M. Tech (VLSI Design and Embedded System) Sumer Saini 2023

AXI BASED IP BLOCK VERIFICATION USING UVM

A DISSERTATION REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTSFOR THE AWARD OF THE DEGREE

OF

MASTER OF TECHNOLOGY

IN

VLSI DESIGN & EMBEDDED SYSTEM

SUBMITTED BY:

SUMER SAINI

2K21/VLS/19

UNDER THE SUPERVISION OF

Dr. ALOK KUMAR SINGH
PROFESSOR

ELECTRONICS & COMMUNICATION ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering)
Bawana Road, Delhi-110042

MAY 2023

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)Bawana Road, Delhi-110042

CANDIDATE'S DECLARATION

I, Sumer Saini, Roll No. 2K21/VLS/19 student of M. Tech (VLSI & Embedded systems), hereby declare that the project Dissertation titled **"AXI Based IP Block Verification using UVM"** which is submitted by me to the Department of Electronics and Communication Engineering, Delhi Technological University, Delhi in partial fulfillment of the requirement for the award of the degree of Master of Technology is unique and has not been copied without proper citation. This work has never before been used to give a degree, diploma associateship, fellowship, or other equivalent title or recognition.

Place: Delhi **Sumer Saini**

Date: 1-05-2023 **(2K21/VLS/19)**

ELECTRONICS & COMMUNICATION ENGINEERING

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering)

Bawana Road, Delhi-110042

CERTIFICATE

I hereby certify that the Project Dissertation titled **"AXI Based IP Block Verification using UVM"** which is submitted by **Sumer Saini, 2K21/VLS/19**, to the Department of Electronics & Communication Engineering, Delhi Technological University, Delhi in partial fulfillment of the prerequisite for the award of the degree of Master of Technology, is a documentation of the student's projectwork completed under my supervision. To the best of my knowledge, this work hasnever been submitted in part or in full for any degree or diploma at this university or anywhere else.

Place: Delhi
Date: 1-05-2023

SUPERVISOR
Dr A. K. Singh,
Professor (ECE)

ABSTRACT

In modern VLSI, a DUT may have multiple interfaces. Each of these interfaces may have different UVM objects associated with them. In an SoC there are no. of IP blocks which are connected to each other with the help of interfaces. One such interface which is most commonly used is AXI interface. This project involves the verification of an AXI interface module that acts as an initiator or target depending on the upstream or downstream transaction. The module converts commands from the FIFO BFM or AXI BFM to AXI transactions or FIFO transactions. The testbench development is based on System Verilog using UVM methodology to create a constrained random test environment for faster verification. The testbench includes bus functional models of AXI and FIFO interfaces, which act as initiators/targets on the DUT's interfaces.

The AXI interface is a widely used interface in digital designs and is popularly used for interconnecting different IPs within a SoC or FPGA. The AXI interface is complex, and verification of designs with AXI interfaces is challenging. The primary goal of this project is to verify the AXI interface module, which acts as an initiator or target depending on the upstream or downstream transaction. The AXI interface module converts the commands from the FIFO BFM or AXI BFM to AXI transactions or FIFO transactions. The verification is done using the UVM methodology, which creates a constrained random test environment, which helps in faster verification. The testbench includes bus functional models of AXI and FIFO interfaces, which act as initiators/targets on the DUT's interfaces.

The AXI interface is a high-performance, high-frequency, and low-latency interface used in digital designs. The AXI interface provides the features of burst transfers, read and write data transfers, and different transfer sizes, including 8-bit, 16-bit, 32-bit, and 64-bit transfers. The AXI interface

also provides various features like data caching, out-of-order execution, and multiple outstanding transactions.

The AXI interface module that is being verified in this project acts as an initiator or target depending on the upstream or downstream transaction. The module converts the commands from the FIFO BFM or AXI BFM to AXI transactions or FIFO transactions. The module is complex, and verifying its functionality requires a robust testbench.

The UVM methodology is widely used for the verification of digital designs. The UVM methodology provides a standardized way of developing testbenches and allows for the creation of a constrained random test environment. The UVM methodology is based on the Object Oriented Programming(OOPs) concept, and it includes classes, objects, and virtual sequences. The testbench is an integral part of the verification process. The testbench includes the bus functional models of AXI and FIFO interfaces, which act as initiators/targets on the DUT's interfaces. The testbench verifies the functionality of the AXI interface module by generating random transactions and comparing the results with the expected output. The testbench also includes assertions and coverage to check the correctness and completeness of the verification process.

In this project verification of AXI interface using IP system interface has done. In this entire designtwo units are there i.e initiator and target for upstream and downstream transactions.

ACKNOWLEDGEMENT

A successful project can never be prepared by the efforts of the person to whom the project is assigned, but it also demands the help and guardianship of people who helped in completion of the project. We would like to thank all those people who have helped us in this research and inspired us during our study.

With profound sense of gratitude, we thank Dr A. K. Singh, Professor (ECE), our Research Supervisors,for his encouragement, support, patience and guidance in this project work. We heartily appreciate the guidance given by him in the project presentation that has improved our presentation skills with his comments and advices. We take immense delight in extending Our acknowledgement to our families and friends who have helped us throughout this projectwork.

Sumer Saini
2K21/VLS/19

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

S.No	Abbreviation	Full Name
1.	UVM	Universal Verification Methodology
2.	OVM	Open Verification Methodology
3.	AXI	Advanced Extensible Interface
4.	FIFO	First In First Out
5.	BFM	Bus Functional Model
6.	DUT	Design Under Test
7.	TLM	Transaction level Modelling
8.	DUV	Design Under Verification
9.	DSP	Digital Signal Processing
10.	SOC	System On Chip
11.	AMBA	Advanced Microcontroller Bus Architecture
12.	APB	Advanced Peripheral Bus
13.	AHB	Advanced High performance Bus
14.	IP	Intellectual Property
15.	VLSI	Very Large-Scale Integration
16.	FPGA	Field Programmable Gate Arrays

CHAPTER 1

INTRODUCTION

The UVM is a standardized methodology for verifying integrated circuit designs. It is derived mainly from the OVM. The UVM class library brings much automation to the SystemVerilog language, such as sequences and data automation features. Unlike the previous methodologies developed independently by the simulator vendors, UVM is an Accellera standard with support from multiple vendors. The UVM class library consists of the following components:

- Components: The UVM component library provides a set of base classes for building verification components, such as drivers, monitors, and scoreboards.
- Sequences: The UVM sequence library provides a set of classes for generating stimulus and checking for responses.
- Transaction-level modeling (TLM): The UVM TLM library provides a set of classes for modeling the interface between the DUV and the verification environment.
- Reporting: The UVM reporting library provides a set of classes for generating reports about the verification environment.
- Utilities: The UVM utilities library provides a set of classes for common tasks, such as creating and managing objects. The UVM class library can be used to create a well-structured and reusable verification environment. The UVM class library is a powerful tool for verifying integrated circuit designs.

