UVM Environment Bring-Up for AXI4 Full Based DMA IP

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

MASTER OF TECHNOLOGY

in VLSI Design

Submitted By

Parul Garg

602262012

Under Supervision of

Dr. Sumit Vyas

Assistant Professor

ELECTRONICS AND COMMUNICATION ENGINEERING DEPARTMENT

THAPAR INSTITIUTE OF ENGINEERING AND TECHNOLOGY

(A DEEMED TO BEUNIVERSITY), PATIALA, PUNJAB

JULY 2024

INTEL INDIA PRIVATE LTD

No 23-56, SRR Campus, Outer Ring Road, Devarabeesanahalli, Varthur Hobli, Bellandur, Bengaluru, Karnataka, 560103., India

CERTIFICATE

Date: 19th July,2024

This is to certify that Parul Garg, a student of MTech (VLSI Design), Thapar Institute of Engineering & Technology, Patiala, pursuing her internship program for 12 months (June 2023 to June 2024) of duration at **INTEL INDIA PRIVATE LTD, Bangalore**. Her title of dissertation is "*UVM Environment Bring-Up for AXI4 Full Based DMA IP*".

Mr. Abhishek Tiwari
FPGA IP Software Engineering Manager
INTEL

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

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

To Whomsoever It May Concern

WWID: 12205348

Employee Name: Parul Garg

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

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

We wish you all the best for your future assignments.

Yours Sincerely

Sumana Dey Srivastava

Date: July 23, 2024

Place: **Bangalore**

DECLARATION

I, **Parul Garg** hereby declare that the work presented in this thesis entitled **"UVM Environment Bring-Up for AXI4 Full Based DMA IP"** is partial fulfilment of the requirement for the award of degree of **Master of Technology (VLSI Design)** submitted at **Electronics & Communication department,** Thapar Institute of Engineering & Technology (Deemed to be University), Patiala is an authentic record of work carried out under the supervision of **Dr. Sumit Vyas (Assistant Professor, ECED, Thapar Institute of Engineering & Technology)** from June 2023 to June 2024. The matter presented in this has not been submitted in part or full to any other university or institute for the award of any other degree.

Date:19th July,2024

Parul Garg
(602262012)

Mr. Abhishek Tiwari
FPGA IP Software Engineering Manager
INTEL

Date: 19th July,2024

Dr. Sumit Vyas
Assistant Professor
Department of Electronics And Communication Engineering
Thapar Institute of Engineering & Technology, Patiala

Date: 19th July,2024

ACKNOWLEDGEMENT

The Task could not be completed without acknowledging to those who guided and supported continuously to make our efforts successful. Taking this opportunity, I express my deepest gratitude and respect to my supervisor, **Dr. Sumit Vyas,** Assistant Professor, Department of Electronics & Communication Engineering, Thapar Institute of Engineering & Technology for his guidance and encouragement throughout this project. I express my deep gratitude and thanks to **Mr. Abhishek Tiwari** (FPGA IP Software Engineering Manager, INTEL), **Mr. Bhanu Vikas, Namani** (FPGA IP Software Development Engineer, INTEL) for their valuable advice and suggestions with unwavering support throughout the project. Many thanks to the members of IP Design Verification Team, for their valuable guidance during the project which helped me understand the technical aspects of the project. I would also like to thank **Dr. Kulbir Singh**, Head of Department, Electronics & Communication Engineering, Thapar Institute of Engineering & Technology for giving such an opportunity to do internship at INTEL India Private Ltd and provide all kinds of support throughout the project. Finally, I thank my family, friends and colleagues for their timely help and valuable suggestions.

ABSTRACT

This project report details the comprehensive verification process for the AXIMM Interface of a DMA IP, focusing on ensuring its functionality, correctness, and adherence to specifications. Direct Memory Access (DMA) is crucial in modern computer architecture, particularly when transferring data between a host machine and an FPGA accelerator. A modular DMA core on the FPGA alleviates the host CPU during transfers, conserves resources, and improves performance. It is essential to verify the functionality of a DMA IP before implementing it on an FPGA. The verification methodology employed includes simulation, formal verification, and emulation techniques to thoroughly validate the IP's design. Key areas covered are functional correctness, protocol compliance, performance analysis, and coverage metrics. The report discusses the challenges faced during verification and the strategies used to overcome them, ultimately ensuring the AXIMM's reliability and robustness for integration into the DMA IP. The team designed reusable sequences, scoreboards, and tests, with the overall testbench structure based on a base-type test. Different tests extend this base-type test and use type overriding with the UVM configuration database to employ various scoreboards and sequences as needed.

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 DMA

In applications that require data acquisition through computer interfaces, the efficient management of high-speed and voluminous data transfers is crucial. Relying solely on the microprocessor for the transfer of data between input/output (I/O) devices and memory is not an optimal use of its capabilities. In such scenarios, a Direct Memory Access (DMA) controller offers a more efficient solution, facilitating communication between peripherals while freeing the microprocessor to focus on other critical tasks [1]. DMA enables automatic data transfers between the system memory and peripherals without the need for intervention by the central processing unit (CPU). When a significant amount of data needs to be moved from the data memory, the CPU initiates the DMA Controller, granting it access to the bus. The DMA Controller then oversees the data transfer process. Upon completion, it notifies the CPU through an interrupt, signaling that the data transfer task has been completed, and the CPU can resume control over the bus access.

As illustrated in Figure 1, the DMA module is integrated into an FPGA, acting as an accelerator for the host computer. The FPGA's user logic is designed to process data retrieved from the host's main memory, with the processed results being sent back to the host. The DMA module serves as a bridge between the user logic and a bus module, such as a PCIe module, facilitating communication. The FPGA is linked to the host computer through physical bus connections, with the necessary drivers and applications stored in the host's main memory. Additionally, memory specifically allocated for DMA operations between the host computer and the FPGA is set aside in advance on the host side.

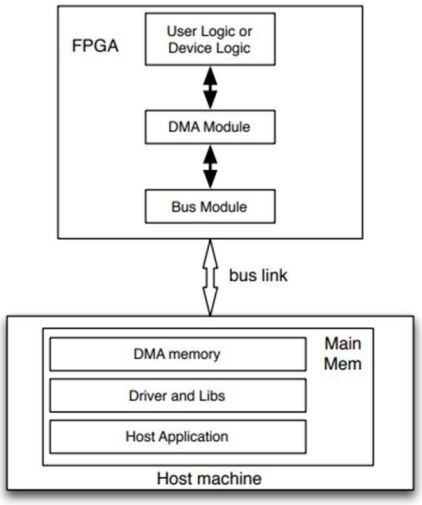


Figure 1.1: DMA Module with FPGA

1.1.1 Working of DMA

DMA controllers operate under the CPU's supervision, but data transfers through DMA occur independently of the CPU. They provide an interface between I/O devices and the bus, allowing multiple external devices to connect via DMA.

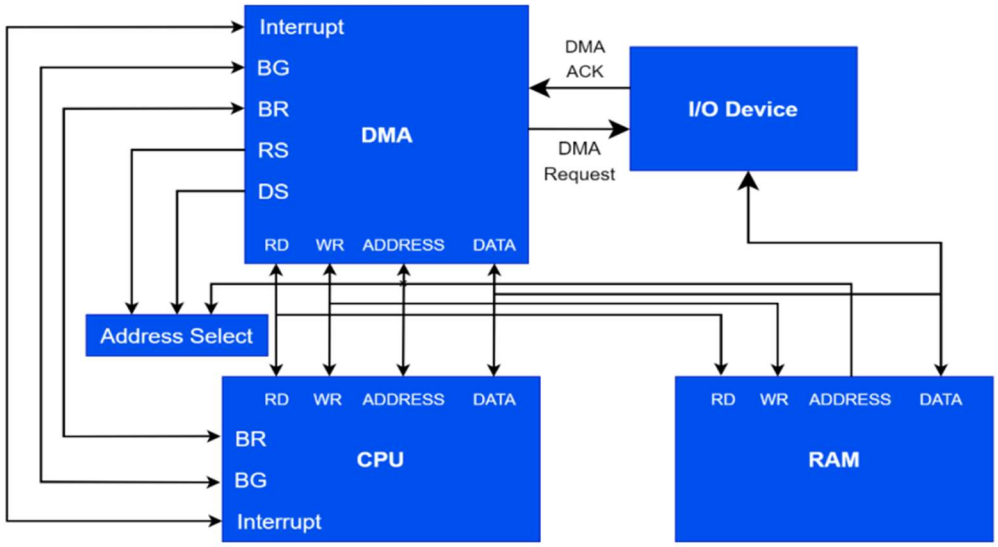


Figure 1.1: Working of DMA Controller

The word count, address, and control registers are additional three registers found in DMA controllers. We use the address register to locate the desired location in the memory. The word

count register keeps track of the words that must be transferred from the input data. Finally, the control register specifies the transmission mode.

The data flow between memory and an auxiliary device is controlled by a DMA controller. The controller is plugged into the motherboard of the computer system and connected to the devices through connectors that require data transfer. The computer system temporarily halts execution once the data flow has been established, and the DMA controller takes over. Additionally, when the CPU transfers data via a data bus, we initialize DMA.

For successful data transfer, DMA must share the bus (data and address) with the CPU. Let's consider a scenario where data is being transferred from an I/O device to memory. In this case, the CPU halts the current program and increments the program counter by one. The CPU then sends a signal to the DMA controller via the address bus. If the DMA controller accepts this signal, it generates a bus request signal and sends a request to the CPU to control the bus. If the CPU grants this request, the DMA controller becomes the bus master. The CPU then provides the memory addresses, the direction of data transfer, and the number of data blocks. Once the DMA controller is set up to handle the data transfer, the CPU resumes its execution.

DMA controllers operate in two modes: single-channel and multi-channel. In single-channel mode, data is transferred between memory and one device, and vice versa. In multi-channel mode, data is transferred simultaneously from memory to multiple devices or from multiple devices to memory.

1.2 MCDMA

The PCIe Multi-Channel DMA enable Intel PCIe users to quickly implement DMA application that efficiently transfers data between the host and multiple local clients. This sub-system can support up-to 2048 physical queues in the host sharing the same FPGA DMA with virtual queues implementation [2].

The MCDMA engine utilizes a software-managed DMA queue for data transfer between the host and the local FPGA.

- Each queue contains software descriptors.
- The driver/software writes these software descriptors to the queues
- Hardware first reads the queue descriptors and execute them sequentially.
- Separate queues are used for read and write DMA operations for each channel.
- Hardware can support 2048 H2D and 2048 D2H queues

• Descriptors:

o Linked list of 4KB pages. Last entry in page is a link to the next 4KB page consisting of descriptors.

o Each descriptor contains a bit that says if the contents of the descriptor are a descriptor or a link.

o Head/Tail index/pointers still maintained, assuming virtually contiguous list of descriptors.

1.2.1 Host-to-Device Data Movement

The Host-to-Device Data Mover (H2DDM) module facilitates the transfer of data from host memory to Device Side memory via the PCIe Hard IP. It operates in two distinct modes: queue descriptor fetching and host-to-device data payload transfer.

During descriptor fetching, completion data destinations are temporarily stored in FIFOs before being sent for actual data transfer by either the Host-to-Device Data Mover or the Device-to-Host Data Mover [2].

For data payload transfer, the H2DDM generates MemRd requests based on descriptor parameters such as source address, data size, and MRRS value:

- The first MemRd request aligns with the MRRS address boundary.
- Subsequent MemRd requests utilize the full MRRS size.
- The final MemRd request covers any remaining partial MRRS space. Received completions are reordered to ensure sequential delivery of read data to user logic.

Upon completion of descriptor fetch, indicating receipt and forwarding of all read data to the Device Side interface, the H2DDM performs additional tasks:

- It schedules interrupts for completed queues, if enabled.
- It also schedules writebacks for completed queues, if enabled.

