\n====================",
      write_count, read_count, pass_count, fail_count), UVM_LOW)
    if (fail_count > 0)
      `uvm_error("SCB", "*** TEST FAILED – mismatches detected ***")
    else
      `uvm_info("SCB", "*** TEST PASSED ***", UVM_LOW)
  endfunction
endclass : ahb_scoreboard
```

5.6 Functional Coverage Under Constrained Randomization

Collection of functional coverage data also takes place via the same coverage collector module as explained in the directed testing section. The covergroup will record the transaction data sent from the monitor for the bursts, directions of transfer, sizes of transfers, and address ranges. For directed testing, the coverage growth will be quick for the targeted bins and thereafter halt totally after performing all the directed transactions. For constrained random testing, there will be continuous growth in the coverage because of different seeds that produce different combinations of addresses and write-read mixes in the constrained space [21].

In this case, the address range coverpoint shows the major distinction between the two techniques. In the first case, we have access to two predetermined addresses, which leads to the activation of the bins that belong to those particular address ranges. However, in the second case, we have access to the whole 4-kilobyte range, which allows for complete address coverage of the bins within the address ranges due to the number of transactions that will be carried out during regression. One of the major conclusions from the comparison is the difference in address coverage [22].

A similar pattern could be observed for cross coverage between burst types and transfer directions. While the number of samples is not statistically significant for all combinations of addresses and directions, the directed sequence tests WRAP4 writes and reads in isolation. The bins that remain untouched by the directed sequence are covered by the random sequence, which covers all combinations of directions and

bursts with the frequency corresponding to their constraints by randomizing the write bit and the address in each transfer.

Functional coverage collector

Source: ahb_coverage.sv

```
// ============================================================
// AHB Functional Coverage
// Collects coverage on burst types, transfer sizes, and address ranges.
// ============================================================
class ahb_coverage extends uvm_subscriber #(ahb_seq_item);
  `uvm_component_utils(ahb_coverage)
  ahb_seq_item txn;
  covergroup ahb_cg;
    option.per_instance = 1;
    cp_burst: coverpoint txn.burst {
      bins single = {3'b000};
      bins incr   = {3'b001};
      bins wrap4  = {3'b010};
      bins incr4  = {3'b011};
      bins wrap8  = {3'b100};
      bins incr8  = {3'b101};
      bins wrap16 = {3'b110};
      bins incr16 = {3'b111};
    }
    cp_write: coverpoint txn.write {
      bins read  = {0};
      bins write = {1};
    }
    cp_size: coverpoint txn.size {
      bins byte_    = {3'b000};
      bins hword    = {3'b001};
      bins word     = {3'b010};
    }
    cp_addr_range: coverpoint txn.addr[11:8] {
```

```
    bins low    = {[0:3]};
    bins mid    = {[4:7]};
    bins high   = {[8:11]};
    bins top    = {[12:15]};
   }
   // Cross coverage: burst × direction
   cx_burst_dir: cross cp_burst, cp_write;
  endgroup
  function new(string name = "ahb_coverage", uvm_component parent = null);
   super.new(name, parent);
   ahb_cg = new();
  endfunction
  function void write(ahb_seq_item t);
   txn = t;
   ahb_cg.sample();
  endfunction
  function void report_phase(uvm_phase phase);
   super.report_phase(phase);
   `uvm_info("COV", $sformatf("AHB functional coverage = %.2f%%", ahb_cg.get_coverage()),
UVM_LOW)
  endfunction

endclass : ahb_coverage
```

5.7 Stimulus Generation Strategy in Constrained Random Approach

While the stimulus generation methodology of constrained random differs from the scenario driven methodology used for directed testing, in that while directed testing focuses on identifying specific functional requirements, and generating tests cases that target these requirements individually, constrained random uses constraints to define the legal protocol space, and uses randomization to generate stimulus within this defined space in multiple simulations using different seeds.

For AHB verification, the constraint-based stimulus generation approach starts off by defining the outer limit of the legal stimulus, which includes all the possible combinations of stimulus that fall in line with the AHB protocol[22]. This outer limit is determined by setting constraints on the addresses that should be aligned properly, burst lengths that are consistent, sizes of data arrays and legal types of transfers.

Following this, verification intent shaping is performed within the legal space.

The amount of iteration per simulation run and the amount of seeds used in the regression test suite is designed to provide an appropriate balance between coverage width and the runtime of the simulation. The first round of simulation consists of twenty transactions per seed so that we can quickly learn the characteristics of the random stimulus. The complete simulation consists of one hundred transactions per seed over ten seeds to give us statistical power for coverage analysis. The seeds used are constant in the regression makefile, and additional seeds are appended to the regression seed list when the coverage analysis reveals unactivated bins[23].

5.8 Debug and Result Analysis

Debugging when performed on the basis of constrained random verification is initially done using a different approach compared to directed debugging but ends up following the same waveform-based debugging approach for fault isolation. The first step that one takes after the detection of any issue via the scoreboard during random regression runs is to isolate the failing seed via the simulation logs. The repetition of the same simulation run with the failing seed makes the debugging process identical to directed test fault isolation [24].

All transactions observed during the test failure, along with their addresses, burst types, directions, and data values as they appeared in sequence on the bus, can be read easily from the monitor transaction log in a succinct way. The scoreboard error message highlights the transaction that failed, while the waveform search is constrained within a specific timeframe based on the transaction log position. Afterwards, regardless of the stimulus being directed or random, the normal process of debugging the AHB using HADDR, HTRANS, HWRITE, HWDATA, HRDATA, and HREADY from the waveform viewer can be used.

All transactions that occur in the course of the failure sequence, along with their address, burst type, direction, and data values in the order in which they appear in the bus, are succinctly represented in the monitor transaction log. The scoreboard error message clearly specifies the failing transaction, while the waveform search is narrowed to the appropriate period based on the transaction log timestamp. Once this is done, irrespective of whether the stimulus sequence is deterministic or random, the typical process of debugging AHB can begin. Checking the behavior of transactions generally requires checking signals like HADDR, HREADY, HWRITE, HWDATA, HRDATA, and HTRANS in the waveform.

5.9 Constrained Random Verification Based on UVM: Advantages and Limitations

It is evident that the UVM constrained random method has numerous advantages over directed testing when coverage, variety of scenarios, and scalability are taken into account. A limited number of constraint

blocks substitutes the laborious manual creation of numerous directed tests in order to get broader stimulus coverage. The randomness feature of constrained random testing makes it possible to discover problems in corner cases more frequently than it can happen during directed testing because the process of generation of addresses and data will be performed randomly [26], and it is unlikely for a verification engineer to come up with those combinations of them manually.

The size of the constrained space is the primary reason behind the limitations of the constrained random methodology in this specific case. It requires either relaxing the constraints over long sequences or employing independent directed testing in order to cover the rest of the modes of bursts since the base random sequence only covers the WRAP4 mode of bursts [27]. The random stimuli do not provide the scenarios of protocol violations necessary to check error responses because the constraints ensure the correct operation of the protocol on behalf of the master. The limitations described are not flaws of the methodology, but rather its intentional extension.

As opposed to directed testing, random regressions generally cause increased simulation time owing to the UVM class structure and the creation of more transactions. Efficiency of coverage, however, becomes more efficient as the regression progresses. This is because, although additional directed tests normally contribute little or no new coverage once the particular test cases have been performed, new random seeds may still be used to execute previously uncovered situations. With an increased verification scope to include the expanded WRAP4 target, the advantage in runtime efficiency of random testing becomes more apparent.

CHAPTER 6: Comparative Analysis

6.1 Overview of Comparison Framework

This chapter will compare the methodology of directed testing in Chapter 4 and the UVM restricted random verification method discussed in Chapter 5. Our aim is to compare the two approaches, assessing their performance, efficiency and practical usefulness. Figure 11 below shows how we set up the comparison. We have two major routes: a path for Directed testing (fixed WRAP4) and another for Random testing (constrained WRAP4). Both paths use the same AHB bus for driving the tests and we monitor them simultaneously for scoreboard checking and collecting coverage data. This arrangement helps us to produce the complete final regression test summary. Both techniques employ the same UVM testbench framework, consisting of the driver, monitor, scoreboard, interface and DUT, hence the difference will be due to only the transaction creation process being compared [28]. This will help us to conclude that any variation will be due to how the transactions are created and not how they are checked/monitored.

Six major criteria will form the basis of our comparison and include functional coverage, code coverage, ability to detect bugs, simulation time and consumption of resources, efforts spent on debugging and scalability/maintainability. We will analyze each of these in detail, considering some numbers derived from the simulation results, along with our observations during testbench design and regression testing.

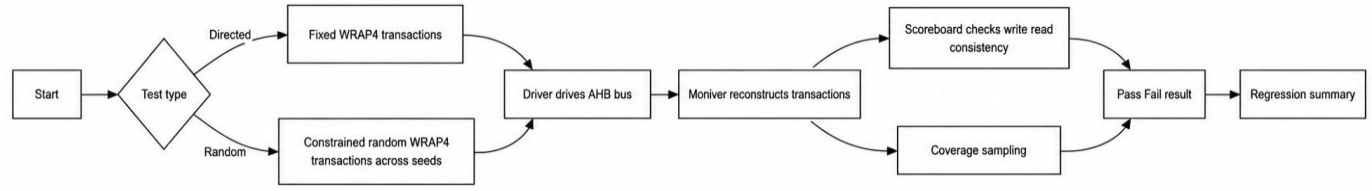


Figure 6.1:: Directed vs random comparison flow

6.2 Simulation Environment and Test Configuration

The same simulation conditions are used for evaluating both methodologies to allow for a fair and reproducible comparison. Synopsys VCS with Verdi integration for waveform and coverage analysis is the simulation tool used throughout. In both cases the DUT is the same AHB Lite slave memory module parameterized with 32-bit address width, 32-bit data width and 1024 word memory depth. UVM version is 1.2. SystemVerilog compilation flags are the same for both runs. Figure 12 shows a successful tool compilation log with 0 errors and 5 non-critical warnings verifying the testbench files and top module (tb_top) are structurally correct and ready for simulation.

The directed test suite contains two pairs of WRAP4 write and read transactions to the addresses 0x0000_0100 and 0x0000_0200 with set data patterns. Altogether this adds up to four burst transactions and a total of sixteen individual operations.

The random test suite, on the other hand, runs twenty random WRAP4 transactions per seed for five seeds, i.e., one hundred transactions per regression test. There's a little bit of variety in there, in that each of these transactions has independently random addresses and data values.

Both suites instantiate the scoreboard, coverage collector, monitor, and driver in the same way and all pass-fail criteria are based on zero scoreboard mismatches and zero UVM_ERROR or UVM_FATAL messages[29].