- Increased productivity: UVM can help to increase productivity by providing a set of reusable components and a standardized methodology.
- Improved quality: UVM can help to improve quality by providing a more structured and organized verification environment.

In the coming years, there will be an enormous growth in the demand for data and network capacity. The development of fifth-generation communication technology (5G) can offer peak data speeds of up to 20Gbps with extremely low latency of 1ms. Future applications for high-performance computing include wireless communication, multimedia processing, atmospheric simulation, and high-performance graphics. DSP will be crucial in this since it offers powerful

computational capability on a big scale. Direct access to storage no longer meets the demands for data access speed due to the growing demand for storage space and access bandwidth in 5G applications [1].

Mandating a universal convention in verification techniques encouraged the development of generic verification components that were portable from one project to another. This promoted cooperation and sharing of techniques among the user base. It also encouraged the development of verification components that were generic enough to be easily extended and improved without modifying the original code.

Here are some of the benefits of using generic verification components:

- Portability: Generic verification components can be used in different projects without having to be modified. This saves time and effort, and it also helps to ensure that the verification environment is consistent across projects.
- Reusability: Generic verification components can be reused in different projects, which can further save time and effort.
- Extensibility: Generic verification components can be easily extended to accommodate new features or changes to the design. This makes it easier to keep the verification environment up-to-date with the design.
- Maintainability: Generic verification components are easier to maintain than custom-developed components. This is because they are well-documented and well-tested, and they have a large user base that can provide support.

Directed Testbench design was a conventional way to verify design but today's design is highly complex and it is impossible to handle verification with directed testbench. As the complexity of the design increases, bugs in the design also rise. Now it has become a necessity to replace the conventional test with an advanced and automated verification environment in order to reduce time and reliability.

1.1 UVM Classes

UVM have three main classes,

- uvm_object class
- uvm_transaction class

- uvm_component class

A factory is a commonly-used concept in object-oriented programming. It is an object that is used for instantiating other objects. The UVM factory is a mechanism that allows users to create UVM objects and components. There are two ways to register an object with the UVM factory:

- In the declaration of class A, one can invoke the uvm_object_utils(A) or uvm_component_utils(A) registration macros.
- Otherwise, the uvm_object_registry(A,B) or uvm_component_registry(A,B) macros can be used to map a string B to a class type A.

UVM allows the use of Macros

name	**function**	**related to**	**parameters**	**Purpose**	**Type of Macro**
`uvm_create	object constructor	`uvm_send	Sequence or Item	to create the object and allow user to set values via overloading or parameterpassing	Sequence action macro
`uvm_send	processor	`uvm_create	Sequence or Item	processes what is created by `uvm_create without randomization	Sequence Action Macros for Pre-Existing Sequences
`uvm_do	processor	`uvm_create	Sequence or Item	executes class or item with randomization	Sequence action macro

1.2 UVM_OBJECT

Core class based operational methods (create, copy, clone, compare, print, record, etc..), instance identification fields (name, type name, unique id, etc.) and random seeding were defined in it. All uvm_transaction and uvm_component were derived from the uvm_object. In the context of hardware verification, UVM (Universal Verification Methodology) is a popular standardized verification methodology that is used to verify complex digital designs. UVM is built on top of SystemVerilog and provides a set of classes and methodologies for building reusable, scalable, and efficient testbenches. One of the key concepts in UVM is the UVM object. A UVM object is an instance of a class derived from the uvm_object base class. UVM objects are used to represent various elements in a digital design, such as registers, memories, and other functional blocks.

UVM objects are typically created dynamically during the simulation and are managed by the UVM factory. The UVM factory is responsible for creating and deleting UVM objects, as well as maintaining a hierarchy of UVM objects. UVM objects are designed to be reusable and configurable. They can be extended to include additional functionality or modified to meet specific verification requirements. UVM objects can also be used to communicate between different parts of a testbench or to pass data between different phases of the UVM methodology. Overall, UVM objects provide a powerful and flexible mechanism for building modular and reusable testbenches that can be used to verify complex digital designs.

1.3 UVM_TRANSACTION

- Used in stimulus generation and analysis.

1.4 AXI INTERFACE

Modern SOCs mostly use communication protocols from the AMBA family to create synchronised communication, including APB, AHB, and AXI. Because verification is often projected to take up 70% of the whole project time, compared to the design stage's 30%, it becomes a very difficult work to complete throughout the creation of these sorts of SOCs. So, with the aid of a special

testing environment known as testing IP, several engineers are involved in evaluating an on-chip communication protocol associated functional qualities and the synchronisation between the IP's they connect [2].

The AXI interface module is to provide a protocol conversion between AXI initiator / target bus to AXI transactions or fifo transaction which is fundamentally a simple command for hostinterface main data bus transferring register access or DMA cycles to / from SoC. According to, whether DUT is working as an initiator or target which is decided by the signals it receives from AXI bfm or fifo bfm. The AXI interface is a high-performance, high-frequency, and low-latency interface used in digital designs. The AXI interface provides the features of burst transfers, read and write data transfers, and different transfer sizes, including 8-bit, 16-bit, 32-bit, and 64-bit transfers. The AXI interface also provides various features like data caching, out-of-order execution, and multiple outstanding transactions. The two-way handshake mechanism is the foundation of the AXI interface. To let the slave know that the address or data on the channel is genuine, the master sends a valid signal. Slave notifies master that it is prepared to take the data by sending a ready signal. In order for a write transaction to take place, the master must first communicate the necessary address and control information over the write address channel. The write data channel, which can be 32 or 64 bits in size and is where the actual data transmission happens, transports write data from master to slave. Slave asserts the rvalid signal to indicate that the data on the read channel is valid. The master device will assert the rready signal, and the master device will read the data [3].