1.2.2 Device-to-Host Data Mover

The Device-to-Host Data Mover (D2HDM) transfer of data takes place from device memory to host memory. It retrieves data from user logic via the Sink interface and initiates Mem Write requests to move this data to the host, guided by descriptor details such as destination address, data size, and transmission value to store data in the host's receive buffer.

In Memory Mapped mode, the D2HDM conducts a sequence of Memory Mapped reads through the master port, utilizing PCIe addresses and DMA transfer sizes. Memory Mapped reads are managed according to the following rules:

- The initial Memory Mapped read aligns with a 64-byte address boundary, with multiple bursts read initially.
- Subsequent Memory Mapped reads occur under the following conditions:
  - The Memory Mapped address is 64-byte aligned.
  - The total payload count specified in the descriptor aligns with 64-byte increments and does not exceed the maximum permissible.
- The final Memory Mapped read handles any remaining data size.

Table 1.1: Acronyms Definition

Term	Definition
API	Application Programming Interface
D2H	Device-to-Host
D2HDM	Device-to-Host Data Mover
DMA	Direct Memory Access
EOF	Streaming of packet for End Of File
MemWr	Memory Write
MemRd	Memory Read
PCIe*	Peripheral Component Interconnect Express (PCI Express*)
File (or Packet)	In the context of streaming, a set of descriptors identifies the beginning (SOF) and end (EOF) of a file or packet. Within the Avalon-ST user interface, sof/eof markers indicate the delineation of individual files or packets.
GCSR	General Control and Status Register
H2DDM	Host-to-Device Data Mover
H2D	Host-to-Device
HIP	Hard IP

HIDX	Queue Head Index (pointer)
SOF	Streaming of packet for Start Of File
IP	Intellectual Property
MCDMA	Multi Channel Direct Memory Access
MRRS	Maximum Read Request Size
MSI-X	Message Signaled Interrupt - Extended

1.3 Literature Review

Deepa Kaith, Dr. Janak Kumar B. Patel, Mr. Neeraj Gupta [5] writers discuss how technology progresses, an increasing amount of logic is being integrated onto a single silicon chip, making the verification process more complex than ever before. Verification now consumes over 70% of the entire design cycle. To accelerate the time-to-market, it's crucial to have a verification environment that is both reusable and capable of identifying all functional discrepancies to prevent the need for redesigns. The Universal Verification Methodology (UVM) was developed to meet these objectives. UVM is designed to be well-organized and reusable with minimal adjustments, without disrupting the operation of the device being tested (DUT), thereby enhancing the efficiency of the verification process. Supported by all leading simulation software providers, a feature does not present in previous methodologies, UVM offers a standardized, unified approach that is compatible across all simulation tools. This document highlights the benefits of using UVM over System Verilog, introduces essential UVM terminologies, and outlines the creation of a straightforward functional verification environment utilizing UVM.

Anjali and J. P. Anita, [6]

This paper presents a comprehensive study on the development of a testbench architecture for an AXI-based Direct Memory Access (DMA) memory system, utilizing the Universal Verification Methodology (UVM) harness technique. As the variations of system-on-chip (SoC) designs escalates, the requirement for efficient and reliable verification methodologies becomes paramount. The Advanced eXtensible Interface (AXI) protocol, widely adopted for high-speed data transfer between components within SoCs, necessitates a robust verification strategy to ensure optimal performance and reliability of DMA memory systems. This research introduces a novel testbench architecture that leverages the UVM harness technique to provide a scalable, reusable, and efficient verification environment specifically tailored for AXI-based DMA systems.

V. Melikyan, S. Harutyunyan, A. Kirakosyan and T. Kaplanyan, [7]

The paper outlines the challenges associated with AXI protocol verification, including the need to accurately model the protocol's various features such as multiple transaction types, burst transfers, and different levels of signaling complexity. It then presents the architecture of the UVM VIP, detailing its modular design, which encompasses configurable agents, sequencers, and monitors, facilitating both directed and random testing strategies to thoroughly exercise the AXI interface under test.

H. Sangani and U. Mehta, [8]

This document delves into the Modern System-On-Chip (SoC) designs are increasingly intricate, incorporating numerous Intellectual Properties (IPs) within a single SoC, which communicate via a variety of bus protocols. Consequently, verification consumes about 70% of the design cycle, making a reusable verification environment for these widely utilized protocols crucial. This document explores the verification of the AXI4-Lite protocol through a UVM-based testbench architecture. To thoroughly verify the AXI protocol channels, data is input into a 4-bit shift register and subsequently retrieved. The UVM testbench functions as a master device, transmitting all necessary control information, data, and addresses to the register via the AXI interface. To gauge the success of the verification objectives, cover points are established, and both functional and code coverage reports are examined. The Synopsys VCS tool facilitates the simulation process.

F. Plasencia-Balabarca, E. Mitacc-Meza, M. Raffo-Jara and C. Silva-Cárdenas, [9]

This paper introduces a flexible, Universal Verification Methodology (UVM)-based verification framework designed for comprehensive testing of an Advanced Encryption Standard (AES) encryption module across multiple bus interfaces, including Avalon, Advanced High-performance Bus (AHB), Advanced eXtensible Interface (AXI), and Wishbone. As the demand for secure data transmission escalates in complexity, the need for an adaptable and reusable verification environment becomes paramount, especially in systems that incorporate various bus protocols to facilitate communication between integrated circuits (ICs). The proposed verification framework leverages the UVM's robustness and scalability to offer a unified solution capable of interfacing with different bus protocols without the need for significant modifications. This approach not only streamlines the verification process but also significantly

reduces the time and resources required to validate the AES encryption module's functionality and compatibility with each bus interface.

AMBA AXI Protocol ARM Document, [10]

ARM introduced the AMBA AXI protocol as part of its Advanced Microcontroller Bus Architecture family to address the increasing demands for higher bandwidth and lower latency in SoC designs (ARM Ltd.). The protocol was designed to support high-speed, high-bandwidth data transfers with features like out-of-order transaction completion and separate channels for different types of data transfers. The protocol specifies five channels (read address, write address, read data, write data, and response) that operate independently, allowing for simultaneous data transactions and increased throughput (ARM Ltd.). This design supports features such as burst-based transfers, split transactions, and exclusive access, which are crucial for efficient memory access and management. Studies have shown that the AXI protocol's design significantly improves data throughput and system performance compared to its predecessors (AMBA AHB and APB). For instance, Kumar and Kumar (2015) demonstrated through simulations that AXI provides better bandwidth utilization and lower latency in data transfers. With the increasing concern over data security, incorporating security features into the AXI protocol could be a potential area of research. This could involve developing mechanisms for secure data transfers and protecting against unauthorized access in SoC designs.

R. Madan, N. Kumar and S. Deb, [11]

various self-checking strategies that can be used in conjunction with UVM were discussed. This enables verification engineers to select the appropriate method for their verification needs from among the available checking mechanisms. Their first solution is based on a checker that uses reference modules. The second way is a conventional implementation of the UVM checker mechanism, which allows vertical reusability but necessitates the creation of an expect function in the scoreboard, limiting its scalability. The third checker mechanism implementation does not employ the expect function and allows numerous reference models of various types to be used simultaneously to verify a single DUV or parallel processes of the same DUV. This minimizes total debug time and engineer effort in verifying complicated systems, while also improving the verification environment's checking capability.

S. S. Math, R. B. Manjula, S. S. Manvi and P. Kaunds, [12]

This paper introduces the AMBA (Advanced Microcontroller Bus Architecture) protocol suite, which incorporates metric-driven verification methods to ensure protocol adherence. This approach supports comprehensive testing of interface IP (Intellectual Property) blocks and SoC (System-on-Chip) architectures. The AMBA AXI4 protocol, an advancement from the AXI3 version, includes enhancements such as support for burst lengths up to 256 beats, updated criteria for write responses, elimination of locked transactions, and guidance on component interoperability. The AMBA AXI4 framework facilitates communication among 16 masters and 16 slaves.The document focuses on a project that implements data transactions on an SoC bus using the AMBA AXI4 protocol, described in Verilog Hardware Description Language (HDL). It discusses simulation results using the Verilog Compiler Simulator (VCS), operating at a frequency of 100MHz, covering both read and write operations of data and addresses. The project includes two test scenarios to perform multiple read and write actions, with a single read operation requiring 160ns and a single write operation consuming 565ns.

M. P. Deepu and R. Dhanabal, [13]

This paper explores the validation of transactions within the Advanced eXtensible Interface (AXI) protocol utilizing System Verilog, a leading hardware description and verification language. The AXI protocol, a key component of the ARM Advanced Microcontroller Bus Architecture (AMBA) specification, is pivotal in facilitating high-speed, efficient communication between components in complex System-on-Chip (SoC) designs. Given the critical role of AXI in ensuring the integrity and performance of SoC architectures, rigorous validation of AXI transactions is essential. The study presents a comprehensive validation framework developed in System Verilog, aimed at meticulously verifying the functionality and performance of AXI protocol transactions. This framework leverages System Verilog's advanced verification features, such as classes, randomization, assertions, and functional coverage, to create a robust environment for AXI transaction validation. The methodology encompasses a wide range of test scenarios, from basic single transactions to complex burst transfers and error conditions, ensuring thorough coverage of the AXI protocol's specifications.

G. Mahesh and S. M. Sakthivel, [14]

This paper explores the validation of transactions within the Advanced eXtensible Interface (AXI) protocol utilizing SystemVerilog, a leading hardware description and verification

language. The AXI protocol, a key component of the ARM Advanced Microcontroller Bus Architecture (AMBA) specification, is pivotal in facilitating high-speed, efficient communication between components in complex System-on-Chip (SoC) designs. Given the critical role of AXI in ensuring the integrity and performance of SoC architectures, rigorous validation of AXI transactions is essential. The study presents a comprehensive validation framework developed in SystemVerilog, aimed at meticulously verifying the functionality and performance of AXI protocol transactions. This framework leverages SystemVerilog's advanced verification features, such as classes, randomization, assertions, and functional coverage, to create a robust environment for AXI transaction validation. The methodology encompasses a wide range of test scenarios, from basic single transactions to complex burst transfers and error conditions, ensuring thorough coverage of the AXI protocol's specifications.

Gayathri M, R. Sebastian, S. R. Mary and A. Thomas, [15]

Authors discuss how rapid technological advancements have resulted in the creation of increasingly complex digital systems, making them more susceptible to errors. Traditional verification methods are inadequate in providing a flexible verification environment and struggle to balance market demands with the urgency of product launches. The Universal Verification Methodology (UVM), based on System Verilog (SV), addresses these challenges by enabling the development of a robust and reusable verification framework.This study introduces an effective SV-UVM framework designed specifically for verifying the Serial Gigabit Media Independent Interface (SGMII) IP core. The SGMII operates as a single-lane interface with a data rate of 1.25 Gbps, facilitating communication between the Ethernet Media Access Control (MAC) and the Physical (PHY) layer. The core integrates an AMBA AXI (Advanced eXtensible Interface) master interface to access the IP's register space.To verify the SGMII core, we utilized a UVM Verification Component (UVC) of the AXI to WB (Wishbone) bridge, ensuring compatibility with Wishbone by configuring various registers within the core. All simulations were performed using NCsim, waveform analysis was conducted using Simvision, and coverage analysis was carried out through the Incisive Metrics Center (IMC).

H. -Y. Yang, [16]

The paper discusses how the designs grow increasingly complex, the task of functional verification has become more challenging than ever before. This difficulty manifests in two primary ways: extended verification durations and the increased complexity of identifying all functional errors. Research[1] has indicated that functional errors are the leading cause of design re-spins. The effectiveness of the verification process is thus becoming critically important. Re-spins can be exceedingly costly, not only due to the advancing costs of manufacturing processes but also because of the delays they introduce to the product's time to market, which can incur even greater expenses than the re-spin itself. To address these challenges, the industry has recently adopted the Universal Verification Methodology (UVM)[2] as a solution. However, even with the standardization provided by UVM for testbench design, it is essential for a simulation environment to be meticulously crafted. This ensures that UVM's benefits are fully leveraged and that the management and execution of a large volume of simulations/regressions are conducted efficiently.