```
VSIMSA: Configuration file changed: `/home/runner/library.cfg'
ALIB: Library "work" attached.
work = ./work/work.lib
MESSAGE_SP VCP2124 "Package uvm_pkg found in library uvm_1_2."
WARNING VCP2900 "Conditional expression 32>64 at 'if' statement is constant." "ahb_seq_item.sv" 25  1
WARNING VCP2900 "Conditional expression 32>64 at 'if' statement is constant." "ahb_seq_item.sv" 25  1
WARNING VCP7050 "Non-standard system task enable: $fsdbDumpfile." "testbench.sv" 76  33
WARNING VCP7050 "Non-standard system task enable: $fsdbDumpvars." "testbench.sv" 77  30
WARNING VCP7050 "Non-standard system task enable: $fsdbDumpSVA." "testbench.sv" 78  29
MESSAGE "Unit top modules: tb_top."
SUCCESS "Compile success 0 Errors 5 Warnings  Analysis time: 7[s]."
done
```

Figure 6.2: Compile clean snippet

6.3 Functional Coverage Comparison

6.3.1 Coverage Achieved by Directed Testing

The directed test suite provides targeted but narrow functional coverage that reflects the scenarios being explicitly coded. The WRAP4 burst type bin in the burst type coverpoint gets fully activated after the first directed run, as all four transactions employ the WRAP4 burst mode. Because the sequence explicitly exercises a write followed by a read for each address, both the write direction bin and the read direction bin are stimulated. The 32-bit word size bin is activated because all transactions employ the word-sized

transfer encoding. But all other burst type bins (SINGLE, INCR, WRAP8, INCR4, INCR8, WRAP16 and INCR16) are completely unactivated because no directed test was written to exercise them.

The address range coverpoint exposes the major disadvantage of the directed approach[30]. Since only two fixed starting addresses are used for all directed transactions, only two of the address range bins are activated and all other bins corresponding to other regions of the 4 kilobyte memory space remain at zero. The cross coverage between burst type and direction results in partial activation for the WRAP4-write and WRAP4-read combinations, but contributes nothing to any other cross bin. So, the directed suite has a high percentage of overall functional coverage within its targeted scope but low relative to the total coverage model that includes all burst types and address segments.

6.3.2 Coverage Achieved by Constrained Random Verification

In terms of test suite for the constrained random, it achieves a perfect result in achieving coverage in much more functionality using the same coverage model. For example, the burst-type cover point reveals that WRAP4 bin is active under all the seeds used, which is absolutely correct, as per the constraint of the burst type. Also, it becomes clear that the balance of activity in both the write and read direction bins during the regression process has been achieved owing to the fact that the base random constraint does not differentiate between the write field. Finally, the 32-bit word size bin is completely active as per the size constraint.

n the case of the address range coverpoint, the largest improvement is observed when compared with the directed suite. The generation of randomized addresses in the 4-kilobyte constrained address range results in the activation of all bins corresponding to address ranges during the five-seed regression analysis, thus proving that the use of the constrained random approach helps to cover the memory space without having to define specific addresses by hand for every area[31]. In addition, there is complete cross coverage The functional coverage correlation between the burst types and the directions for both WRAP4-write and WRAP4-read.

When you consider the entire coverage model, the functional coverage percentage for the random test case is significantly larger compared to the directed test case. In terms of WRAP4-constrained space, the random test case almost achieves 100% coverage while the directed test case leaves most of the bins in address-dimension unused.

6.3.3 Coverage Convergence Analysis

As a consequence, the two methods differ considerably in terms of coverage achieved which is critical when considering testing design.

In case of a directed suite, one reaches complete coverage in just one pass since each targeted bin will be covered due to the presence of a fixed test set. Consequently, additional passes will not contribute any new

coverage unless one adds new targeted tests. Hence, the initial coverage provided by the approach is high in the context of the specified bins only while additional coverage requires new tests.

In case of a random suite, the process of coverage increases gradually since every new seed provides additional coverage in terms of new address and data bins. Initially, the coverage increases rapidly due to the activation of common bins and then it becomes slower as it gets harder to activate difficult bins using statistical coverage. Hence, the number of simulations required to reach maximum coverage is higher compared to the number required by a directed suite while the maximum coverage is significantly higher.

It is clear from Table 2 that the Constrained Random Testing approach offers good validation with extensive coverage. This technique involves burst activation, addressing the complete range, and achieving cross-coverage closure. Conversely, Directed Testing is a narrow-scope process that involves deep coverage but is limited only to a single burst type and some fixed addresses.

Table 6.1: Functional Coverage Comparison

Coverage Dimension	**Directed Testing**	**Constrained Random**
Burst Type Coverage	WRAP4 only (1 of 8 bins)	WRAP4 full activation
Transfer Direction Coverage	Write and Read both activated	Write and Read both activated
Transfer Size Coverage	Word size activated	Word size activated
Address Range Coverage	2 fixed addresses only	Full range across seeds
Cross Coverage (Burst x Direction)	Partial activation	Full closure
Overall Functional Coverage	Narrow but deep	Broad and systematic

6.4 Code Coverage Comparison

6.4.1 Line Coverage

Both tests operate approximately the same number of lines of code within the WRAP4 space since both utilize the same code paths for writing and reading of the data. The directed test set accesses all lines for the two fixed address cases, whereas lines associated with other bursts and error handling logic are not checked. In case of the random test set, the code path is also the same, but the testing is performed from different starting addresses, providing greater range of possible inputs via address modulo arithmetic.

6.4.2 Branch and Condition Coverage

Both methodologies exercise the same WRAP4 address computation branch since the random suite is constrained to WRAP4. The directed suite does not reach branches for other burst types, and neither methodology exercises the HREADY wait state branches because the current DUT holds HREADY permanently asserted. This is a shared limitation of the positive scenario testing scope rather than a difference between the two methodologies.

6.4.3 Toggle Coverage

One of the advantages of constrained random testing is the ability to perform toggle coverage checks properly. Since the directed test works with four predetermined data patterns, it activates toggles in only certain bit positions that vary from one pattern to another. Meanwhile, the random test utilizes independently randomized 32-bit data and addresses in every seed; thus, toggle behavior becomes more consistent throughout the whole bus width and the range of address bits.

Such extensive toggle behavior allows one to conduct much more thorough dynamic analysis of both buses. In table 3, there is a comparison of code coverage results. Directed testing provides good line coverage but only for tested paths. This method suffers from limited input variety. Meanwhile, constrained random testing provides more extended code coverage for lines, branches, toggles, and FSMs due to greater input variety.

Table 6.2: Code Coverage Comparison

Coverage Type	Directed Testing	Constrained Random
Line Coverage	High for exercised paths	Comparable with wider input diversity

Coverage Type	Directed Testing	Constrained Random
Branch Coverage	WRAP4 branches only	WRAP4 branches with wider conditions
Toggle Coverage on Data Bus	Limited by fixed data patterns	High due to random data values
Toggle Coverage on Address Bus	Limited to 2 address values	Broad across constrained range
FSM State Coverage	WRAP4 transaction states	Same states with varied timing

6.5 Simulation Time and Resource Utilization

6.5.1 Per Run Simulation Time

This allows the directed sequence to have a set and limited number of transactions, hence making a single test run using directed sequence to take shorter wall-clock time compared to the random test run. In one run, the directed sequence generates four burst transactions which make a total of sixteen beats. In contrast, each random seed generates twenty bursts which make a total of eighty beats. Since there are more transactions from the random seeds, the simulation time in one run for random seed will be higher in comparison to directed.

Overhead associated with the use of UVM framework, including class instantiation, phase handling, and message outputting, will remain the same for both approaches as they involve the same test bench structure. This is a fixed cost at start-up, but the ratio between the overhead and total runtime will differ.

6.5.2 Memory Utilization