Fig. 1.1 AXI IP interface

The working of DUT is as follows :

1. The DUT will present upstream request when there is command available from the upstream command FIFO (rd_avail / wr_avail), and pop the command out of the FIFO when the command get (rd_cget / wr_cget) is asserted.

a. AXI interface component will map command avail to the read / write address channel “AxVALID” signal, and return command get (wr_cget) based on the corresponding read / write address channel “AxREADY” signal.

i. If the command indicates write data is associated, AXI interface component will assert the write data channel “WVALID” signal 1 clock after command get (wr_cget) assertion, and continuously popping out the data from the upstream data FIFO for that transaction to AXI data bus, provided the corresponding write data channel “WREADY” signal is also asserted continuously. Given there is no data get signal to the fifo bfm, if “WREADY” signal is de- asserted anytime when the “WVALID” signal is asserted for that transaction, the continuous data popping out will need to be re-directed to a streaming data FIFO instead, and replay fromthis streaming data FIFO to the AXI data bus when the “WREADY” signal is asserted later.

2. The DUT will present downstream request put where there is a command free (p_free / np_free) fromthe downstream command FIFO.

a. AXI interface component will map command free (p_free / np_free) to the read / write address channel “AxREADY” signal, and return command put (p_cput / np_cput) based on the corresponding read / write address channel “AxVALID” signal.

i. AXI interface component will assert the write data channel “WREADY” signal 1 clock after command put, and start returning data put (p_dput) based on the corresponding write data channel “WVALID” signal if the command indicates write data is associated.

3. The AXI interface will present downstream read completion command put (rdcp_cput) when the readresponse is returned.

a. AXI interface component will always assert read data channel “RREADY” signal, and based on the corresponding read data channel “RVALID” signal, it evaluates / stores all the read data in a

streaming data FIFO to decide the consolidated completion status; only then generate the read completion command put (rdcp_cput), followed by the data put (rdcp_dput) from the streaming FIFO.

4. The AXI interface will present upstream completion when there is command available from the upstream completion FIFO (rdcp_cavail), and pop the command out of the FIFO when the completion get (rdcp_cget) is asserted.

a. If the command indicates write data is associated, AXI interface component will assert command get (cp_cget) on the first assertion of the upstream completion FIFO avail (cp_cavail), only then assert read data channel "RVALID" signal 1 clock later, and continuously popping out the data from the upstream data FIFO for that transaction to AXI data bus, provided the corresponding read data channel "RREADY" signal is also asserted continuously. Given there is no data get signal to the fifo bfm, if "RREADY" signal is de-asserted anytime when the "RVALID" signal is asserted for that transaction, the continuous data popping out will need to be re-directed to a streaming data FIFO instead, and replay from this streaming data FIFO to the AXI data bus when the "RREADY" signal is asserted later.

b. AXI interface component will always assert write response channel "BREADY" signal, and return write completion put (wrcp_cput) based on the corresponding write response channel "BVALID" signal.

c. The AXI interface agent will need to add write completion support so that it can fence any status update that depending on upstream write until the corresponding write response / completion has been received. Need to be able to pipeline multiple upstream write outstanding before receiving the first write completion.

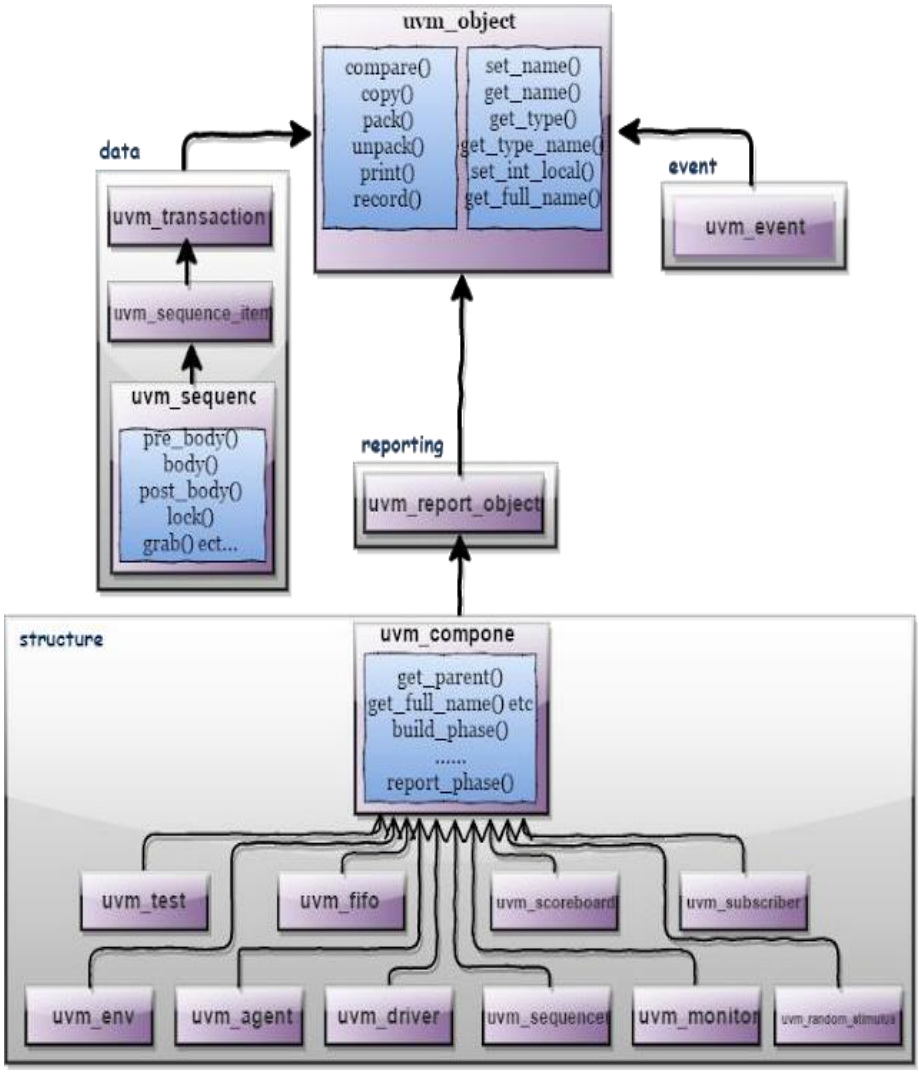


Fig. 1.2 UVM class hierarchy

CHAPTER 2

AXI INTERFACE TESTBENCH ENVIRONMENT

Below is testbench verification environment.