2. Verification

Verification encompasses various activities and techniques to validate that the system meets its intended requirements and behaves as expected. The specific objectives can vary depending on the context and goals of the verification effort. Here are the objectives behind verification and formal property verification:

- Error Detection: One of the main objectives of verification is to detect and uncover errors, design flaws, or unexpected behaviors in the system. Through a methodical process of analysis and testing, verification seeks to uncover and resolve possible problems that might cause system failures or malfunctions.
- Requirement Satisfaction: Verification ensures that the system satisfies its specified requirements. It involves verifying that the system's behavior aligns with the intended functionality, performance, safety, or other critical properties defined in the system's requirements documentation.
- Compliance with Standards and Regulations: Verification aims to demonstrate compliance with industry standards, regulations, or specific safety guidelines. It ensures that the system adheres to the necessary standards and can be deployed in safety-critical or regulated domains.
- Risk Mitigation: Verification helps in identifying and mitigating risks associated with system failures or errors. By conducting thorough verification activities, potential risks can be assessed, and necessary measures can be taken to minimize their impact or likelihood of occurrence.
- Design Refinement: Verification provides feedback for design refinement and improvement. By analyzing and validating the system's behavior, verification identifies areas where the design can be enhanced, optimized, or corrected to improve its reliability, efficiency, or performance.
- Confidence and Trust: Verification, including formal property verification, aims to provide confidence and trust in the system's correctness and reliability. By providing rigorous analysis, testing, and evidence, verification ensures that the system operates as intended, which fosters trust among stakeholders, users, and regulatory bodies.
- Cost and Time Efficiency: Verification aims to optimize the use of resources, time, and effort involved in the development process. By detecting errors early and ensuring correct behavior, verification reduces the cost and time associated with rework, debugging, and post-deployment issues.
- Documentation and Compliance Traceability: Verification generates documentation and

evidence of the verification activities performed. This documentation helps in compliance traceability, audits, and future maintenance and enhancement of the system.

2.1 Types of Verification

Verification is a fundamental aspect of the development process for systems, designs, and products, ensuring that they meet specified requirements and perform as intended. It serves as a critical quality assurance measure, providing confidence in the correctness, completeness, and reliability of the developed solution. This report aims to provide an introduction to verification, its significance, and its application across various domains.

The primary objective of verification is to ensure that the system or product functions as expected and meets the intended purpose. It involves the rigorous examination of design documents, specifications, and implementation artifacts to identify any discrepancies or deviations. By conducting thorough reviews, inspections, and tests, verification aims to detect design flaws, programming errors, and other issues that may impact performance, safety, security, or reliability.

Effective verification requires a combination of techniques and methods tailored to the specific context and objectives. These may include inspections, formal methods, simulation, testing, and static analysis. Each technique offers distinct advantages and limitations, and the selection of the appropriate approach depends on factors such as system complexity, criticality, and available resources.

There are several types of verification that are commonly used in the development process. These types of verification serve different purposes and focus on specific aspects of the system or product. Here are some key types of verification:

1. **Design Verification**: This type of verification focuses on assessing whether the system or product design meets the specified requirements. It involves reviewing design documents, models, and specifications to ensure that they accurately capture the desired functionality, behavior, and performance.

2. **Code Verification**: Code verification involves checking the implementation of the system or product against the design and coding standards. It aims to identify coding errors, syntax issues, adherence to coding guidelines, and potential vulnerabilities.

3. **Functional Verification**: Functional verification aims to confirm that the system or product performs its intended functions correctly. It involves testing individual

components, subsystems, or the entire system to ensure that it behaves according to the defined functional requirements.

4. **Performance Verification**: Performance verification focuses on evaluating the system or product's performance characteristics, such as response time, throughput, scalability, and resource utilization. It aims to validate that the system meets the specified performance requirements.

5. **Security Verification**: Security verification assesses the system or product's resilience against potential security threats and vulnerabilities. It involves identifying security weaknesses, conducting penetration testing, and ensuring that appropriate security measures are implemented.

6. **Compliance Verification**: Compliance verification ensures that the system or product adheres to relevant standards, regulations, and industry-specific requirements. It involves assessing compliance with legal, safety, quality, and industry-specific guidelines and specifications.

7. **Interface Verification**: Interface verification verifies the interactions and compatibility between different components, subsystems, or external interfaces. It ensures that data exchange, communication protocols, and interactions between system elements function correctly.

8. **Configuration Verification**: Configuration verification focuses on verifying that the system or product is correctly configured and deployed. It involves validating the configuration settings, dependencies, and compatibility with the intended environment.

9. **Documentation Verification**: Documentation verification ensures that the system or product documentation is accurate, complete, and up-to-date. It involves reviewing user manuals, technical documentation, and system specifications to confirm their alignment with the implemented system.

10. **Formal Verification**: Formal verification is the act of proving or disproving the correctness of a system with respect to a certain formal specification or property. It checks

different possible states covered by DUT which makes bug hunting safe. The user specifies illegal scenarios, tool drives all possible combinations.

These different types of verification are often performed in combination to provide comprehensive validation and assurance of the system or product's correctness, quality, and compliance with requirements. The specific types and extent of verification activities vary depending on the nature of the system, industry standards, and project requirements. By understanding the significance of verification and the methods employed to assess system correctness, this report aims to emphasize the crucial role that verification plays in delivering high-quality solutions. Through effective verification practices, organizations can mitigate risks, improve system reliability, and meet customer expectations.

Overall, verification serves as a critical quality assurance measure, ensuring that the developed system or product adheres to the specified requirements and performs as expected. It is essential for reducing risks, improving system dependability, and providing high-quality solutions that satisfy users' and stakeholders' needs.

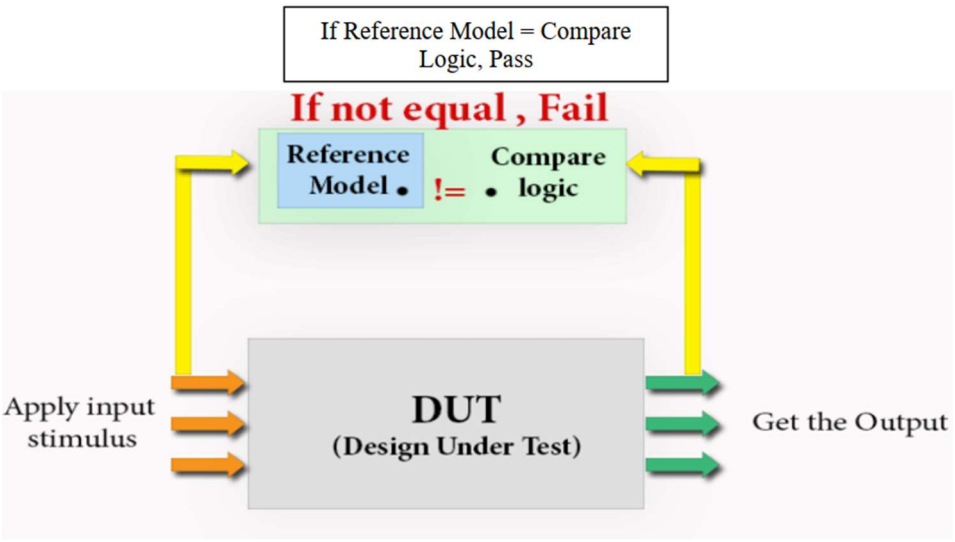


Figure 2.1: Block Diagram for Verification

2.2 Verification Cycle

The Verification Cycle is a critical phase in the development process of hardware and software systems, aimed at ensuring that the design meets specified requirements and functions correctly. This cycle involves a series of steps that systematically test and validate the system's components and overall performance. Here's a breakdown of the Verification Cycle, highlighting its key stages and objectives:

1. Planning and Preparation

- **Objective Setting**: Define what needs to be verified, including the system's specifications and performance criteria.
- **Test Plan Creation**: Develop a comprehensive plan that outlines the verification strategy, including the methods and tools to be used, the scope of testing, and the criteria for success.

2. Development of Verification Environment

- **Environment Setup**: Build or configure the verification environment tailored to the system under test (SUT). This may involve hardware, software, or a combination of both, depending on the nature of the system.
- **Testbench Development**: Create a testbench, which is a configuration used to apply test scenarios to the SUT. In hardware verification, this often involves using languages like SystemVerilog or VHDL.

3. Implementation of Test Cases

- **Test Case Design**: Design test cases based on the objectives set in the planning phase. Each test case should aim to verify a specific aspect of the system's functionality or performance.
- **Test Execution**: Run the test cases against the SUT. This can be done manually or automatically, depending on the complexity of the system and the verification environment.

4. Analysis and Debugging

- **Results Analysis**: Analyze the outcomes of the test cases to identify any discrepancies between the expected and actual behavior of the system.
- **Debugging**: Investigate and fix any issues uncovered during testing. This step may involve revising the design, updating the test cases, or both.

5. Coverage Analysis and Closure

- **Coverage Measurement**: Assess the extent to which the test cases have exercised the system's functionalities. Coverage metrics can include code coverage, functional coverage, and more.
- **Closure and Reporting**: Once the coverage goals are met and the system is deemed to meet its specifications, the verification cycle can be concluded. A final report is prepared, summarizing the verification activities, findings, and any recommendations for future development.

6. Regression Testing

- **Continuous Verification**: As the system undergoes changes or updates, regression testing is performed to ensure that previously verified functionalities remain intact. This step may cycle back through the previous phases as necessary.

The Verification Cycle is iterative, often requiring multiple rounds of testing and debugging to address all potential issues. Its thoroughness is crucial for minimizing risks and ensuring the reliability and performance of the final system.

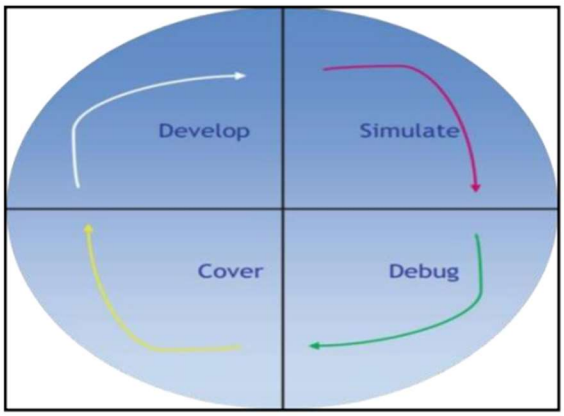


Figure 2.2: Design Verification Loop

2.3 Verification Challenges

Verification challenges arise from the complexity and scale of modern hardware and software systems. As these systems grow in sophistication, ensuring their correctness and performance according to specifications becomes increasingly difficult. Below are key challenges encountered during the verification process:

1. Increasing System Complexity

- **Complexity**: Modern systems, especially in the semiconductor and software industries, are incredibly complex, integrating millions of lines of code or transistors. This complexity makes it challenging to understand all possible states and interactions within the system.
- **Interconnectivity**: Systems often comprise numerous interconnected components and modules, each with its own functionality. Verifying the correct interaction between these components adds another layer of complexity.

2. Time-to-Market Pressures

- **Deadlines**: The competitive nature of the technology sector puts pressure on companies to release new products quickly. This urgency can limit the time available for thorough verification, potentially leading to undiscovered errors.
- **Resource Constraints**: Limited resources, including computational power and human expertise, can constrain the depth and breadth of verification efforts, impacting the quality of the final product.

3. Evolving Standards and Technologies

- **New Standards**: As technology evolves, new standards and protocols emerge, requiring verification teams to continuously update their knowledge and tools. It can be challenging to keep up with the challenges.
- **Emerging Technologies**: Innovations such as artificial intelligence, quantum computing, and new semiconductor materials introduce novel verification challenges, for which established methodologies may not be fully equipped.