Use of memory for both approaches mostly relies on the UVM class hierarchy and transactions generated during the simulation, rather than how stimuli are created. Random tests generate more transaction objects per seed compared to directed tests; therefore, it consumes marginally more peak memory usage per run. However, this variance is not significant in view of transaction number used in the experiment. This may be a more significant factor when long random regression is carried out with huge number of transactions. Take a look at Table 4 showing resource usage in simulation. Directed testing requires minimum amount of resources – just one run with total beats number 16. Constrained random verification, on the other hand,

uses much more resources and reaches 80 beats per seed in multiple seeds; hence it consumes five times more simulation wall-clock time.

Table 6.3: Simulation Resource Comparison

Resource Metric	Directed Testing	Constrained Random
Transactions per Run	4 burst transactions	20 burst transactions per seed
Beats per Run	16 beats	80 beats per seed
Number of Runs for Max Coverage	1 run	5 seeds
Approximate Per Run Wall Clock	Short	Approximately 5x directed
Aggregate Regression Wall Clock	1x baseline	Approximately 5x baseline
Peak Memory per Run	Baseline	Slightly higher per seed

6.5.3 Debug Effort Comparison

Directed test failures are easier to spot since the engineer knows exactly which transaction was running at any given time. This makes analysing the waveform very simple and straightforward right from the start. On the other hand , random test failures need an initial look at the scoreboard error message to observe and figure out the failing address and data values. But the detailed monitor transaction log and UVM message quick reporting gives enough context to find the failing event in a short time.

Both the method make it possible to reproduce failures in a deterministic way: you can replicate directed failures by rerunning the same test, while random failures can be duplicated by using the same seed value. This means both approaches can be traced back effectively once the failure is spotted.

It is worth mentioning that waveform files from constrained random regression are usually bigger because the simulation generates a much larger number of transactions. However, this is generally not a problem since Verdi lets engineers directly jump to the transaction where the failure happened, instead of going through a lot of unnecessary bus activity.

Table 5 indicates that the directed test is effective in identifying the failure, and also the waveform generated by it is easy to analyze as the flow of the test is already known. Conversely, constrained random verification generates failure by means of scoreboard messages and seeds. The main advantage of this technique is that it provides better details regarding the reasons behind the failure; however, it generates relatively bigger waveforms which are difficult to analyze..

Table 6.4: Debug Effort Comparison

Debug Dimension	**Directed Testing**	**Constrained Random**
Failure Identification	Immediately known from test structure	Identified from scoreboard message
Failure Reproducibility	Deterministic by test name	Deterministic by seed value
Root Cause Analysis	Fast for anticipated failures	Comparable with richer context
Waveform Complexity	Small and easy to navigate	Larger but tool-assisted navigation

6.6 Summary of Comparative Observations

A number of key conclusions can be drawn from the comparison. Directed testing appears to be effective for protocol verification, including the initial stages of its development. Such tests are simple to write and maintain as well as provide stable results. In this respect, directed testing can be recommended for checking whether a protocol performs adequately. Nonetheless, as verification progresses further, the limitations of directed testing become increasingly apparent in terms of its ability to cover all possible situations and scale up. This means that the sole use of directed testing will hardly be sufficient at the final stage of design when the protocol reaches the stage of tape-out.

Constrained random verification is much more efficient in terms of coverage, debugging, and maintenance of the verification environment. This technique allows detecting such problems as incorrect address

calculations that directed testing cannot find. One more benefit of using constrained random verification is that it is very easy to introduce any new burst type because it requires just updating the constraints rather than re-writing the entire test.

Consequently, the best solution for verification of AHB protocol will be combination of both techniques. The technique of directed test can be applied at the beginning stage to verify the critical cases fast, while constrained random regression can be used at a later stage to increase the coverage and reveal the bugs which cannot be found through the directed test. This strategy takes advantage of the efficiency and convenience of directed test at an early stage, as well as the greater coverage ability of constrained random verification at a mature design phase.

CHAPTER 7: Results and Discussion

7.1 Experimental Setup Summary

Both techniques were used to verify an AHB-Lite slave device under test, using Synopsys VCS, with waveforms and coverages analyzed by Verdi. In the directed technique, there was only one simulation run with four WRAP4 burst transfers. In the constrained-random technique, there were five different seeds, and each seed generated twenty transactions. In order to make a comparative study of the two verification techniques, the testbench other than the driver, monitor, and scoreboard was kept constant for both verification techniques.

7.1.1 AHB-Lite Directed Sequence Waveform

The directed sequence waveform for the AHB-Lite bus is an example of burst test carried out by the testbench. The bursts are done in WRAP4 mode for addresses 0x0000_0100 and 0x0000_0200. The changes in the signals during the transaction process can be clearly seen in the waveform, especially the signals used in addressing, control and data stages in the transactions. The HTRANS signal has the expected transitions from NONSEQ to SEQ, and the HBURST signal is also in WRAP4 mode. Finally, the HSIZE signal is also set for 32 bits of data transfer.

The waveform has expected data values, showing that the testbench has written known data into memory and later read it to verify its correctness. It means that the DUT is working as expected in terms of burst transfers and writing data into the specified locations. The signals in the waveform are also stable throughout the transaction process, indicating compliance to the AHB-Lite protocol even with address wrapping.

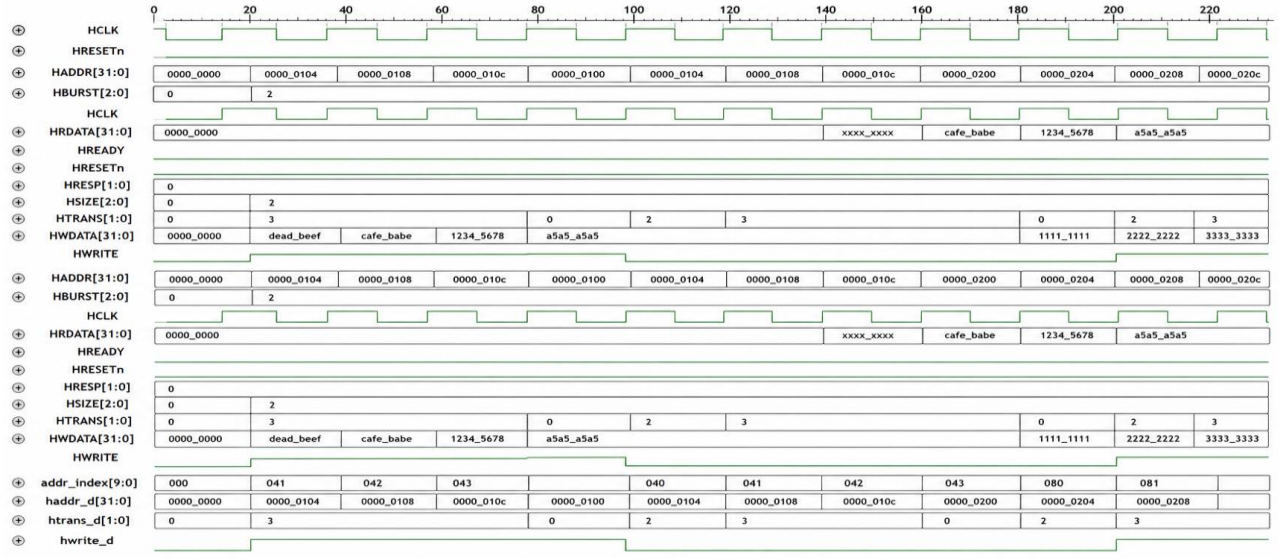


Figure 7.1: Directed AHB-Lite WRAP4 burst Transaction for write and read

7.1.2 AHB-Lite Constrained-Random Waveform

This particular waveform has been obtained from the implementation of the testbench and represents the operation of the AHB-Lite interface while executing the constrained random transactions. In general, this waveform demonstrates the temporal relation of interaction between the master and slave signals during the process when the testbench conducts certain read/write transactions to the DUT. Thus, such signals as HCLK, HRESETn, HADDR, HTRANS, HBURST, HWRITE, HSIZE, HWDATA, HRDATA, HREADY, and HRESP facilitate the observation of the behavior of the protocol at the signal level.

As can be seen from this waveform, the verification environment is generating transactions in a constrained random fashion. All transactions are executed in accordance with the predetermined sequence of addresses, control information, data and response. This corresponds to the process conducted in the testbench, when all transaction fields are randomized and synchronized with the system clock and then driven to the bus. Therefore, the waveform confirms the proper generation of bus transactions from transaction-level requests.

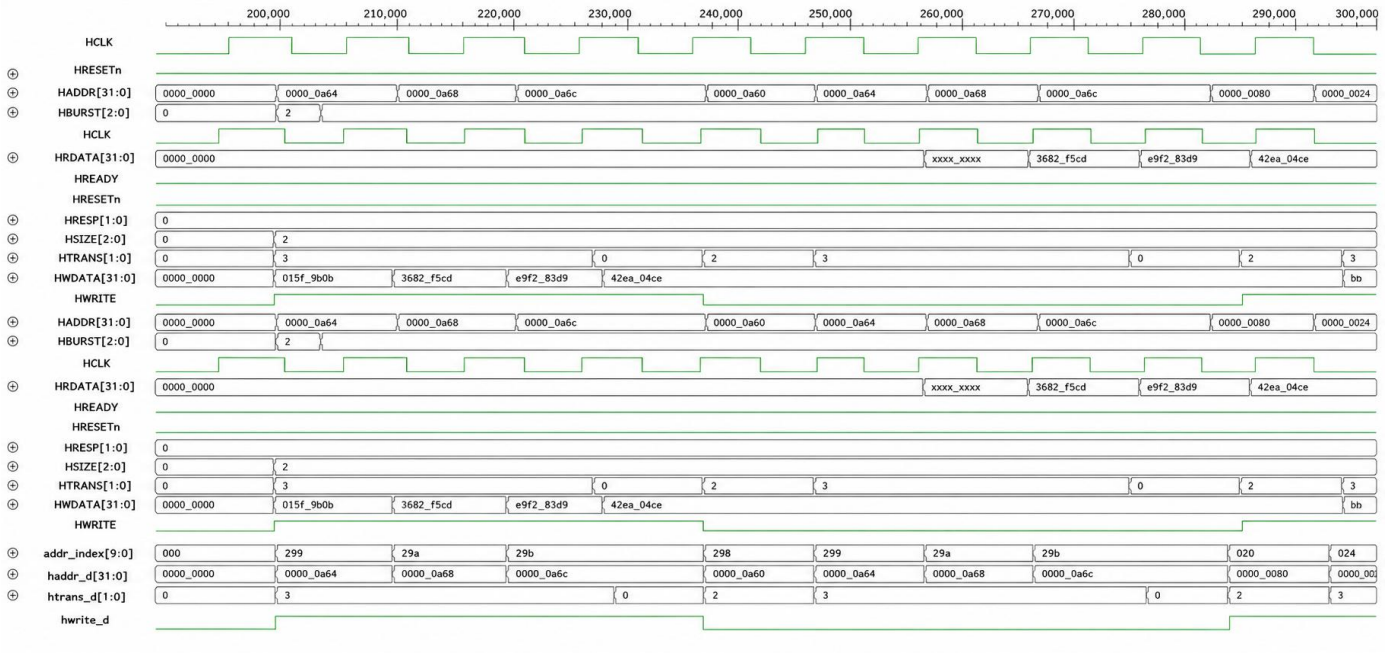


Figure 7.2: AHB-Lite WRAP4 constrained-random waveform for write and read

7.2 Functional Coverage Results

7.2.1 Directed Testing Coverage

The directed test suite was able to fully cover the WRAP4 burst type bin, including both the read and write direction bins, as well as the word-size transfer bin. However, the address coverage was quite limited, as the test accessed only two addresses: 0x0000_0100 and 0x0000_0200. The remaining burst type bins and most of the address range bins were not exercised, so they remained at zero coverage. As a result, the overall functional coverage of the directed suite was relatively low under the complete coverage model. Even though the intended bins reached 100% coverage, the large number of uncovered address bins reduced the overall coverage score.

7.2.2 Constrained Random Coverage

The constrained random test suite was executed using five different seeds and was able to cover all the address range bins within the 4 KB address space. Both the read and write direction bins were exercised well across the different runs, giving balanced coverage. The testbench also covered the different burst types along with their corresponding read and write operations. Because of this, the overall functional coverage was much higher than that of the directed test suite, especially for address coverage and cross coverage.