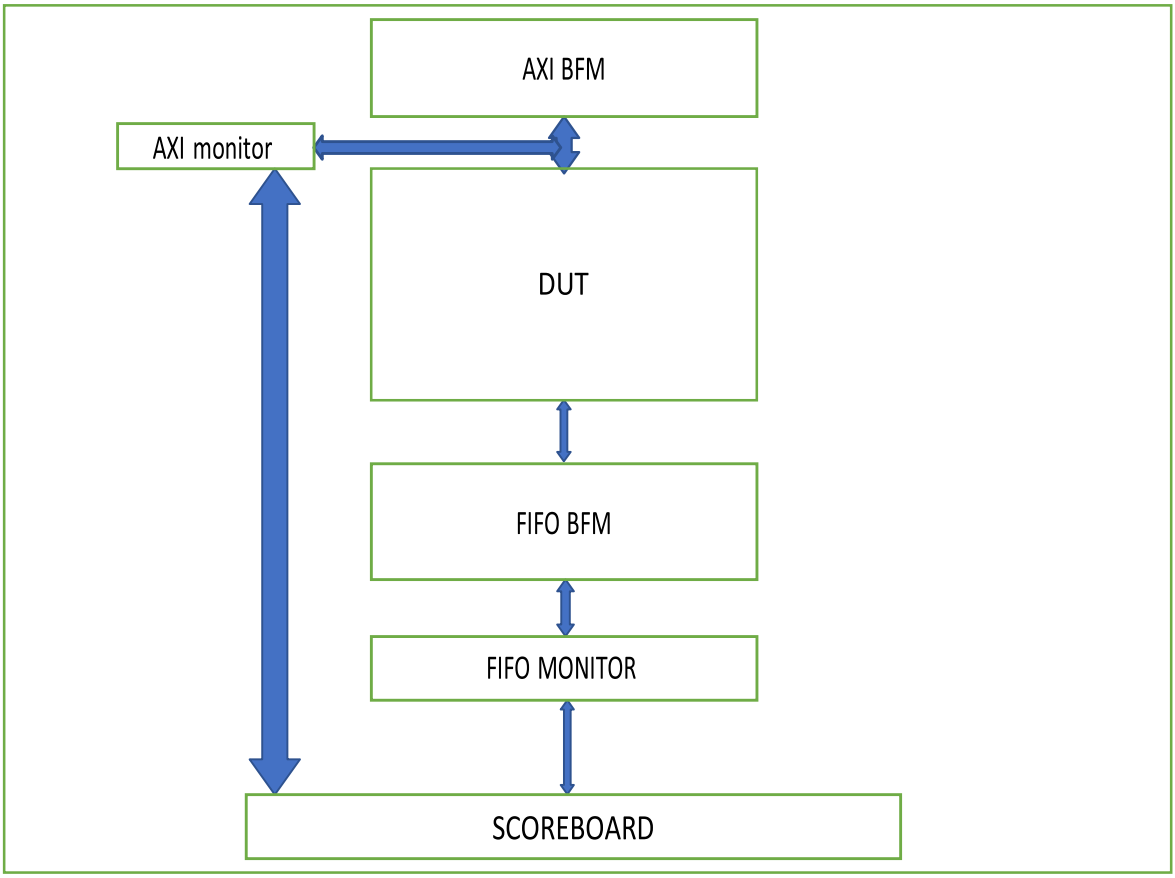


Fig. 2.1 Verification environment

2.1 TEST CLASS

In our testbench verification environment there are several test classes which are specific to the test casesused to verify the various scenarios.

Inside the test class in a fork join loop inside run phase we are connecting sequences and sequencer. Then the sequence items are randomized according to the constraints defined inside the sequence file instantiated in the test class.

The verification environment is starts to execute when the run_test("my_test") method is called, this method is defined in a top module. To run particular test case we need to pass that particular test case name as argument while calling run_test.

2.2 FIFO BFM ENVIRONMENT

FIFO environment includes fifo sequence_item class ,fifo_sequencer class, driver class and monitor class.

2.3 FIFO SEQUENCE ITEM

This class is responsible for generating stimulus and for the randomization of the signals. Constraints can also be defined according to the test case need.

2.3.1 FIFO BFM DRIVER

In FIFO Driver below logic is being used.

In run_phase () reset and drive_data task are being called.

In reset task reset of all signals is being done.

In Drive data below logic is used. Here Fork Join_any is being used in forever loop.

In run phase drive data logic is used , so during run phase driver call get data to get input signals from the sequence item through sequencer. Now after getting signal values driver logic checks for some conditions inside forever loop then according to the sequence values driver decides whether it is a upstream transaction or downstream transaction.

1. From sequence we can set values of sequence item variables like wr_cavail to 1 tostart upstream write transaction or start_rd_trans to 1 to start upstream read transaction.
2. If from sequence reset variable in sequence item is set to 1 then reset begin end loop comes into effect and reset is passed from driver to interface.
3. Interface is set inside top module using config db and according to need interface can be accessedby using get command inside class and components, by providing proper context and path

hierarchy while using set and get commands.