4. Coverage and Completeness

- **Coverage Metrics**: Determining appropriate coverage metrics and achieving sufficient coverage are significant challenges. It's difficult to ensure that all relevant aspects of the system have been thoroughly tested.
- **Undiscovered Bugs**: Even with high coverage, critical bugs can remain undiscovered due to gaps in the verification process or the inherent limitations of the verification tools and methodologies.

5. Scalability of Verification Efforts

- **Scalability**: As systems scale, the verification effort does not scale linearly but can grow exponentially. Managing and scaling verification resources effectively to cover all necessary aspects of a large system is a daunting task.
- **Tool Limitations**: Verification tools may not scale well with increasing system size, leading to performance bottlenecks and inefficiencies in the verification process.

6. Integration and System-Level Verification

- **Integration Issues**: Verifying individual components does not guarantee that the system as a whole will function correctly. System-level integration introduces new variables and interactions that can be challenging to model and verify.
- **Environment Simulation**: Accurately simulating the operational environment of the system, including interactions with users and other systems, is complex but crucial for uncovering real-world issues.

3. AXI4 (AXI-MM)

3.1 Overview

The AMBA (Advanced Microcontroller Bus Architecture) represents an open standard for the communication and management of functional blocks within a System on Chip (SoC), offering various on-chip communication protocols such as CHI (Coherence Hub Interface), AXI (Advanced eXtensible Interface), ACE (Advanced Coherency Extension), AHB (Advanced High-Performance Bus), and APB (Advanced Peripheral Bus), all developed by ARM (Advanced RISC Machine). The flexibility of AMBA protocols facilitates the reuse of IP across different SoC designs, catering to varying requirements in terms of area, power, and performance.

Being widely recognized open standards, AMBA protocols ensure compatibility among IPs from various suppliers for SoCs, promoting easy integration and IP reuse. This compatibility is crucial for accelerating the time to market. Specifically, the AMBA AXI protocol is crafted to support high-frequency, high-bandwidth interfaces suitable for a broad range of applications, including embedded systems, automotive technology, and cellphones, without necessitating complex bridges for different peripheral devices. The AXI protocol introduces several enhancements over its predecessors, ensuring backward compatibility and complementing CHI.

The AMBA 4.0 specification outlines three variants of AXI4-Interfaces:

- **AXI4 (Full AXI4)**: Tailored for high-performance applications requiring memory-mapped access.
- **AXI4-Lite**: Designed for straightforward, low-throughput memory-mapped interactions, such as those involving control and status registers.
- **AXI4-Stream**: Suited for high-speed data streaming.

The AXI protocol is characterized by several notable features:

- Distinct phases for address/control and data.
- Capability to handle unaligned data transfers through byte strobes.
- Transactions based on bursts, with only the start address being issued.
- Dedicated read and write data channels, facilitating cost-effective Direct Memory Access (DMA).
- Provision for issuing multiple outstanding addresses.

- Completion of transactions can occur out of order.
- Registers can be easily added to achieve timing closure. Additionally, the AXI protocol encompasses optional extensions for low-power operation signaling.

The AXI protocol facilitates:

- The advance issuance of address information prior to the actual data transfer.
- The handling of multiple outstanding transactions.
- The allowance for transactions to complete in an out-of-order fashion.

AXI Read and Write Channels Overview - Within the AXI protocol, five distinct channels are specified:

- For **Read operations**, two channels are utilized:
  - Read address channel.
  - Read data channel.
- For **Write operations**, three channels are designated:
  - Write address channel.
  - Write data channel.
  - Write response channel.

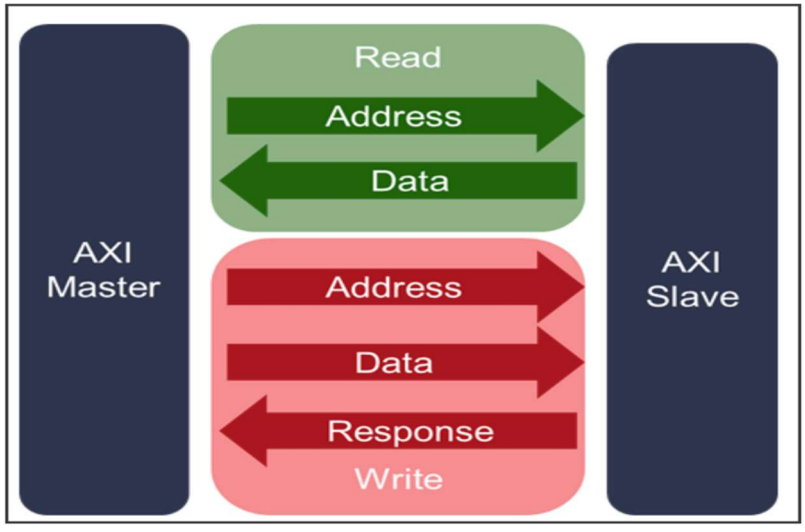


Figure 3.1: AXI Read and Write Channels

3.2 Channel architecture of reads

Read address channel carries all required address and control information for a transaction.

- Read address ID (ARID)
- Burst information: burst type, burst length, and burst size.
- Atomic characteristic of transfer
- Protection type for transaction
- Cacheable characteristic of transfer
- Handshake signals: ARVALID and ARREADY

Read data channel carries data and response information for a transaction.

- Read ID tag (*RID must match ARID)
- Read data bus (range from 8,16,32,64,128,256,512 or 1024 bits wide)
- Read Response signals indicate status of read transfer.
- Read Last indicate last transfer in burst.

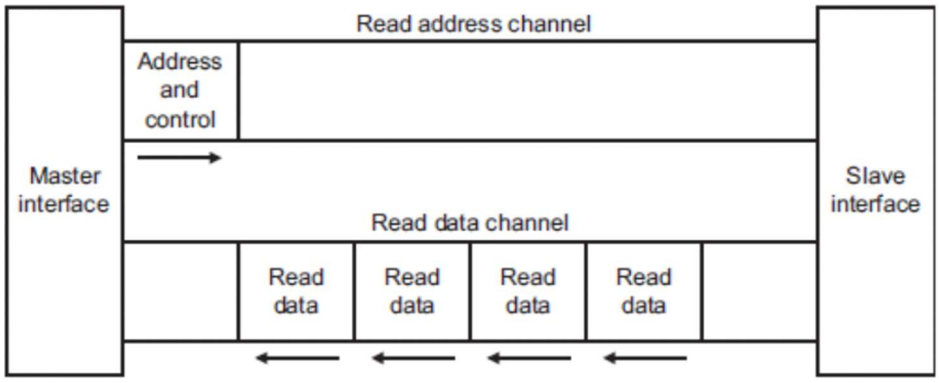


Figure 3.2: Channel architecture of reads

3.3 Channel architecture of writes

Write address channel carries all required address and control information.

- Write address ID (AWID)
- Burst information: burst type, burst length, and burst size.
- Atomic characteristic of transfer
- Protection type for transaction
- Cacheable characteristic of transfer
- Handshake signals: AWVALID and AWREADY

Write data channel carries data and response information.

- Write ID tag (*WID must match AWID)
- Write data bus (range from 8,16,32,64,128,256,512 or 1024 bits wide)
- Write strobes indicate which byte lanes to updates in memory.
- Write Last indicate last transfer in burst.
- Handshake signals: WVALID and WREADY

Write Response channel carries response information from slave *once per each burst.

- Response ID, BID (*BID must match AWID)
- Write Response signals indicate status of write transaction.
- Handshake signals: BVALID and BREADY

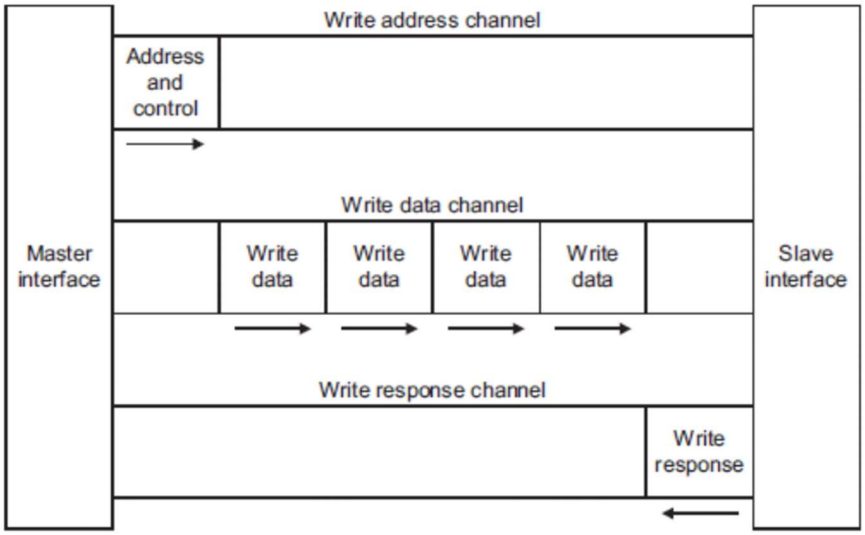


Figure 3.3: Channel architecture of writes

3.4 Interface Signal Definitions

Table 3.1: Write Address Signals

Signal	Source	Definition of Signal
AWID[x:0]	Manager / Master	Provides ID for each transaction
AWADDR[31:0]	Manager / Master	Address for write request
AWLEN[7:0]	Manager / Master	Number of transfers support in each transaction
AWSIZE[2:0]	Manager / Master	No. of bytes to be transfer in each beat
AWBURST[1:0]	Manager / Master	Indicates type of burst to be performed

AWLOCK	Manager / Master	Indicates the atomic characteristics of the transaction
AWCACHE[3:0]	Manager / Master	This signal indicates the system performance
AWPROT[2:0]	Manager / Master	Provides system level security and privileged access to each transaction.
AWQoS[3:0]	Manager / Master	Use to priorities the transactions
AWREGION[3:0]	Manager / Master	Region identifier
AWVALID	Manager / Master	Use to validate the associated signal in order to pass valid information
AWREADY	Subordinate/slave	Indicates weather slave is ready for the transactions.

Table 3.2: Write Data Signals

Signal	**Source**	**Definition of Signal**
AWDATA[x:0]	Manager / Master	Write data signal
AWSTRB[x:0]	Manager / Master	Used to send valid bytes for each transfer
AWLAST	Manager / Master	Indicates last transfer for the transaction
AWVALID	Manager / Master	Used to validate the associated signal in order to pass valid information.
AWREADY	Subordinate/ slave	Indicates weather slave is ready for the transactions.

Table 3.3: Write Response Signals

Signal	**Source**	**Definition of Signal**
BID[x:0]	Subordinate/slave	Provides ID for each write transaction
BRSEP[1:0]	Subordinate/slave	Write Response signal
BVALID	Subordinate/slave	Use to validate the associated signal in order to pass valid information.
BREADY	Manager / Master	Indicates weather master is ready for the transactions.

Table 3.4: Read Address Signals

Signal	Source	Definition of Signal
ARID[x:0]	Manager / Master	Provides ID for each transaction
ARADDR[31:0]	Manager / Master	Address for write request
ARLEN[7:0]	Manager / Master	No.of transfers support in each transaction
ARSIZE[2:0]	Manager / Master	No. of bytes to be transfer in each beat
ARBURST[1:0]	Manager / Master	Indicates type of burst to be performed
ARLOCK	Manager / Master	Indicates the atomic characteristics of the transaction
ARCACHE[3:0]	Manager / Master	This signal indicates the system performance
ARPROT[2:0]	Manager / Master	Provides system level security and privileged access to each transaction.
ARQoS[3:0]	Manager / Master	Use to prioritise the transactions
ARREGION[3:0]	Manager / Master	Region identifier
ARVALID	Manager / Master	Use to validate theassociated signal inorder to pass valid information.
ARREADY	Subordinate/slave	Indicates weather slave is ready for thetransactions.