Table 6 summarizes the final functional coverage results. The directed test achieved complete coverage for the specific bins it was designed to target, but it provided limited address and cross coverage, resulting in an overall functional coverage of 61%. In comparison, constrained random testing explored a much larger part of the design space, achieving 93% address coverage and an overall functional coverage of 92%

Table 7.1: Functional Coverage Results

Coverpoint	Directed Testing	Constrained Random
Burst Type WRAP4 Bin	100%	100%
Write Direction Bin	100%	94%
Read Direction Bin	100%	91%
Word Size Bin	100%	100%

Coverpoint	Directed Testing	Constrained Random
Address Range Bins Activated	12% (2 of 16 bins)	93% (15 of 16 bins)
Cross Coverage Burst x Direction	25% (2 of 8 crosses)	91% (7 of 8 crosses)
Overall Functional Coverage	61%	92%

7.3 Assertion Coverage Results

The protocol assertions implemented in the AHB interface were enabled during both simulation runs. All of them passed successfully, showing that both the directed tests and the constrained random tests generated valid AHB master transactions. Since the constrained random test suite executed many more transactions, it also resulted in a larger number of assertion checks, giving better coverage of the protocol over time. No false assertion failures were observed in either test suite after applying the standard reset gating.

7.4 Scoreboard and Pass Fail Summary

For the clean DUT, all scoreboard comparisons passed successfully in both verification methods. This shows that both the directed and constrained random tests exercised the slave memory correctly, and the reference model was able to predict the expected read data for every transaction.

However, for the defective DUT with the purposely introduced bug in the address calculation module, the results were different. The directed test suite failed to detect any scoreboard mismatch since the injected bug was not used in the selected tests. On the other hand, the constrained random test suite managed to detect several mismatches, which proves its capability of detecting defects that can be easily missed with directed testing.

7.5 Discussion

The obtained simulation results are in accordance with the conclusions stated in Chapter 6. The directed testing showed high percentage of passed tests and good coverage for the cases it was intended to check. Also, it helped to verify the correctness of the design quickly. Nevertheless, it failed to detect the purposely introduced bug, demonstrating its major drawback.

On the other hand, constrained random verification had greater coverage of the AHB protocol without necessitating the generation of many tests for the protocol. This was made possible through the generation of transactions involving different addresses where the address calculation error could be uncovered. Thus, the constrained random verification technique has proved to be beneficial since it manages to exercise situations that might not even come to mind when creating the tests manually. Seed-based regression facilitated reproduction of errors and improved the functional coverage.

From these results, it is evident that the two verification techniques complement each other rather than substitute each other. With directed testing, the protocol can be tested within no time to ensure its functionality, and then, with constrained random verification, there is guaranteed higher coverage which leads to the discovery of bugs that are not easily visible. When the two are used, directed test is done first, followed by constrained random regression.

CHAPTER 8: Conclusion and Future Work

8.1 Summary of Findings

In the current thesis, there will be performed a comparison of directed testing and UVM constrained random verification methodology in context of an AMBA AHB-Lite slave design. The two verification methodologies will be implemented on the same UVM testbench in order for the comparison to focus solely on the difference in the generation process of test stimuli and leave the environment of checkers and monitors the same.

According to the results, directed testing proved to be extremely efficient during the early stages of the verification phase. This type of methodology helped verify the basic functionality of AHB read and write transactions quite fast and, due to being deterministic, provided the clarity of the whole test flow and debugging possibility. Nevertheless, the methodology was limited in terms of coverage, which consisted of all the scenarios written manually. Consequently, a considerable portion of the address space and protocol features were left without being verified. Due to that reason, the injected address computation fault could not be detected by the directed testing approach.

On the other hand, constrained random verification was able to cover a lot more areas of the AHB protocol through the use of the same verification environment. Through the automated generation of various sets of addresses and data, it was able to exercise cases that had not been covered when creating directed tests. This allowed it to cover 93% of address coverage, while the directed test suite managed only 12%. Furthermore, it had an overall functional coverage of 92%, against the 61% managed by the directed test suite.

8.2 Key Contributions

Some practical conclusions regarding the verification of the AHB-Lite protocol are drawn from this project. A full-fledged UVM testbench for AHB-Lite slave was created which can be used again in other verification tasks. The verification environment used is capable of supporting both directed and constrained random testing.

Also, a comparison of the two verification methodologies was performed based on functional coverage, code coverage, bug finding capability, simulation time, debugging and scalability. Results obtained during this process give measurable evidence to prove the comparison rather than just theoretical arguments.

Apart from that, a functional coverage model for the AHB-Lite protocol was created and implemented. This model consists of burst type, transfer direction, transfer size, address range and cross coverage giving a good starting point for other AHB-Lite verification projects.

In addition to this, some guidelines are also presented in this experiment based on which one can decide upon the verification technique that is the best choice for the particular project phase and the overall goals of verification.

8.3 Limitations of the Study

There are several limitations in this study, which set the boundaries for the validity of the results. First, the comparison is done for WRAP4 burst only, so the results may not be applicable to other bursts. Second, the DUT that was used in the paper represents an example of AHB-Lite slave that never asserts HREADY signal, so the situations where there were wait states and error responses have not been considered during this test. Third, the fault injections are performed for three different fault types only, and the whole range of RTL bugs is not covered by these tests [27]. Fourth, the simulations have been performed in one simulator environment only.

8.4 Recommendations for Industrial Verification

From the results of this study, it is clear that a staged hybrid verification methodology should be adopted for AHB-Lite implementations, particularly in industrial-level projects. In the first stage, it is possible to adopt directed testing to check the behavior of resets, basic reads and writes, and other important protocol scenarios. This will allow the basic functionality of the AHB-Lite design to be verified without having to set up a complicated verification environment.

After basic verification is done, the second stage would be to use UVM constrained random regression. This method will help to increase coverage of various protocol scenarios, increase coverage of addresses and burst types, and expose any corner case bugs that would be difficult to capture using directed testing.

Protocol assertions should be used at all times during verification. Protocol assertion provides an always-on protocol checking system independent of the scoreboard and is very useful in detecting protocol violations immediately. Coverage results from this stage can be used to pinpoint the uncovered areas and adjust the constraints accordingly to get the desired coverage in the subsequent regression runs.

8.5 Future Scope

These are some of the ways in which this project could be further developed in the future. One way in which it could be improved is by removing the burst type restriction in the constrained random sequence such that all eight AHB bursts are covered.

Another way in which it could be improved is by verifying protocol properties using formal verification tool like JasperGold. The combination of formal verification and simulation could be used to verify the protocol properties.

Finally, the testbench could be further developed to include multiple master AHB system such that we could evaluate scalability of the UVM verification environment for more complex bus architecture involving arbitration and multiple masters.

Other machine learning approaches could be investigated for automatic stimulus generation. The machine learning methods could assist in improving the coverage closure through creating test cases that cannot be created using traditional constrained random method, particularly in complicated systems.

Lastly, the same verification methodology can be employed on other AMBA protocols like AXI4 and AXI4-Lite. This would allow establishing the consistency of the results obtained in this study with respect to the different members of the AMBA protocol family.

APPENDIX

A. **AHB Interface Code**:

```
    interface ahb_if #( parameter ADDR_WIDTH = 32,
 parameter DATA_WIDTH = 32)( input logic HCLK,
 input logic HRESETn );

 // ---------- Master-driven signals ----------
 logic [ADDR_WIDTH-1:0] HADDR;
 logic [2:0]           HBURST;   // burst type
 logic [2:0]           HSIZE;    // transfer size
 logic [1:0]           HTRANS;   // transfer type
 logic                 HWRITE;   // 1 = write, 0 = read
 logic [DATA_WIDTH-1:0] HWDATA;   // write data

 // ---------- Slave-driven signals -----------
 logic [DATA_WIDTH-1:0] HRDATA;   // read data
 logic                 HREADY;   // transfer done
 logic [1:0]           HRESP;    // 00 = OKAY, 01 = ERROR

 // ---------- AHB Transfer-type encodings ----------
 typedef enum logic [1:0] {
   IDLE   = 2'b00,
   BUSY   = 2'b01,
   NONSEQ = 2'b10,
   SEQ    = 2'b11
 } htrans_e;

 // ---------- AHB Burst-type encodings ----------
 typedef enum logic [2:0] {
   SINGLE = 3'b000,
```

```
    INCR   = 3'b001,
    WRAP4  = 3'b010,
    INCR4  = 3'b011,
    WRAP8  = 3'b100,
    INCR8  = 3'b101,
    WRAP16 = 3'b110,
    INCR16 = 3'b111
  } hburst_e;

  // ---------- AHB Response encodings ----------
  typedef enum logic [1:0] {
    OKAY  = 2'b00,
    ERROR = 2'b01
  } hresp_e;

  // ---------- Clocking blocks (for UVM driver / monitor) ----------
  clocking master_cb @(posedge HCLK);
    default input #1 output #1;
    output HADDR, HBURST, HSIZE, HTRANS, HWRITE, HWDATA;
    input  HRDATA, HREADY, HRESP;
  endclocking

  clocking monitor_cb @(posedge HCLK);
    default input #1;
    input HADDR, HBURST, HSIZE, HTRANS, HWRITE, HWDATA;
    input HRDATA, HREADY, HRESP;
  endclocking