4. Driver is responsible for the conversion of signal level transaction to the pin level transaction.

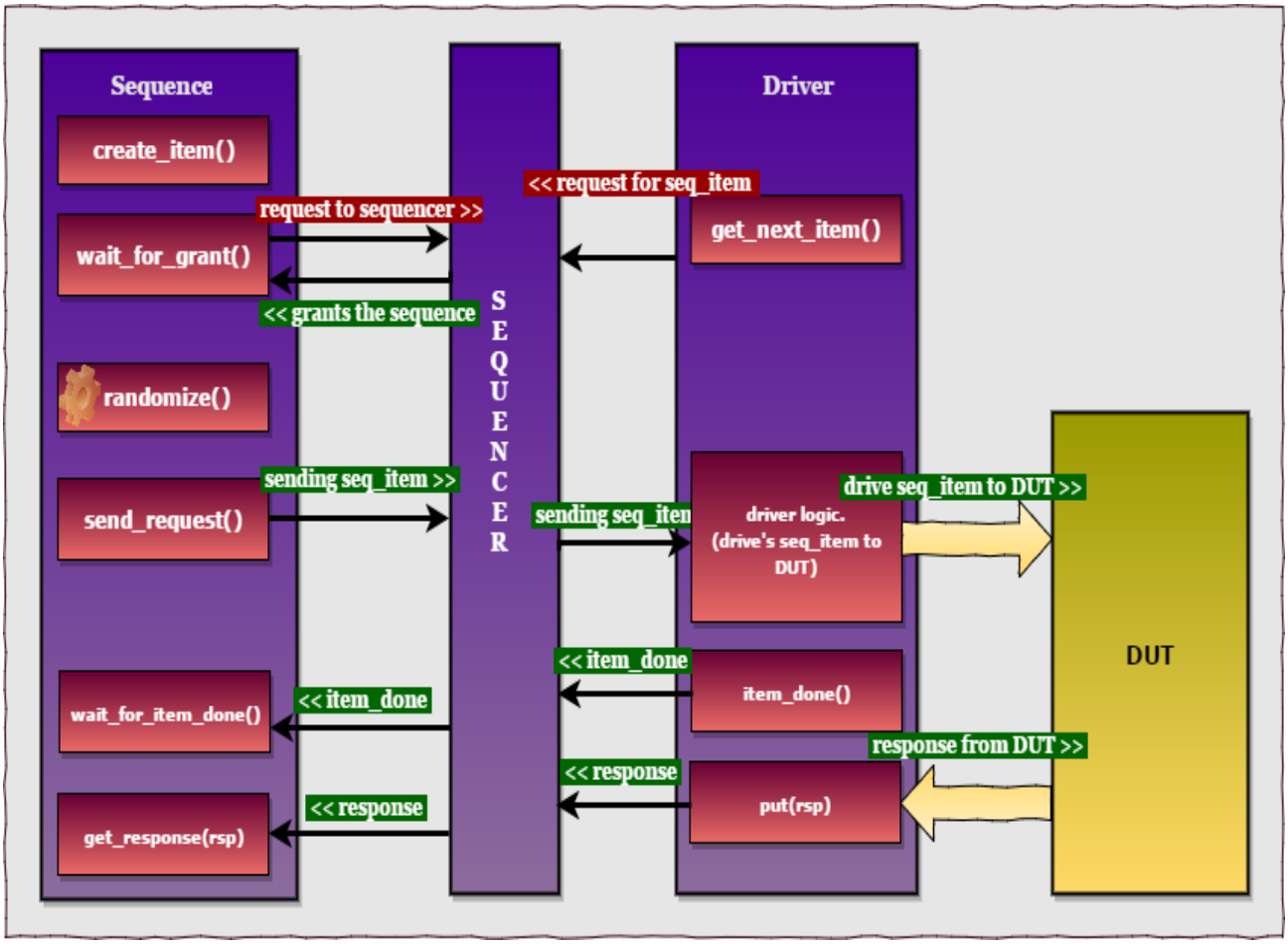


Fig. 2.2 FIFO bfm driver

2.3.2 FIFO SEQUENCE

FIFO sequence generates the stimulus for the test cases defined. In our testbench we are generating stimulus for upstream and downstream transactions. Other than this we have to check different scenarios like reading and writing data to and from a memory or vice versa, parallel upstream and downstream transactions, multiple read write, so in these scenarios we need to put constraints on the signals inside thesequence class. For all these different test cases we made different sequence files which generates the signals according to the test. Sequence executes when we connect the sequence and sequencer inside testcase.

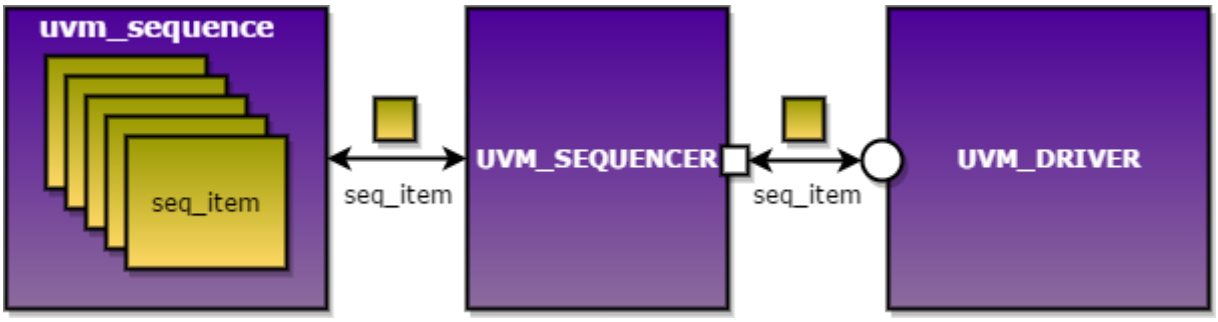


Fig. 2.3 FIFO sequence

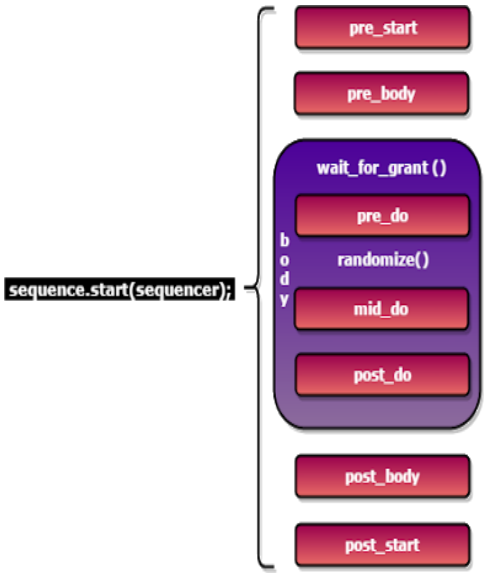


Fig. 2.4 UVM sequence methods

Inside the body method of sequence, logic of generation and sending sequence item is written.

Steps used for the communication between sequence and driver,

1. create item
2. wait for grant(optional).
3. randomization.
4. send request.

2.4 FIFO SEQUENCER

FIFO sequencer send the data generated by the sequence to the driver. Communication between driver and sequencer is done using TLM interface. Ports and exports are defined in driver and

sequencer respectively for the transaction.