Table 3.5: Read data Signal

Signal	Source	Definition of Signal
RID[x:0]	Subordinate/ slave	Provides ID for each read transaction
RDATA[x:0]	Subordinate/ slave	Read data signal
RESP[1:0]	Subordinate/ slave	Read response signal
RLAST	Subordinate/ slave	Indicates last transfer for the transaction
RVALID	Subordinate/ slave	Use to validate the associated signal in order to pass valid information.
READY	Manager / Master	Indicates weather master is ready for the

3.5 Channel Signal Descriptions

Awid : Master - Write Address ID

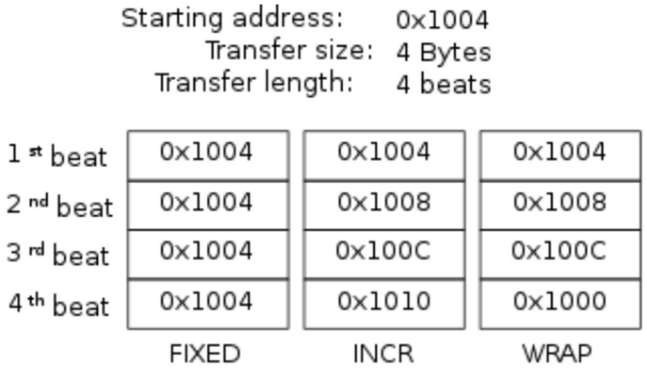
1. Each transaction has its own id.

Awaddr : Master - Write Address

1. Write address for the first transfer.
2. Drives Address of the first-byte in the tx to the slave.
3. The slave must calculate the subsequent transfers in the burst.
4. Burst must not cross a 4KB address boundary.

Awlength : Master - Burst Length

1. Specifies the precise count of transfers within a burst.
2. Burst_length = AWLEN + 1
3. AWLEN[7:0], for write transfers for INCR burst(new feature for AXI4)
4. AWLEN[3:0], for write transfers for FIXED, WRAP burst
5. AWLEN for different burst types :
    a. FIXED : 1 to 16
    b. INCR : 1 to 256 transfers
    c. WRAP : 2,4,8 or 16
6. Early termination of burst is not supported.
    a. Hence, master can reduce data transfer in write burst by de-asserting pstobe.



Burst types

Awsize : **Master - Size of each transfer:**

1. Gives the size of each transfer in the burst.

AxSIZE[2:0]	Bytes in transfer 0b000 1
0b001	2
0b010	4
0b011	8
0b100	16

0b101	32
0b110	64
0b111	128

2. No of address bytes of transfer in the burst will be calculated as two powers of AWSIZE of Address Channel bus.
3. If the AXI bus is wider than the burst size, the AXI interface must determine from the transfer address which byte lanes of the data bus to use for each transfer. See Data read and write structure.
4. The size of any transfer must not exceed the data bus width of either agent in the transaction.

Awburst: MASTER - Type of transfer

fixed :

1. The start address is the same for all transfers in the burst.
2. The byte lanes still may differ based on the assertion of WSTRB
3. This burst type is used for repeated accesses to the same location such as when loading or emptying a FIFO.

Incr :

1. The address will be incremented for the next beat/transfer from the start address.
2. The increment will be based on the size of the transfer.
3. This burst type is used for access to normal sequential memory.

wrap :

1. A wrapping burst functions like an incrementing burst, but the address cycles back to the lower address upon reaching a predefined upper address boundary.
2. Restrictions to wrap burst :
    a. Start address must be aligned to the size of each transfer.
    b. Length of bursts must be 2,4,8 or 16b transfers
3. The lowest address aligned with the data being transferred is typically used for burst transfer.
    a. Lowest address is the **wrap boundary**.
    b. **Wrap Boundary** = size of each transfer on the burst * total num of transfers
4. It increments till the wrap boundary from the start address, i.e.,
    a. **Incremented address** = wrap_boundary + total size of data to be transferred

5. The initial address may exceed the wrap boundary, indicating that for any WRAP burst starting above this boundary, the address will cycle back.

6.

AxBURST[1:0]	**Burst type**
0b00	FIXED
0b01	INCR
0b10	WRAP
0b11	Reserved

Write Data channel:

1. Data bus width : 8,16,32,64,128,256,512,1024 bits wide
2. Different bus width with for write and read data channels
3. Write strobe feature
4. Different combination of strobes
5. Valid scenario - 8bits data on 32 bits data bus
6. Invalid scenarios - 16bits data on 32 bits data bus but strobes like 111 - invalid

Note: The read channel doesn't require buffer because the data sent from slave with the ID will be routed correctly by the interconnect. Hence, no buffer on this side.

Write Response:

1. Valid default value is 0, other signals can be anything
2. Ready can be low or high in default state.
3. It should be high, only if master can accept the response in 1 clock cycle - transfer in
1 clock cycle
4. If low, it takes 2 clock cycles min.
5. Response is driven only after the write-data transaction is completed.

Read Address:

1. Valid default value is 0, other signals can be anything
2. Ready can be low or high in default state.
3. If high, it should be able to accept the data.- transfer in 1 clock cycle
4. If low, it takes 2 clock cycles min. Hence, not recommended but based on implementation.

Read Response:

1. Valid default value is 0, other signals can be anything
2. Ready can be low or high in default state.
3. If high, master should be able to accept the data.- transfer in 1 clock cycle
4. If low, it takes 2 clock cycles min.

5. Response, along with data, is driven only after the read address transaction is completed.

00 – normal access success, exclusive access failure

01 – exclusive access success

10 – slave error

11 – address decode error

3.6 Handshake Process:

All five transaction channels employ the VALID/READY handshake mechanism for the exchange of address, data, and control details. This bidirectional flow control system allows both the master and the slave to manage the pace of information exchange. The source activates the VALID signal to denote the availability of address, data, or control information. Conversely, the destination activates the READY signal to show its readiness to receive the information. A transfer takes place only when both VALID and READY signals are simultaneously HIGH.

In the illustration provided, the source introduces the address, data, or control information post-T1 and activates the VALID signal. Following this, the destination activates the READY signal post-T2, necessitating the source to maintain the stability of its information until the transfer is executed at T3, upon acknowledgment of this assertion.

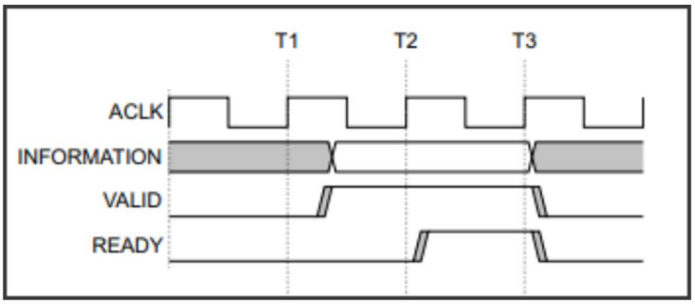


Figure 3.6a: Valid before Ready

Once VALID is activated, it must stay so until the handshake is completed, which happens at a rising clock edge where both VALID and READY are activated.

In another scenario depicted, the destination activates READY post-T1, even before the address, data, or control information is deemed valid, signaling its readiness to receive the information. The source then presents the information and activates VALID post-T2, with the transfer

occurring at T3 upon acknowledgment of this assertion. Here, the transfer is completed in a single cycle.

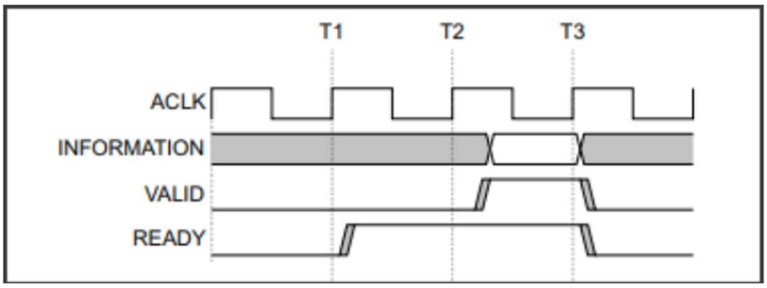


Figure 3.6b: Ready before Valid

The destination is allowed to wait for VALID to be activated before it activates READY. If READY is deactivated before VALID is activated, it's permissible to deactivate READY.

In a further illustration, both the source and destination indicate post-T1 their readiness to proceed with the transfer of address, data, or control information. In this case, the transfer takes place at the moment of the rising clock edge when the VALID and READY signals are both recognized, indicating that the transfer happens at T2.

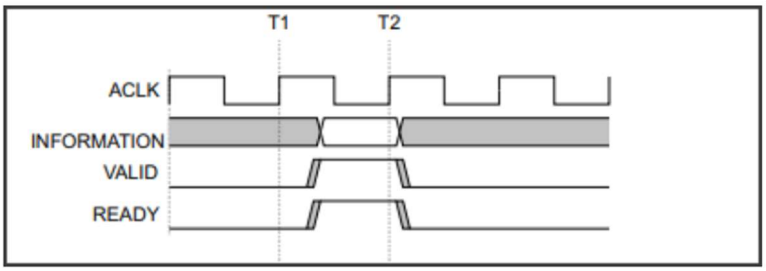


Figure 3.6c: Ready with Valid

3.7 Relationships between the channels:

Dependencies in Read Transactions

The diagram below illustrates the dependencies of handshake signals in a read transaction, revealing that:

1. The master is not required to delay for the slave's ARREADY signal before sending out ARVALID.
2. The slave has the option to wait for the ARVALID signal before issuing ARREADY.
3. The slave is permitted to issue ARREADY even if ARVALID has not yet been sent.
4. The slave should wait until both ARVALID and ARREADY are active before sending RVALID to signal the availability of valid data.
5. The slave should not hold off on sending RVALID waiting for the master's RREADY signal.
6. The master may wait for the RVALID signal before issuing RREADY.
7. The master is allowed to issue RREADY even before RVALID is sent.

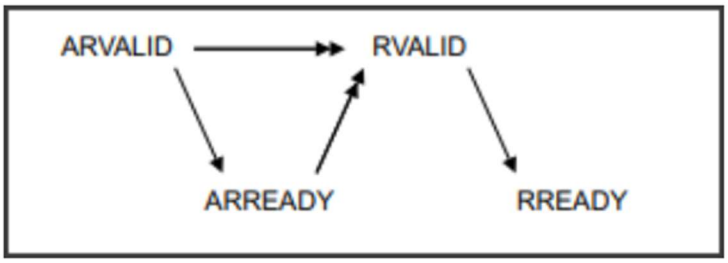


Figure 3.7a: Read transaction dependencies

Dependencies in Write Transactions

1. The master should not delay issuing AWVALID or WVALID waiting for the slave's AWREADY or WREADY signals.
2. The slave has the option to wait for either AWVALID, WVALID, or both before issuing AWREADY.
3. The slave is permitted to issue AWREADY before either AWVALID or WVALID, or both, have been sent.
4. The slave may wait for either AWVALID, WVALID, or both before issuing WREADY.
5. The slave is allowed to issue WREADY even if AWVALID or WVALID, or both, have not yet been sent.
6. The slave must wait until both WVALID and WREADY are active before issuing BVALID. Additionally, the slave must wait for WLAST to be active before issuing BVALID, as the write response, BRESP, should only be signaled after the final data transfer in a write transaction.
7. The slave should not delay issuing BVALID waiting for the master's BREADY signal.
8. The master may wait for BVALID before issuing BREADY.