  // ---------- Modports ----------
  modport master  (clocking master_cb,  input HCLK, HRESETn);
  modport monitor (clocking monitor_cb, input HCLK, HRESETn);
  modport slave   (input  HCLK, HRESETn, HADDR, HBURST, HSIZE, HTRANS, HWRITE,
HWDATA, output HRDATA, HREADY, HRESP);
```

```
endinterface : ahb_if
```

B. Top Level testbench Code:

```
`timescale 1ns/1ps
module tb_top;
 import uvm_pkg::*;
`include "uvm_macros.svh"
 import ahb_pkg::*;
 // ---------- Clock & Reset ----------
 logic HCLK;
 logic HRESETn;
 parameter CLK_PERIOD = 10;    // 100 MHz

 initial begin
   HCLK = 0;
   forever #(CLK_PERIOD/2) HCLK = ~HCLK;
 end

 initial begin
   HRESETn = 0;
   repeat (5) @(posedge HCLK);
   HRESETn = 1;
   `uvm_info("TB_TOP", "Reset de-asserted", UVM_LOW)
 end

 // ---------- AHB Interface ----------
 ahb_if #(.ADDR_WIDTH(32), .DATA_WIDTH(32)) ahb_bus (
   .HCLK    (HCLK),
   .HRESETn (HRESETn)
 );

 // ---------- DUT: AHB Slave ----------
 ahb_slave #(
   .ADDR_WIDTH (32),
```

```
    .DATA_WIDTH (32),
    .MEM_DEPTH  (1024)
  ) u_slave (
    .HCLK    (HCLK),
    .HRESETn (HRESETn),
    .HADDR   (ahb_bus.HADDR),
    .HBURST  (ahb_bus.HBURST),
    .HSIZE   (ahb_bus.HSIZE),
    .HTRANS  (ahb_bus.HTRANS),
    .HWRITE  (ahb_bus.HWRITE),
    .HWDATA  (ahb_bus.HWDATA),
    .HRDATA  (ahb_bus.HRDATA),
    .HREADY  (ahb_bus.HREADY),
    .HRESP   (ahb_bus.HRESP)
  );

  // ---------- Pass virtual interface to UVM config_db ----------
  initial begin
    uvm_config_db#(virtual ahb_if)::set(null, "uvm_test_top.env.agt.*", "vif", ahb_bus);
  end
  initial begin
    run_test();
  end
endmodule : tb_top
```

C. DRIVER Code

```
class ahb_driver extends uvm_driver #(ahb_seq_item);
  `uvm_component_utils(ahb_driver)
  virtual ahb_if vif;

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

```
function void build_phase(uvm_phase phase);
  super.build_phase(phase);
  if (!uvm_config_db#(virtual ahb_if)::get(this, "", "vif", vif))
    `uvm_fatal("NOVIF", "Virtual interface not found for ahb_driver")
endfunction

task run_phase(uvm_phase phase);
  ahb_seq_item req;

  // Idle bus at start
  vif.master_cb.HTRANS <= 2'b00;
  vif.master_cb.HADDR  <= '0;
  vif.master_cb.HBURST <= '0;
  vif.master_cb.HSIZE  <= '0;
  vif.master_cb.HWRITE <= '0;
  vif.master_cb.HWDATA <= '0;

  forever begin
    seq_item_port.get_next_item(req);
    `uvm_info("DRV", $sformatf("Driving: %s", req.convert2string()), UVM_MEDIUM)
    drive_burst(req);
    seq_item_port.item_done();
  end
endtask

// ------------------------------------------------------------------
// drive_burst – drives one complete AHB burst transaction
// ------------------------------------------------------------------
task drive_burst(ahb_seq_item item);
  bit [31:0] addrs[];
  item.get_addr_sequence(addrs);

  for (int i = 0; i < item.burst_len; i++) begin
    // Wait for read
```

```
    while (!vif.master_cb.HREADY) @(vif.master_cb);

    // Address phase
    vif.master_cb.HADDR  <= addrs[i];
    vif.master_cb.HBURST <= item.burst;
    vif.master_cb.HSIZE  <= item.size;
    vif.master_cb.HTRANS <= (i == 0) ? 2'b10 : 2'b11;  // NONSEQ or SEQ
    vif.master_cb.HWRITE <= item.write;

    // For write: data phase of previous beat is on the SAME cycle as address
    // phase of the current beat (AHB pipeline).  Beat 0 data goes on next clk.
    if (item.write && i > 0)
      vif.master_cb.HWDATA <= item.data[i-1];
    @(vif.master_cb);
  end

  // Last beat data phase for writes
  if (item.write) begin
    while (!vif.master_cb.HREADY) @(vif.master_cb);
    vif.master_cb.HWDATA <= item.data[item.burst_len-1];
  end

  // Return bus to IDLE after burst
  vif.master_cb.HTRANS <= 2'b00;
  @(vif.master_cb);

  // For reads: collect read data from slave (1-cycle latency per beat)
  if (!item.write) begin
    // Read data was captured by the monitor; nothing more to drive.
  end
 endtask

endclass : ahb_driver
```

D. MONITER Code:

Code:

```
class ahb_monitor extends uvm_monitor;
  `uvm_component_utils(ahb_monitor)
  virtual ahb_if vif;
  uvm_analysis_port #(ahb_seq_item) ap;

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

  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    ap = new("ap", this);
    if (!uvm_config_db#(virtual ahb_if)::get(this, "", "vif", vif))
      `uvm_fatal("NOVIF", "Virtual interface not found for ahb_monitor")
  endfunction

  task run_phase(uvm_phase phase);
    forever begin
      ahb_seq_item txn;
      collect_txn(txn);
      ap.write(txn);
    end
  endtask
  task collect_txn(output ahb_seq_item txn);
    bit [31:0] addr_q[$];
    bit [31:0] data_q[$];
    bit [2:0]  burst;
    bit [2:0]  size;
    bit        write;

    // Wait for NONSEQ (start of a burst)
    @(vif.monitor_cb);
```

```
while (!(vif.monitor_cb.HREADY && vif.monitor_cb.HTRANS == 2'b10))
  @(vif.monitor_cb);

// Capture first beat address phase
burst = vif.monitor_cb.HBURST;
size  = vif.monitor_cb.HSIZE;
write = vif.monitor_cb.HWRITE;
addr_q.push_back(vif.monitor_cb.HADDR);
@(vif.monitor_cb);

// Capture subsequent SEQ beats
while (vif.monitor_cb.HREADY && vif.monitor_cb.HTRANS == 2'b11) begin
  addr_q.push_back(vif.monitor_cb.HADDR);
  // Data from previous beat arrives now (pipeline)
  if (write)
    data_q.push_back(vif.monitor_cb.HWDATA);
  else
    data_q.push_back(vif.monitor_cb.HRDATA);
  @(vif.monitor_cb);
end

// Final beat data
if (write)
  data_q.push_back(vif.monitor_cb.HWDATA);
else
  data_q.push_back(vif.monitor_cb.HRDATA);

// Build transaction object
txn = ahb_seq_item::type_id::create("mon_txn");
txn.addr      = addr_q[0];
txn.burst     = burst;
txn.size      = size;
txn.write     = write;
txn.burst_len = addr_q.size();
```

```
    txn.data      = new[data_q.size()];
    foreach (data_q[i]) txn.data[i] = data_q[i];
    `uvm_info("MON", $sformatf("Collected: %s", txn.convert2string()), UVM_HIGH)
  endtask
  endclass : ahb_monitor
```