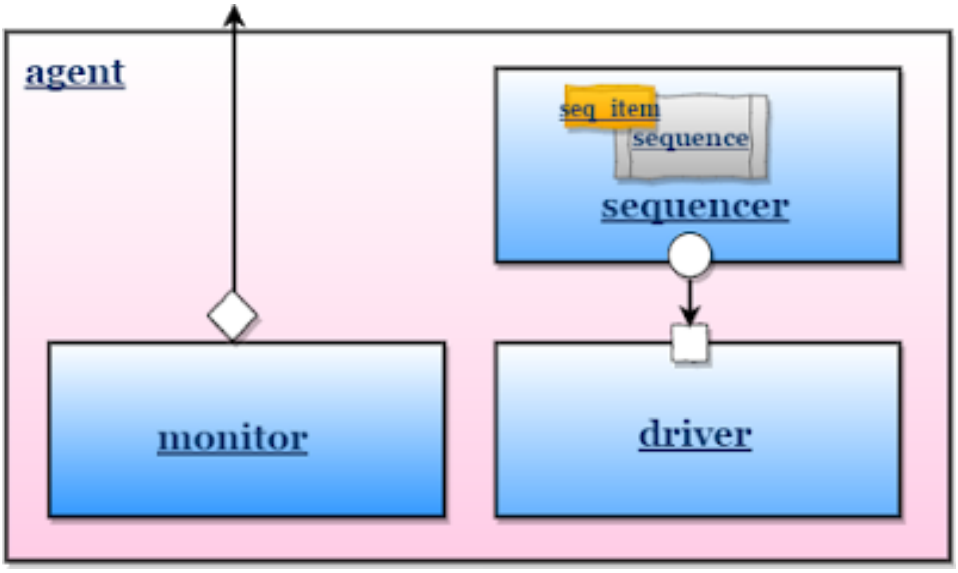


Fig 2.5 FIFO sequencer

2.5 FIFO MONITOR

Interface is set inside the top module using config db. Monitor access the data from the DUT with the help of virtual interface through get config db. Monitor is responsible for the conversion of signal level orpin level activity to transaction level. Monitor uses analysis port and virtual interface handle that pointsto the DUT signals. Monitor in our environment collecting data and send that data to scoreboard and also inside monitor class data is sampled for the functional coverage.

2.6 TESTBENCH SCOREBOARD

In the scoreboard we receives the values as fifo bfm monitor and AXI response. Monitoring scoreboard communication typically involves the use of various technologies and methods to track and receive real-time updates from the scoreboard. Here's a general overview of how it happens:

1. Scoreboard Display: The scoreboard at a sports event typically displays information such as

scores, game time, team names, and sometimes additional statistics like fouls, timeouts, or player stats. This information is usually displayed electronically using LED panels or similar display technologies.

2. Data Collection: To monitor scoreboard communication, data needs to be collected from the scoreboard. This can be done through different means:

a. Scoreboard API: Some modern scoreboards provide an Application Programming Interface (API) that allows authorized users or developers to access the scoreboard data directly. Through the API, users can retrieve real-time updates and game information programmatically.

b. Data Feeds: Scoreboard providers or sports organizations may offer data feeds that deliver the scoreboard information in a standardized format. These data feeds can be accessed by subscribing to their services and receiving regular updates via the internet.

c. Manual Input: In some cases, individuals manually observe the scoreboard and enter the information into a monitoring system. This method is less common for real-time monitoring but may be used for smaller events or when automated options are not available.

3. Data Processing: Once the scoreboard data is collected, it needs to be processed to extract relevant information. This may involve parsing the data feed, converting it into a usable format, and storing it in a database or memory for further analysis.

4. Communication and Display: The processed scoreboard data is then communicated to the end-users or displayed on various platforms. This can be done through websites, mobile apps, social media, or any other medium that allows users to access the live scores and game information.

5. Real-Time Updates: To ensure up-to-date information, the monitoring system continuously fetches and updates the scoreboard data at regular intervals or in real-time. This could be achieved through automated processes that periodically retrieve new data or through push notifications sent by the scoreboard provider. It's important to note that the specific implementation of scoreboard monitoring can vary depending on the event, the scoreboard technology used, and the available resources or platforms for data retrieval and communication. Inside the scoreboard two queues are created to store the values getting from fifo bfm monitor andAXI response. If condition is used to

drive the compare logic but comparison is done based on triggering of anevent which is triggered only when both queues get data. Communication between monitor and scoreboard is done by TLM analysis portScoreboard compares the values, Values received from fifo monitor and Values received as AXI response

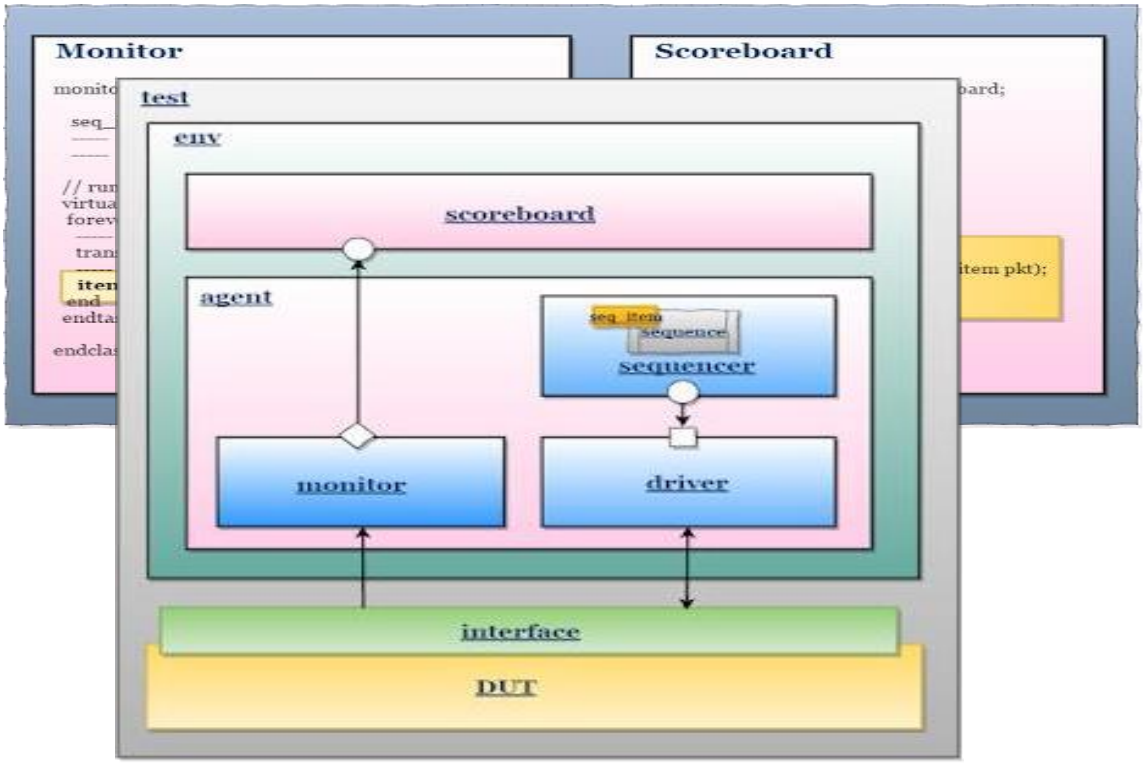


Fig. 2.6 Monitor scoreboard communication

Fig. 2.7 Testbench environment

CHAPTER 3

ENHANCEMENTS

In existing test cases, some of the functionalities of the design are not covered. So enhancements needs to be done in testbench ,which are given below :