9. The master is allowed to issue BREADY even before BVALID is sent.

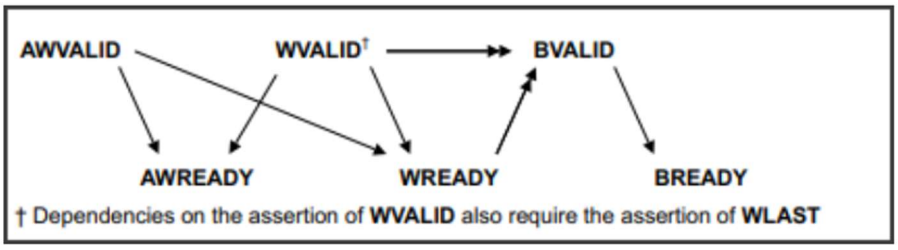


Figure 3.7b: Write transaction dependencies

Dependencies for Write Responses

1. The slave has the option to wait for either AWVALID, WVALID, or both to be asserted before it asserts AWREADY.
2. The slave is allowed to assert AWREADY even if AWVALID, WVALID, or both have not yet been asserted.
3. The slave may wait for either AWVALID, WVALID, or both to be asserted before it asserts WREADY.
4. The slave is permitted to assert WREADY even before AWVALID, WVALID, or both are asserted.
5. Before asserting BVALID, the slave must ensure that AWVALID, AWREADY, WVALID, and WREADY have all been asserted. Additionally, the slave must wait for WLAST to be asserted before asserting BVALID, as the write response, BRESP, should only be indicated after the completion of the last data transfer in a write transaction.
6. The slave should not delay asserting BVALID in anticipation of the master's BREADY signal.
7. The master has the option to wait for BVALID to be asserted before asserting BREADY.
8. The master is allowed to assert BREADY even if BVALID has not yet been asserted.

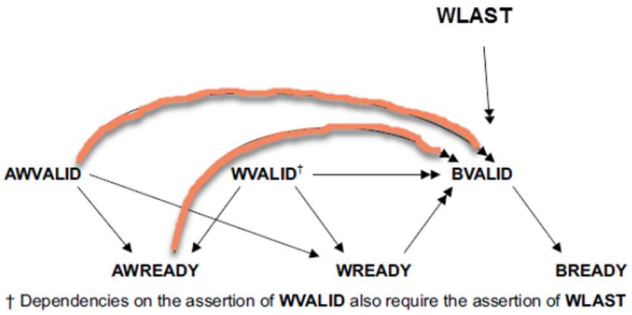


Figure 3.7c: Write Response dependencies

4. Architecture

4.1 AXI4 Full Testbench Architecture

The verification of the AXI protocol's diverse features is achieved through the creation of a Verification IP (VIP). Presented below is the adaptable architecture of a UVM testbench (TB) designed for this verification purpose. The architecture's flexibility is provided by several adjustable parameters, such as:

- **Number of Master Agents**: This setting allows users to define the quantity of master agents incorporated into the testbench, which are entities that initiate transactions on the AXI bus.
- **Number of Slave Agents**: This parameter specifies the count of slave agents within the testbench. These agents are tasked with responding to the transactions initiated by the master agents.
- **Active & Passive Agents**: The testbench can be configured to include both active agents, which directly initiate transactions or engage actively in the protocol, and passive agents, which monitor the transactions and interactions without direct participation.
- **Virtual Interface**: This configuration option pertains to the setup of a virtual interface that links the UVM testbench components to the actual design under test (DUT) signals. It allows for customization of the virtual interface to align with the specific needs of the DUT and the verification setup.

These adjustable parameters enable the customization of the UVM testbench to verify the AXI protocol thoroughly and flexibly across different scenarios and configurations.

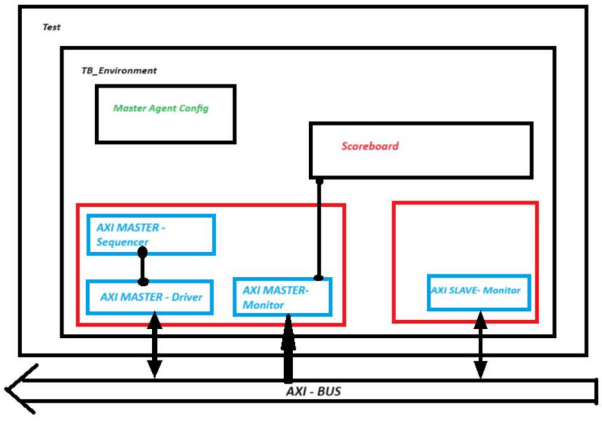


Figure 4.1: AXI VIP TB Architecture

The AXI VIP is comprised of several components, as outlined below:

- **AXI Test**: This class encapsulates the environment and sets up the configuration object within the configuration database. During the run phase, it initiates the generation of stimuli by launching the sequence.
- **AXI Environment**: This component houses various UVM components such as agents, scoreboard, virtual sequencer, etc., which are essential for the testbench. These components are selected based on the configuration parameters, and the environment also establishes connections between agent monitors and the scoreboard through the TLM interface.
- **AXI Master & Slave Agents**: These agents include a monitor, driver, and sequencer, which are activated based on the 'is active' status in the configuration database. The agents link the driver and sequencer through the TLM interface.
- **Sequence Item**: Represents a data item that adheres to the DUT protocol.
- **Sequence**: Responsible for generating sequence items.
- **Sequencer**: Directs the sequence items from the sequence to the driver.
- **Driver**: Executes the transaction-level signals onto the interface following the DUT protocol.
- **Monitor**: Gathers signals from the interface, transforms them into transactions, and disseminates these transactions to the scoreboard, coverage models, etc.
- **Scoreboard**: Conducts comparisons between the reference data and the DUT outputs and executes the coverage models.
- **Configuration Object**: Serves as a repository object to relay information from the test to the subordinate components, influencing their operations, construction, and interconnections.

4.2 Implementation

The described read and write system is realized through a UVM Testbench framework, comprising two types of agents: a master agent and a slave agent. The slave agent is equipped with a driver, while the master agent encompasses all UVM components [3].

In the AXI protocol, the Master device initiates a burst transaction by providing only the starting address. It falls upon the slave to determine subsequent addresses based on the burst's length and the size of each transfer within the burst, as specified by the Master. Additionally, the Master

assigns an AWID to the address. Data transmission to slaves and the corresponding responses are directed to the appropriate Master by recognizing these AWIDs. When the Master issues a valid address, it signals this by setting the VALID signal to HIGH. Upon readiness to receive the signals from the Master, the slave signals its acknowledgment by setting the READY signal to HIGH. The Master is required to maintain the address and other control information, such as AWBURST, AWLEN, AWSIZE, etc., until the slave acknowledges receipt by setting the READY signal to HIGH. Signals are organized into channels for clarity and organization. The Master sends the initial write address of the burst along with control signals through the write address channel like AWSIZE, AWBURST, AWLEN, etc. The write data channel is used by the Master to transmit write data to the slave. Acknowledgment from the slave to the Master is conveyed through the write response channel. For the entire burst operation in a write transaction, only one response is sent back to the Master, and this is done via a distinct channel. An AXI transaction is considered complete once the slave has returned a response to the Master for the transmitted data. Through the read address channel, the Master sends the address and control signals such as ARSIZE, ARBURST, ARLEN, etc., to the slave. The slave then calculates subsequent addresses based on the control information provided by the Master. The slave transmits read data back to the Master through the read data channel and is responsible for acknowledging each piece of read data. Thus, the slave is capable of sending both read data and acknowledgments through a single channel, the read data channel, eliminating the need for a separate read response channel in AXI.

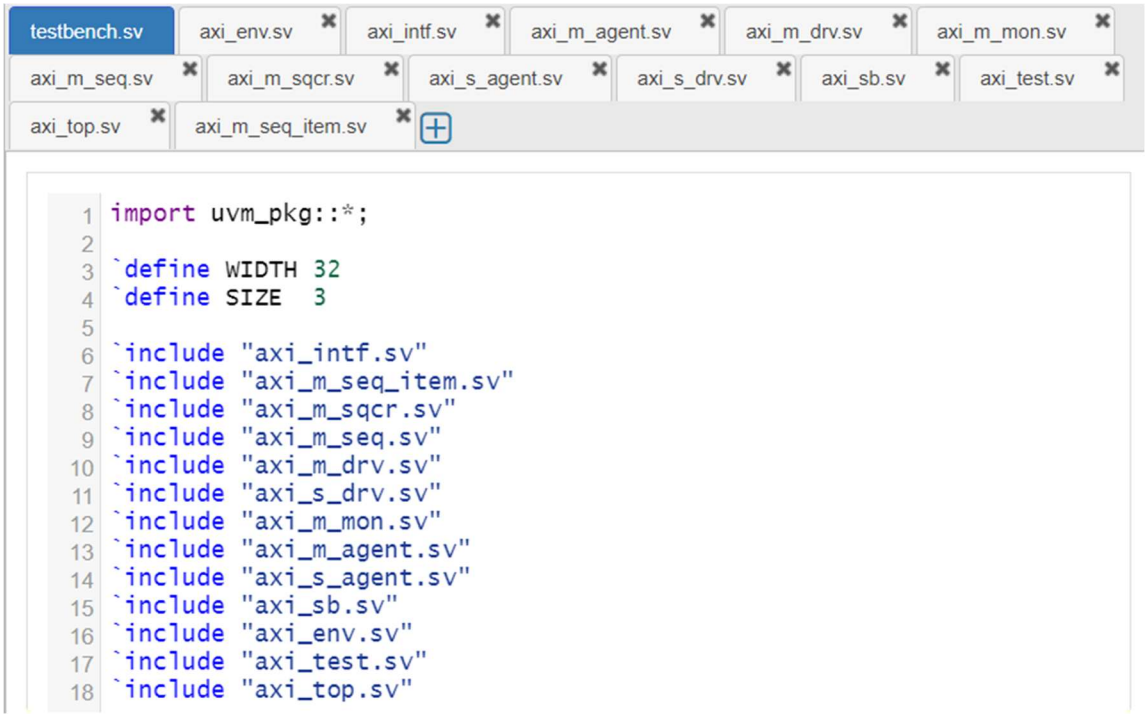


Figure 4.2: EDA Playground AXI TB Code

4.3 DMA IP AXI4 Architecture

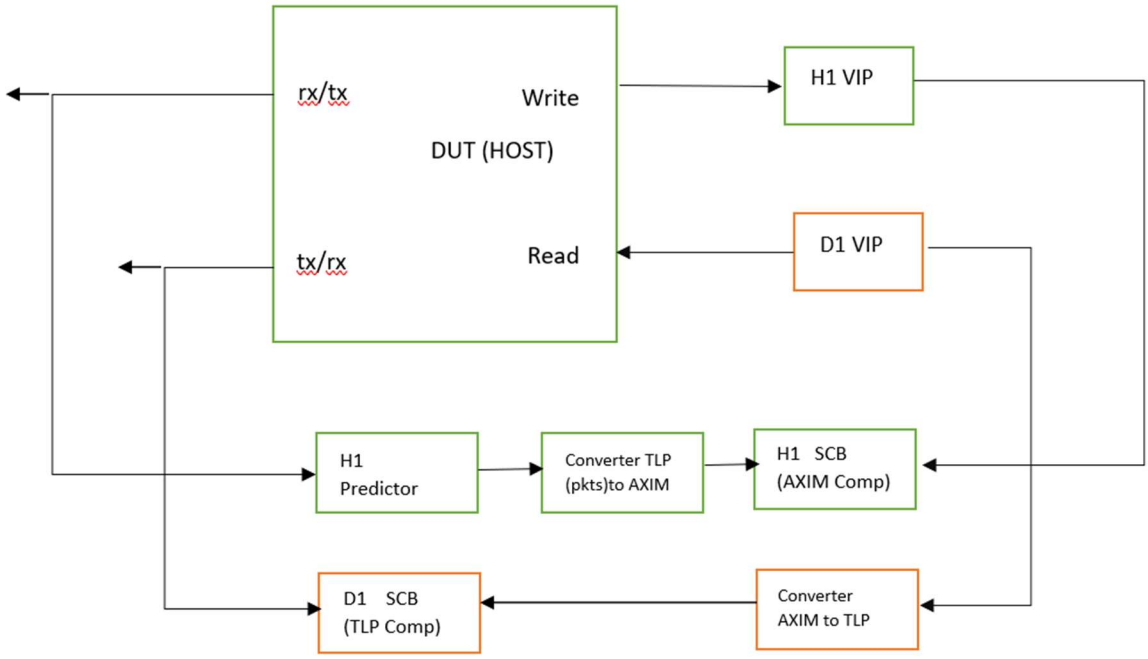


Figure 4.3: DMA IP AXI4 Architecture

The above Figure illustrates the testbench architecture used to verify the DMA IP with AXI Interface. Here in this architecture when data is moving from Host to Device only the write operation is performed and when data is moving from Device to Host read operation is performed, in both the cases DUT acts as the Master and both the VIPs (H1 and D1) are acting as a slave.