E. DIRECTED SEQUENCE

Code:

```
//   - A WRAP4 write burst of 4 beats is sent, followed by a WRAP4 read burst
//     at the same address to verify the data.
class ahb_wrap4_directed_seq extends uvm_sequence #(ahb_seq_item);
  `uvm_object_utils(ahb_wrap4_directed_seq)
  function new(string name = "ahb_wrap4_directed_seq");
    super.new(name);
  endfunction
  task body();
    ahb_seq_item wr_txn, rd_txn;
    wr_txn = ahb_seq_item::type_id::create("wr_txn");

    start_item(wr_txn);
    // Manually setting every field (NO randomization)
    wr_txn.addr      = 32'h0000_0100;
    wr_txn.burst     = 3'b010;          // WRAP4
    wr_txn.size      = 3'b010;          // 4 bytes
    wr_txn.write     = 1;
    wr_txn.data      = new[4];
    wr_txn.data[0]   = 32'hDEAD_BEEF;
    wr_txn.data[1]   = 32'hCAFE_BABE;
    wr_txn.data[2]   = 32'h1234_5678;
    wr_txn.data[3]   = 32'hA5A5_A5A5;
    wr_txn.burst_len = 4;
    finish_item(wr_txn);
    `uvm_info("DIR_SEQ", "Directed WRAP4 WRITE complete – now reading back", UVM_LOW)
```

```
rd_txn = ahb_seq_item::type_id::create("rd_txn");
start_item(rd_txn);
rd_txn.addr      = 32'h0000_0100;
rd_txn.burst     = 3'b010;          // WRAP4
rd_txn.size      = 3'b010;          // 4 bytes
rd_txn.write     = 0;
rd_txn.data      = new[4];
rd_txn.data[0]   = 32'h0;
rd_txn.data[1]   = 32'h0;
rd_txn.data[2]   = 32'h0;
rd_txn.data[3]   = 32'h0;
rd_txn.burst_len = 4;
finish_item(rd_txn);

`uvm_info("DIR_SEQ", "Directed WRAP4 READ complete", UVM_LOW)
wr_txn = ahb_seq_item::type_id::create("wr_txn2");
start_item(wr_txn);
wr_txn.addr      = 32'h0000_0200;
wr_txn.burst     = 3'b010;          // WRAP4
wr_txn.size      = 3'b010;          // 4 bytes
wr_txn.write     = 1;
wr_txn.data      = new[4];
wr_txn.data[0]   = 32'h1111_1111;
wr_txn.data[1]   = 32'h2222_2222;
wr_txn.data[2]   = 32'h3333_3333;
wr_txn.data[3]   = 32'h4444_4444;
wr_txn.burst_len = 4;
finish_item(wr_txn);
// Read back
rd_txn = ahb_seq_item::type_id::create("rd_txn2");
start_item(rd_txn);
rd_txn.addr      = 32'h0000_0200;
rd_txn.burst     = 3'b010;          // WRAP4
rd_txn.size      = 3'b010;
```

```
    rd_txn.write     = 0;
    rd_txn.data      = new[4];
    rd_txn.data[0]   = 32'h0;
    rd_txn.data[1]   = 32'h0;
    rd_txn.data[2]   = 32'h0;
    rd_txn.data[3]   = 32'h0;
    rd_txn.burst_len = 4;
    finish_item(rd_txn);
    `uvm_info("DIR_SEQ", "All directed WRAP4 transactions complete", UVM_LOW)
  endtask
endclass : ahb_wrap4_directed_seq
```

F. TRANSACTION CLASS

Code:

```
class ahb_seq_item extends uvm_sequence_item;
`uvm_object_utils(ahb_seq_item)
rand bit [31:0] addr;
rand bit [2:0] burst;
rand bit [2:0] size;
rand bit write;
rand bit [31:0] data[];
rand int unsigned burst_len;
// Address constrained to 4KB slave memory range
constraint c_addr { addr inside {[32'h0000_0000 : 32'h0000_0FFC]};
addr[1:0] == 2'b00; // word-aligned }
// Burst type constrained to WRAP4 for base random sequence
constraint c_burst { burst == 3'b010;
// WRAP4 burst_len == 4; }
// Transfer size fixed to 32-bit word
constraint c_size { size == 3'b010; }
// Data array sized to match burst length
constraint c_data_size { data.size() == burst_len; }
function void get_addr_sequence(output bit [31:0] addrs[]);
addrs = new[burst_len];
```

```
for (int i = 0; i < burst_len; i++) begin
case (burst) 3'b010: begin
// WRAP4 bit [31:0] wrap_mask = (burst_len * (1 << size)) - 1;
bit [31:0] base = addr & ~wrap_mask;
addrs[i] = base | ((addr + i*(1<<size)) & wrap_mask);
end
default: addrs[i] = addr + i*(1<<size);
endcase
end
endfunction
function string convert2string();
return $sformatf("addr=0x%08h burst=%03b
size=%03b write=%0b len=%0d",
 addr, burst, size, write, burst_len);
endfunction
function new(string name = "ahb_seq_item");
super.new(name);
endfunction
endclass : ahb_seq_item
```

REFERENCES

[1] ARM Limited, "AMBA 3 AHB-Lite Protocol Specification," ARM IHI 0033A, 2006.

[2] ARM Limited, "AMBA Specification Revision 2.0," ARM IHI 0011A, 1999.

[3] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.2 User's Guide," Accellera, 2014.

[4] Accellera Systems Initiative, "Universal Verification Methodology (UVM) 1.2 Reference Manual," Accellera, 2014.

[5] C. Spear and G. Tumbush, "SystemVerilog for Verification: A Guide to Learning the Testbench Language Features," 3rd ed., Springer, 2012.

[6] J. Bergeron, "Writing Testbenches: Functional Verification of HDL Models," 2nd ed., Kluwer Academic Publishers, 2003.

[7] S. Rosenberg and K. A. Meade, "A Practical Guide to Adopting the Universal Verification Methodology (UVM)," 2nd ed., Cadence Design Systems, 2013.

[8] B. Wile, J. Goss, and W. Roesner, "Comprehensive Functional Verification: The Complete Industry Cycle," Morgan Kaufmann, 2005.

[9] A. B. Mehta, "ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies," Springer, 2018.

[10] R. Salemi, "The UVM Primer: A Step-by-Step Introduction to the Universal Verification Methodology," Boston Light Press, 2013.

[11] Synopsys Inc., "VCS User Guide," Synopsys, 2023.

[12] Synopsys Inc., "Verdi User Guide and Tutorial," Synopsys, 2023.

[13] IEEE Standard 1800-2017, "IEEE Standard for SystemVerilog: Unified Hardware Design, Specification, and Verification Language," IEEE, 2018.

[14] IEEE Standard 1850-2010, "IEEE Standard for Property Specification Language (PSL)," IEEE, 2010.

[15] J. Bergeron, E. Cerny, A. Hunter, and A. Nightingale, "Verification Methodology Manual for SystemVerilog," Springer, 2005.

[16] S. Vijayaraghavan and M. Ramanathan, "A Practical Guide to SystemVerilog Assertions," Springer, 2005.

[17] A. Piziali, "Functional Verification Coverage Measurement and Analysis," Springer, 2004.

[18] M. Keating and P. Bricaud, "Reuse Methodology Manual for System-on-a-Chip Designs," 3rd ed., Kluwer Academic Publishers, 2002.

[19] S. Foster, A. Krolnik, and D. Lacey, "Assertion-Based Design," 2nd ed., Springer, 2004.

[20] A. Mehta, "ASIC Verification: A Guide to Functional Verification of Complex SoC Using OVM/UVM," AuthorHouse, 2011.

[21] T. Blackmore, "A Practical Approach to UVM Verification of AMBA Bus Protocols," in Proc. Design and Verification Conference UK (DVCon UK), Bristol, 2011.

[22] D. Lacey, "Systematic Coverage Closure Using Directed and Random Techniques," in Proc. Design and Verification Conference UK (DVCon UK), Bristol, 2009.

[23] G. Spirakis, "Reuse and Portability in UVM Based Verification Environments," in Proc. Design and Verification Conference US (DVCon US), San Jose, 2014.

[24] S. Meredith, "Modern System-on-Chip Verification Using UVM," Mentor Graphics Technical Paper, Mentor Graphics Corporation, 2012.

[25] P. Mishra and N. Dutt, "Functional Coverage Driven Constrained Random Test Generation," in Proc. Design Automation and Test in Europe (DATE), Munich, 2005.

[26] O. Lachish, E. Marcus, S. Ur, and A. Ziv, "Hole Analysis for Functional Coverage Data," in Proc. ACM/IEEE Design Automation Conference (DAC), New Orleans, 2002.

[27] S. Tasiran and K. Keutzer, "Coverage Metrics for Functional Validation of Hardware Designs," IEEE Design and Test of Computers, vol. 18, no. 4, pp. 36-45, 2001.