1. AXI BFM Functional Coverage is not enabled. There is no local coverage present as well (for AXI interface). This is a coverage gap and it should have the AXI interface coverage as well to ensure we are sending all the possible stimulus as per DUT specifications.

a. Analyze Existing Coverage for AXI side.

b. Enable AXI coverage on AXI interface.

2. There are delays all over the place in the test stimulus. These delays needs to be removed to check if back to back request scenarios are properly covered.

3. There are no delays modeled in the handshake signals on the FIFO interface. All the handshakes are honored immediately. Actual RTL might behave differently.

4. Multiple transactions on a single interface are not enabled. The DUT has the capability of issuing multiple transactions before the previous transactions are completed(upstream). In the current testbench, the transactions are serialized and sent one after another.

5. For Downstream Transactions, there is no pipelining realized. Downstream transactions are put on the interface one at a time when all the phases of a previous transaction are over. In an actual scenario, the agent may issue another transaction while the current transaction is in progress. The DUT has ways to backpressure this second transaction until the processing of the first transaction is over.

6. rd_cavail, wr_cavail are always set to 1. In the actual RTL, they would be toggling and when a downstream transaction arrives, the npfree might be 0 and at a later time, it might toggle to 1 to allow the transaction.

These kind of scenarios are not exercised. Similarly, a lot of other signals are always constrained to a fixed values in the tests. There are tests which might use another values, but they are very minimal.

7. The randomization done on the traffic seems to be minimal.
8. Lot of places in the code, Testplusargs WRITE_DOWN and READ_DOWN command line argument is used even for the building/connecting the components and driving the transactions on the DUT interface. For Downstream read and write transactions, arguments are used to start the transaction (driver code to get the next transaction and drive it has these arguments used). Generally, the construction/connection of the testbench components and agent specific tasks do not depend on the test specific simulation arguments.

Delay removal enhancement

In an existing testbench, there are several test cases that involve multiple transactions occurring simultaneously or sequentially. These transactions include both upstream and downstream read/write operations. It is essential to test the performance of a design after achieving good functional coverage or covering all the functionalities and corner cases. The performance of a design depends on how efficiently and quickly it can handle transactions back-to-back or in parallel. However, the current approach in the testbench involves adding a delay after each transaction before initiating the next one. This delay-based approach is not an ideal method to test a design's performance.

To test a design's performance accurately, it is essential to simulate realistic traffic scenarios that involve multiple transactions happening concurrently without any delay. Such scenarios can help identify any potential bottlenecks and issues in the design, which may not be apparent during functional testing. Additionally, it is essential to test the design with a higher workload to determine its maximum throughput and scalability.

In conclusion, the existing testbench's approach of adding delays after each transaction may not provide accurate performance testing results. To test a design's performance accurately, it is essential to simulate realistic traffic scenarios involving multiple transactions happening concurrently without any delay and test the design with a higher workload to determine its maximum throughput and scalability.

Another critical aspect of performance testing is to determine the design's latency and response time. Latency refers to the time it takes for a request to reach the design and for the design to

respond, while response time refers to the time it takes for the design to complete a transaction after receiving a request. Both latency and response time can significantly impact a design's overall performance and user experience. Therefore, it is essential to measure and analyze these metrics during performance testing.

In addition to latency and response time, it is also crucial to monitor the design's resource utilization during performance testing. The resource utilization can help identify if the design is efficiently utilizing its available resources, including CPU, memory, and I/O. If a design is underutilizing its resources, it may indicate an opportunity to optimize the design further. On the other hand, if a design is overutilizing its resources, it may indicate potential performance and scalability issues. Therefore, monitoring the design's resource utilization during performance testing can provide valuable insights into the design's efficiency and scalability. To address the performance issue in the existing testbench, the delays after each transaction were removed. However, upon removing the delays, it was discovered that almost half of the test cases were no longer working. This unexpected outcome can be attributed to the fact that some components of the testbench, such as the driver and sequence components, had delays built into their functionality. By removing these delays, the behavior of the testbench was significantly altered, leading to test failures.

To resolve this issue, the testbench components were carefully analyzed to identify which components required a delay to function correctly. The delay values were then adjusted, and the testbench was retested to ensure that all test cases were working as expected. The process of identifying and adjusting the delay values required a significant amount of effort and expertise, as it was critical to strike a balance between achieving good performance and ensuring that all test cases were functioning correctly.

In conclusion, removing delays from a testbench to improve performance can have unintended consequences on the testbench's functionality. Careful analysis and adjustment of delay values may be required to ensure that all test cases are working correctly while achieving good performance.