The DUT has two sides on the right side which has AXIM Interface there is H1 VIP is used for Host to Device Data Movement and D1 VIP is used for Device to Host Data Movement and on the left-hand side there are receiver and transmitting signals from where the expected packets are driven from/to the DUT which are connected with PCIE interface. Whenever any write operation is performed DUT asserts the AWVALID and whenever H1 VIP is ready it acknowledges the signal by asserting AWREADY to the DUT, all the data that is to be written on the device sides comes out in the form of AXIMM packets, goes to the AXIM Scoreboard as actual packet on other side rx signal transfer the data to Host-to-Device predictor in the form of PCIE packets which we normally call as TLP packets and predictor predicts the expected data that needs to be written and then Converter converts the TLP packets into AXIM packets so that scoreboard

can compare for the packets actual and expected as it is an AXIM scoreboard so TLP packets need to be converted.

Similarly, When Dut wants to read some data it will assert the ARVALID to the D1 VIP which will acknowledge the signal with ARREADY whenever it is ready to receive the read request from the master, then all the data that is to be read by DUT is transferred to the converter in the form of AXIM packets which is the actual packets, here we are converting the AXIM to TLP packets because for Device-to-host the available scoreboard which can only compare TLP packets and this is one of the advantages of UVM to use the already existing components and molding them as per our functional needs. So Device-to-Host TLP scoreboard will receive the expected packet through DUT or some PCIE component and the actual packet from the converter, finally compares them and if the actual packets match the expected packet then the test passes otherwise it fails.

4.4 Verification Flow

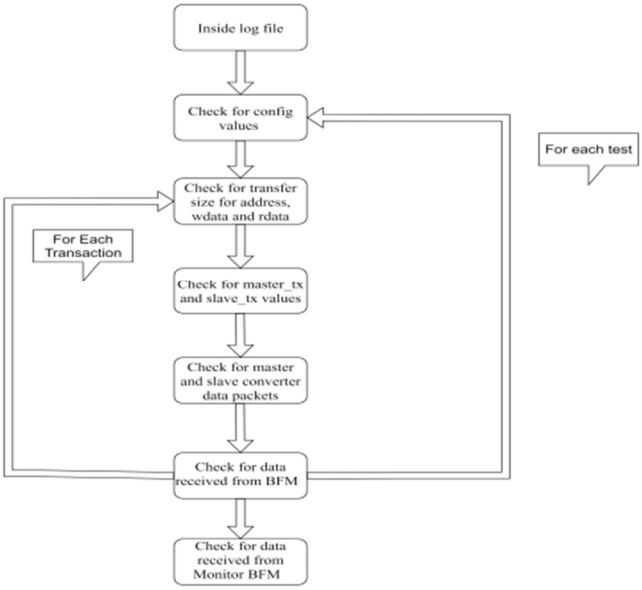


Figure 4.4: Debugging Flow

Step 1:

Create the log file or report which has all the messages and generate the waveform using Synopsis tool.

Step 2:

Go to the log, then inside the log try to check for all the values related to the master agent, slave

agent, and environment configurations that have been generated from the test.

Step 3:

Once the config values are correct then check for the address and data to be transmitted from master_tx class as well from slave_tx class. In master_tx_class check for the transaction type, there are write type and read type and check size, length, id address and the data.

Step 4:

Once the data has been randomized and sent to master or slave BFMs. The master driver BFM will drive the write transaction and the slave samples the data depending on the configurations of the master and similarly slave driver BFM will drive the data and response and sample the write data depending on the configurations of a slave [4].

The master driver BFM will print the write task data which has been driven by the master and write response sampled data and similarly, slave driver BFM will print which has been driven by the slave and sampled data. In the end both the master BFM and slave BFM data must be the same.

Step 5:

Once the data has been driven or sampled from master and slave driver, monitor will capture the data for all the write and read channels and it will print the sampled data for each of the channels.

Check the values for all the channels in Master Monitor:

- Sampled data from monitor bfm : write address channel
- Sampled data from monitor bfm : write data channel
- Sampled data from monitor bfm : write response channel
- Sampled data from monitor bfm : read address channel
- Sampled data from monitor bfm : read data channel

Step 6:

And finally we have scoreboard checks which basically compare the rid from master and the slave, raddr of the master and the slave, rlen of the master and the slave, rsize of the master and the slave, and another transfer signal from the master and the slave.

4.5 Waveforms and Logs:

1. Passed Test

- **Write Operation Waveform:**

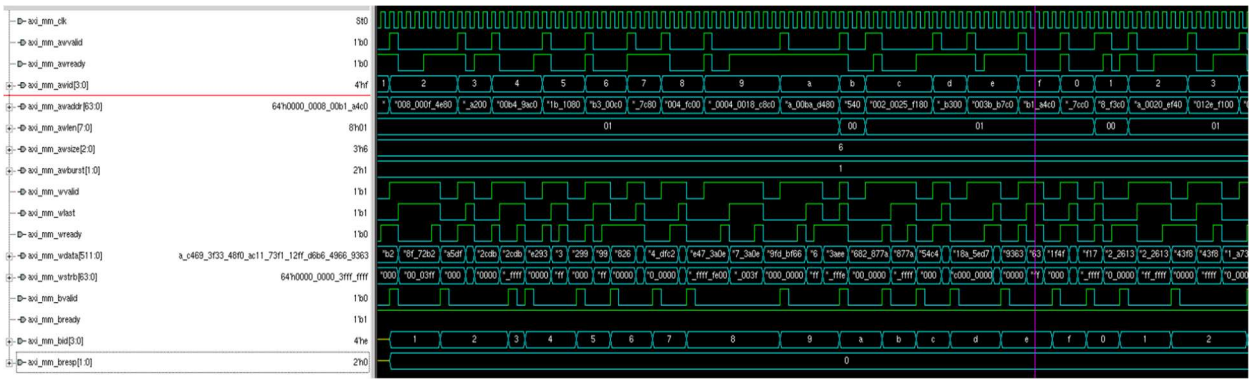


Figure 4.5a: Write Waveform

- **Write Operation Log:**

```
UVM_INFO /nfs/../../../../../../../../../../../../../../../../../../../../testbench/env/scoreboard/aximm_tx_scb.sv(781) @ 654175000.00 ps: uvm_
test_top.env_inst[0].aximm_tx_scb Packets Comparision Successfull
=============================================================================================
| No.of. Expected Pkt    |  No.of. Actual Pkt     |  No.of. Pkt Matched  |  No.of. Pkt Mis-Matched  |
|------------------------|------------------------|----------------------|--------------------------|
|                     54 |                     54 |                   54 |                        0 |
=============================================================================================

UVM_INFO /nfs/../../../../../../../../../../../../../../../../../../../../testbench/tests/base_test.sv(3322) @ 654175000.00 ps: uvm_test_top
[final_phase] Entered...
UVM_INFO /nfs/../../../../../../../../../../../../../../../../../../../../testbench/tests/base_test.sv(3341) @ 654175000.00 ps: uvm_test_top
[final_phase]
*********************************************************************************************************************
********************   Test Sequence:              wr_rand_seq      :: Passed    ********************
*********************************************************************************************************************


--- UVM Report catcher Summary ---


Number of demoted UVM_FATAL reports  :    0
Number of demoted UVM_ERROR reports  :    0
Number of demoted UVM_WARNING reports:    0
Number of caught UVM_FATAL reports   :    0
Number of caught UVM_ERROR reports   :    0
Number of caught UVM_WARNING reports :    0
```

Figure 4.5b: Write Simulation Log

The above figure illustrate that when write operation was performed and there was no UVM_ERROR or UVM_FATAL was observed. Sequennce name is wr_rand_seq which passed in 654175000.00 ps. Total no of packets that were transferred in this particular test is 54.

- **Read Operation Waveform:**

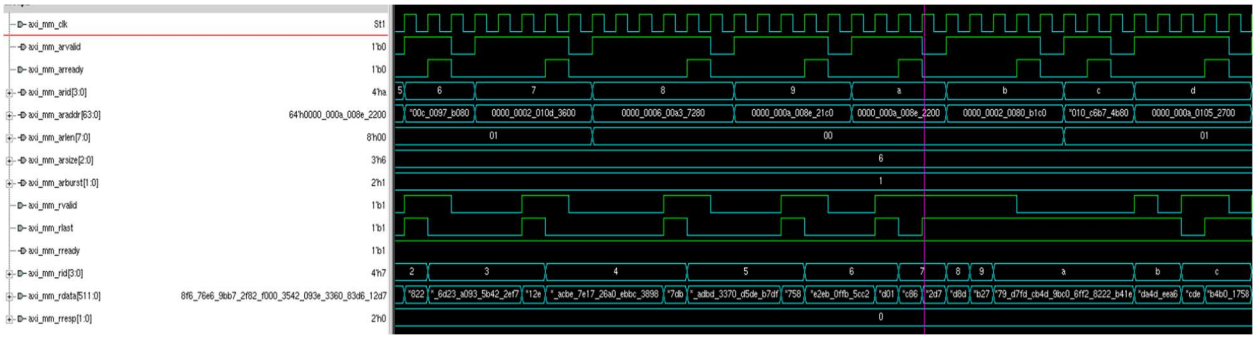


Figure 4.5c: Read Waveform

- **Read Operation Log:**

```
UVM_INFO /nfs/../../../../../../../../../../../../../../../../../../../testbench/env/scoreboard/aximm_tx_scb.sv(781) @ 553070000.00 ps: uvm_
test_top.env_inst[0].aximm_tx_scb Packets Comparision Successfull
================================================================================================
| No.of. Expected Pkt    |  No.of. Actual Pkt     |  No.of. Pkt Matched   |  No.of. Pkt Mis-Matched  |
|------------------------|------------------------|-----------------------|--------------------------|
|                     52 |                     52 |                    52 |                        0 |
================================================================================================

UVM_INFO /nfs/../../../../../../../../../../../../../../../../../../../testbench/tests/base_test.sv(3322) @ 553070000.00 ps: uvm_test_top
[final_phase] Entered...
UVM_INFO /nfs/../../../../../../../../../../../../../../../../../../../testbench/tests/base_test.sv(3341) @ 553070000.00 ps: uvm_test_top
[final_phase]
************************************************************************************************************
********************   Test Sequence:              rd_rand_seq      :: Passed    ********************
************************************************************************************************************


--- UVM Report catcher Summary ---


Number of demoted UVM_FATAL reports  :    0
Number of demoted UVM_ERROR reports  :    0
Number of demoted UVM_WARNING reports:    0
Number of caught UVM_FATAL reports   :    0
Number of caught UVM_ERROR reports   :    0
Number of caught UVM_WARNING reports :    0
```

Figure 4.5d: Read Simulation Log

The above figure illustrate that when write operation was performed and there was no UVM_ERROR or UVM_FATAL was observed. Test name is rd_rand_seq which passed in 553070000.00 ps. Total no of packets that were transferred in this particular test is 52.