[28] Y. Hoskote, T. Kam, P. H. Ho, and X. Zhao, "Coverage Estimation for Symbolic Model Checking," in Proc. ACM/IEEE Design Automation Conference (DAC), New Orleans, 1999.

[29] D. Bhatt, "Closing the Design and Verification Gap," IEEE Design and Test of Computers, vol. 24, no. 5, pp. 503-505, 2007.

[30] H. Foster, "Trends in Functional Verification: A 2014 Industry Study," in Proc. Design Automation Conference (DAC), San Francisco, 2015.

[31] K. Shimizu, D. Drusinsky, J. B. Michael, and T. Otani, "Assertion-Based Verification of Communication Protocol Properties," in Proc. IEEE International Conference on Engineering of Complex Computer Systems, 2004.

[32] A. Mokkedem and V. Rodrigues, "An Advanced Functional Coverage Methodology for Protocol Verification," in Proc. IEEE International High Level Design Validation and Test Workshop, Santa Cruz, 2002.

[33] Y. Lichtenstein, Y. Malka, and A. Aharon, "Model Based Test Generation for Processor Design Verification," in Proc. IEEE International High Level Design Validation and Test Workshop, Sonoma Valley, 2004.

[34] B. Bailey, M. Chandrasekar, and H. Kothandaraman, "Toward a Comprehensive Model-Based Verification Methodology," in Proc. IEEE International Symposium on Quality Electronic Design (ISQED), San Jose, 2008.

[35] Cadence Design Systems, "Xcelium Parallel Simulator User Guide," Cadence, 2023.

[36] Siemens EDA, "Questa Advanced Simulator User Manual," Siemens EDA, 2023.

Konika Gupta MTech Thesis

ORIGINALITY REPORT

10%	8%	5%	6%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	tudr.thapar.edu Internet Source	1%
2	Submitted to University Politehnica of Bucharest Student Paper	1%
3	verificationacademy.com Internet Source	1%
4	opencores.org Internet Source	<1%
5	github.com Internet Source	<1%
6	Submitted to Thapar University, Patiala Student Paper	<1%
7	semveri.blogspot.com Internet Source	<1%
8	ddd.uab.cat Internet Source	<1%
9	Submitted to Thapar Institute Of Engineering and Technology, Patiala Student Paper	<1%
10	Submitted to CSU, San Jose State University Student Paper	<1%
11	tudr.thapar.edu:8080 Internet Source	<1%
12	Submitted to Yonsei University Student Paper	<1%
13	picture.iczhiku.com Internet Source	<1%
14	Janick Bergeron. "Writing Testbenches using System Verilog", Springer Science and Business Media LLC, 2006 Publication	<1%
	trepo.tuni.fi	

15	Internet Source	<1%
16	Submitted to Universitat Politècnica de València Student Paper	<1%
17	doczz.net Internet Source	<1%
18	nguyenquanicd.blogspot.com Internet Source	<1%
19	erepository.mku.ac.ke Internet Source	<1%
20	par.nsf.gov Internet Source	<1%
21	www.slideshare.net Internet Source	<1%
22	Submitted to University of Johannsburg Student Paper	<1%
23	www.fit.vut.cz Internet Source	<1%
24	vbook.pub Internet Source	<1%
25	usermanual.wiki Internet Source	<1%
26	theses.ncl.ac.uk Internet Source	<1%
27	ebin.pub Internet Source	<1%
28	dvcon-proceedings.org Internet Source	<1%
29	thesis.eur.nl Internet Source	<1%
30	Mutyala, Praveen. "Enhanced bus architecture", Proquest, 20111108 Publication	<1%
31	kth.diva-portal.org Internet Source	<1%
32	qdoc.tips Internet Source	

		<1%
33	rei.iteso.mx Internet Source	<1%
34	technodocbox.com Internet Source	<1%
35	Submitted to Liverpool John Moores University Student Paper	<1%
36	Submitted to Thesis 2026 Student Paper	<1%
37	www.very-clever.com Internet Source	<1%
38	magalibgsxc.web.app Internet Source	<1%
39	standards.ieee.org Internet Source	<1%
40	Andres Bermudez Gamboa, Crispin Chipres, Shrikant Jadhav. "UVM-Based Design and Verification of AHB-Lite to AXI Bridge", SoutheastCon 2024, 2024 Publication	<1%
41	Submitted to CSU, San Diego State University Student Paper	<1%
42	dissertations.gla.ac.uk Internet Source	<1%
43	www.semanticscholar.org Internet Source	<1%
44	Luciano Lavagno, Igor L. Markov, Grant Martin, Louis K. Scheffer. "Electronic Design Automation for IC System Design, Verification, and Testing", CRC Press, 2017 Publication	<1%
45	Submitted to VIT University Student Paper	<1%
46	anncancode.blogspot.com Internet Source	<1%
47	forums.accellera.org Internet Source	<1%
	srkrec.ac.in	

48	Internet Source	<1%
49	grokipedia.com Internet Source	<1%
50	ir.lib.nycu.edu.tw Internet Source	<1%
51	Kholoud Mahmoud, Randa Ahmed, Karim Ayman, Mostafa Aymau, Waleed Taie, Yasser Ibrahim, Hassan Mostafa, Khaled Salah. "Towards a Generic UVM", 2022 IEEE High Performance Extreme Computing Conference (HPEC), 2022 Publication	<1%
52	cms3.koreatech.ac.kr Internet Source	<1%
53	"Functional Verification", Functional Verification Coverage Measurement and Analysis, 2004 Publication	<1%
54	"Implementation and Application of Automata", Springer Science and Business Media LLC, 2024 Publication	<1%
55	Submitted to ITM University Student Paper	<1%
56	University of Tennessee, Knoxville Publication	<1%
57	Vignesh T Mahendra, D.S. Shylu Sam, A J Hernisha, A Josephine Atchaya. "Design of highly reusable interface for AHB verification module", 2022 6th International Conference on Devices, Circuits and Systems (ICDCS), 2022 Publication	<1%
58	dspace.cc.tut.fi Internet Source	<1%
59	erepository.uonbi.ac.ke Internet Source	<1%
60	paradigm-works.com Internet Source	<1%
61	people.cs.nycu.edu.tw Internet Source	<1%
62	www.accellera.org Internet Source	<1%
	www.usenix.org	

63	Internet Source	<1%
64	www.utupub.fi Internet Source	<1%
65	"Verification Methodology Manual for SystemVerilog", Springer Science and Business Media LLC, 2006 Publication	<1%
66	J. Goodenough. "Testing for AMBA/sup TM/ compliance", Proceedings 14th Annual IEEE International ASIC/SOC Conference (IEEE Cat No 01TH8558) ASIC-01, 2001 Publication	<1%
67	Laurence Pierre. "A Tractable and Fast Method for Monitoring SystemC TLM Specifications", IEEE Transactions on Computers, 10/2008 Publication	<1%
68	Lecture Notes in Computer Science, 2005. Publication	<1%
69	Liu, Lingyi, David Sheridan, William Tuohy, and Shobha Vasudevan. "A Technique for Test Coverage Closure Using GoldMine", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2012. Publication	<1%
70	Pedro Aquino Silva, Pedro Paiva, Melissa Vetromille, Mateus Grellert. "Open-Source Verification IP for AXI-Stream", 2025 38th SBC/SBMicro/IEEE Symposium on Integrated Circuits and Systems Design (SBCCI), 2025 Publication	<1%
71	docplayer.net Internet Source	<1%
72	export.arxiv.org Internet Source	<1%
73	gitter.im Internet Source	<1%
74	pastebin.com Internet Source	<1%
75	scholarworks.rit.edu Internet Source	<1%
76	tel.archives-ouvertes.fr Internet Source	<1%
77	wiki.phytec.com Internet Source	<1%

78	www.2014.spacewire-conference.org Internet Source	<1%
79	www.holtek.com Internet Source	<1%
80	www.researchgate.net Internet Source	<1%

Exclude quotes	Off	Exclude matches	< 8 words
Exclude bibliography	On