2. Failed Test

- **Waveform:**

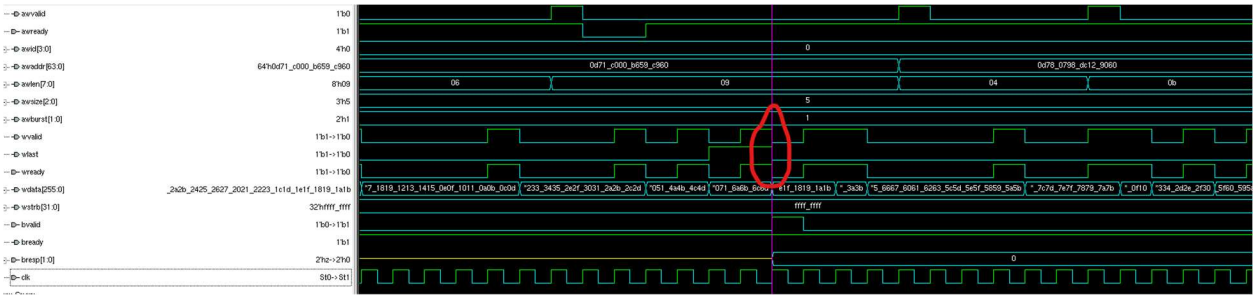


Figure 4.5e: Failed Test Waveform

- **Log:**

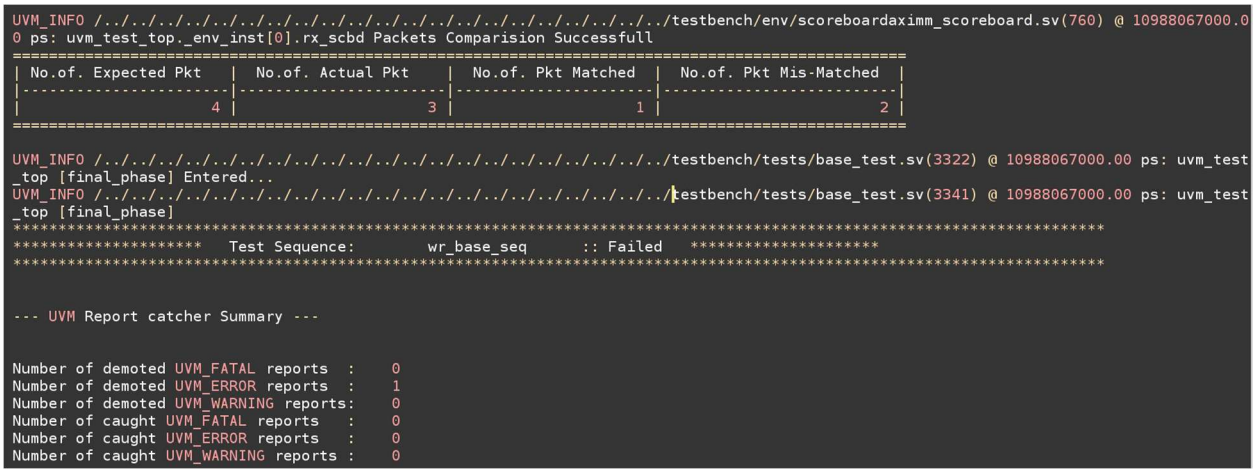


Figure 4.5f: Failed Test Log

The above figure illustrates that the test has failed because of the UVM_ERROR as no. of expected packet did not match with the actual packet and only one of the packets matches and rest did not match.

UVM_ERROR /../../../../../../../../../../testbench/env/scoreboard/aximm_scoreboard.sv(489)@4673000.00ps:uvm_test_top.env_inst[0].rx_scbd [RX_SCB_EXP_NOT_MATCH_WITH_ACT]

This Error is due to the WLAST that is asserted before all the data has been transmitted from master to slave even though WDATA is still transmitting the data. Hence this bug was found to be RTL issue hence Design team was informed regarding the same. Same bug will be re verified once the designers fix the issue.

5. Conclusion

The project on UVM Environment Bring-Up for AXI4 Full Based DMA (Direct Memory Access) IP has successfully demonstrated the effectiveness of a structured and systematic approach to IP verification. Through the meticulous establishment of a UVM environment tailored for the AXI4 protocol and DMA IP, we have achieved comprehensive verification coverage, ensuring that the DMA IP operates reliably and efficiently in various scenarios and configurations.

Key achievements of this project include:

1. **Robust Verification Framework:** The development and deployment of a robust UVM-based verification framework specifically designed for AXI4 Full Based DMA IP. This framework facilitated the creation of reusable verification components, enhancing the efficiency of the verification process.
2. **Effective Debugging and Optimization:** The project highlighted the importance of an iterative verification process, where debugging and optimization played critical roles. By systematically addressing issues identified during simulations, we enhanced the reliability and performance of the DMA IP.
3. **Collaboration and Knowledge Sharing:** The project fostered a collaborative environment among the verification team, designers, and other stakeholders. This collaboration, coupled with effective knowledge sharing, contributed to a deeper understanding of the AXI4 protocol and DMA IP, facilitating a more efficient verification process.
4. **Future-Ready Verification Environment:** The UVM environment established for this project is not only effective for the current verification needs but also scalable and adaptable for future projects. This ensures a solid foundation for verifying upcoming IPs and technologies, thereby reducing time-to-market and improving product quality.

In summary, the UVM Environment Bring-Up for AXI4 Full Based DMA IP project has set a high standard for IP verification within our organization. It underscores the critical role of a well-structured verification methodology in the development of reliable and high-performance Ips.

References

[1] S. Palntikar. Verilog HDL Guide to Digital System Design and Synthesis, 2nd Edition, Pearson.

[2] PCIe* Multi Channel DMA IP document.

[3] Mentor Graphics, “UVM Cookbook”, Online Methodology Documentation from the Verification Methodology Team.

[4] UVM based testbench architecture for logic sub-system verification 2017 International Conference on Technological Advancements in Power and Energy (TAP Energy).

[5] A Technical RoadMap from System Verilog to UVM", International Journal on Recent and Innovation Trends in Computing and Communication, vol. 3, no. 3, pp. 1302-1306, 2015.

[6] Anjali and J. P. Anita, "AXI based DMA Memory System Testbench Architecture Using UVM Harness Technique," *2019 9th International Conference on Advances in Computing and Communication (ICACC)*, Kochi, India, 2019, pp. 152-157.

[7] V. Melikyan, S. Harutyunyan, A. Kirakosyan and T. Kaplanyan, "UVM Verification IP for AXI," *2021 IEEE East-West Design & Test Symposium (EWDTS)*, Batumi, Georgia, 2021, pp. 1-4.

[8] H. Sangani and U. Mehta, "UVM based Verification of Read and Write Transactions in AXI4-Lite Protocol," *2022 IEEE Region 10 Symposium (TENSYMP)*, Mumbai, India, 2022, pp. 1-5.

[9] F. Plasencia-Balabarca, E. Mitacc-Meza, M. Raffo-Jara and C. Silva-Cárdenas, "A Flexible UVM-Based Verification Framework Reusable with Avalon, AHB, AXI and Wishbone Bus Interfaces for an AES Encryption Module," *2019 IEEE Latin American Test Symposium (LATS)*, Santiago, Chile, 2019, pp. 1-4.

[10] ARM 20 11. AMBA AXI protocol v. I. O specification, ARM Limited, [online] Available: https://developer.arm.com/documentation/ihi0022/latest/.

[11] R. Madan, N. Kumar and S. Deb, "Pragmatic approaches to implement self-checking mechanism in UVM based TestBench," 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, 2015.

[12] S. S. Math, R. B. Manjula, S. S. Manvi and P. Kaunds, "Data transactions on system-on-chip bus using AXI4 protocol," *2011 INTERNATIONAL CONFERENCE ON RECENT ADVANCEMENTS IN ELECTRICAL, ELECTRONICS AND CONTROL ENGINEERING*, Sivakasi, India, 2011, pp. 423-427.

[13] M. P. Deepu and R. Dhanabal, "Validation of transactions in AXI protocol using system verilog," *2017 International conference on Microelectronic Devices, Circuits and Systems (ICMDCS)*, Vellore, India, 2017, pp. 1-4.

[14] G. Mahesh and S. M. Sakthivel, "Verification of memory transactions in AXI protocol using system verilog approach," *2015 International Conference on Communications and Signal Processing (ICCSP)*, Melmaruvathur, India, 2015, pp. 0860-0864.

[15] Gayathri M, R. Sebastian, S. R. Mary and A. Thomas, "A SV-UVM framework for Verification of SGMII IP core with reusable AXI to WB Bridge UVC," *2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS)*, Coimbatore, India, 2016, pp. 1-4.

[16] H. -Y. Yang, "Highly automated and efficient simulation environment with UVM," *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, Hsinchu, Taiwan, 2014, pp. 1-3.

[17] A. S. Chandgaonkar, V. Ingale, V. Patil and V. Agarwal, "Development Of SV UVM Testbench For Verification Of AMBA AXI3 IP Used For Memory Access Application," *2023 14th International Conference on Computing Communication and Networking Technologies (ICCCNT)*, Delhi, India, 2023, pp. 1-6.

[18] H. Sangani and U. Mehta, "UVM based Verification of Read and Write Transactions in AXI4-Lite Protocol," *2022 IEEE Region 10 Symposium (TENSYMP)*, Mumbai, India, 2022, pp. 1-5.

[19] SystemVerilog TestBench. (2020,April12). Verification Guide. https://verificationguide.com/systemverilog/systemverilog-testbench/

[20] Vlsi4freshers.UVM.https://www.vlsi4freshers.com/2020/04/uvm-testbench-architecture.

[21] Documentation- Arm Developer. (n.d.). APB Interface. https://developer.arm.com/documentation/ihi0024/c/Introduction/About-the-AXI-protocol

[22] Arm Ltd. (n.d.). AMBA 4 / AMBA 3 / AMBA 2. Arm | The Architecture for the Digital World. https://www.arm.com/architecture/system-architectures/amba/amba-4

[23] Verification IP. https://www.aldec.com/

[24] Verification Guide: (2020, March17) Verification Guide. https://verificationguide.com

Thesis

ORIGINALITY REPORT

9%	4%	5%	2%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Shaila S. Math. "Data transactions on system-on-chip bus using AXI4 protocol", 2011 INTERNATIONAL CONFERENCE ON RECENT ADVANCEMENTS IN ELECTRICAL ELECTRONICS AND CONTROL ENGINEERING, 12/2011 Publication	2%
2	Hardi Sangani, Usha Mehta. "UVM based Verification of Read and Write Transactions in AXI4-Lite Protocol", 2022 IEEE Region 10 Symposium (TENSYMP), 2022 Publication	1%
3	Gayathri M, Rini Sebastian, Silpa Rose Mary, Anoop Thomas. "A SV-UVM framework for Verification of SGMII IP core with reusable AXI to WB Bridge UVC", 2016 3rd International Conference on Advanced Computing and Communication Systems (ICACCS), 2016 Publication	1%
4	community.arm.com Internet Source	1%

5	Submitted to Eastern Mediterranean University Student Paper	1 %
6	Submitted to RMIT University Student Paper	<1 %
7	fccid.io Internet Source	<1 %
8	Submitted to University of Strathclyde Student Paper	<1 %
9	www.mdpi.com Internet Source	<1 %
10	www.scientific.net Internet Source	<1 %
11	ipfs.io Internet Source	<1 %
12	summit.sfu.ca Internet Source	<1 %
13	technodocbox.com Internet Source	<1 %
14	data.epo.org Internet Source	<1 %
15	usermanual.wiki Internet Source	<1 %
16	www.chipdesignmag.com	

Internet Source

<1%

Exclude quotes	On	Exclude matches	< 8 words
Exclude bibliography	On		

Thesis

GRADEMARK REPORT

FINAL GRADE

/0

GENERAL COMMENTS

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21

PAGE 22

PAGE 23

PAGE 24

PAGE 25

PAGE 26

PAGE 27

PAGE 28

PAGE 29

PAGE 30

PAGE 31

PAGE 32

PAGE 33

PAGE 34

PAGE 35

PAGE 36

PAGE 37

PAGE 38

PAGE 39

PAGE 40

PAGE 41

PAGE 42

PAGE 43