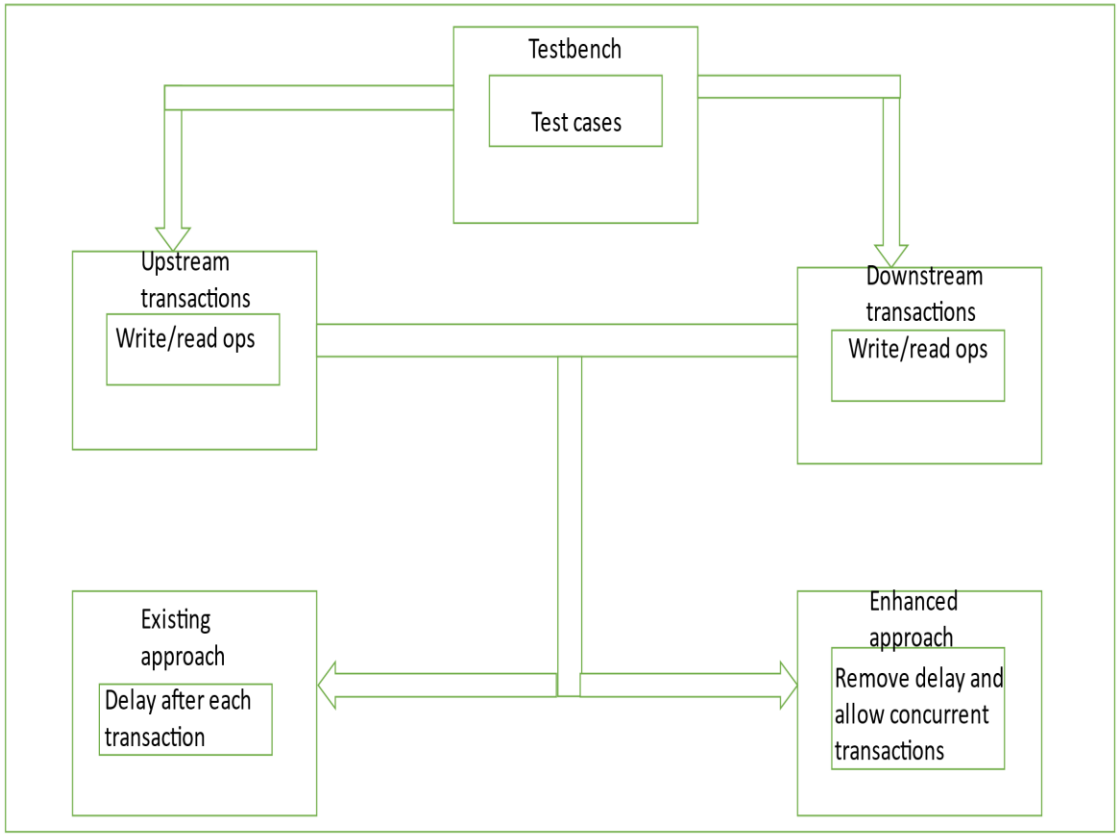


Fig 3.1 approach to allow concurrent transactions

As shown in the diagram, the current approach in the testbench involves adding a delay after each transaction before initiating the next one. This delay-based approach is not an ideal method to test a design's performance since it does not simulate real-world scenarios where transactions occur concurrently or back-to-back. The enhanced approach is to remove the delay and allow concurrent transactions to occur. This approach will provide a more realistic simulation of the design's performance and ensure that it can handle multiple transactions efficiently and quickly.

It is important to note that removing the delay and allowing concurrent transactions in the testbench can increase the complexity of the test environment. This complexity can result in issues such as race conditions, deadlocks, or other synchronization problems that may affect the test results. Therefore, it is essential to carefully design and implement the testbench to handle these scenarios and ensure that the results accurately reflect the design's performance. Additionally, it is

recommended to use performance metrics such as throughput, latency, and response time to evaluate the design's performance accurately. By following these best practices, the testbench can provide reliable and accurate results that reflect the design's ability to handle concurrent transactions and deliver optimal performance.

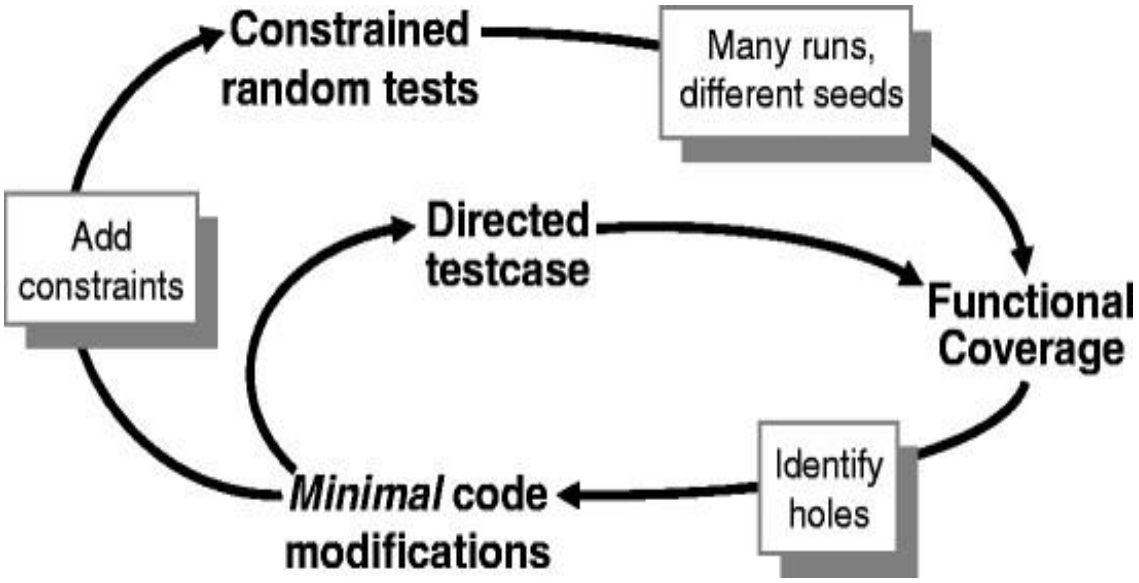
In the context of the design verification process, the test stimulus is a critical component of the testing infrastructure. It generates a sequence of input signals that are applied to the DUT to verify its functionality. In this module, the test stimulus includes delays all over the place, which can impact the accuracy and effectiveness of the verification process. These delays can cause issues such as missed coverage or incorrect functionality. Therefore, it is important to remove these delays to ensure that back-to-back request scenarios are properly covered.

Back-to-back request scenarios refer to a situation where two requests are made in close proximity to each other without any delays between them. This is a common scenario in real-world applications and it is essential that the verification process covers these scenarios thoroughly. However, the delays present in the test stimulus can prevent the back-to-back request scenarios from being properly covered. The delays can cause a delay between the requests, resulting in a gap that is not representative of real-world scenarios. To address this issue, the delays need to be removed from the test stimulus. This can be achieved by modifying the test stimulus code to eliminate the delays. Once the delays are removed, the back-to-back request scenarios can be tested thoroughly to ensure that the DUT behaves as expected. This is essential for verifying the functionality of the DUT in real-world scenarios, as back-to-back requests are a common occurrence in many applications. By ensuring that these scenarios are properly covered, the verification process can provide a high level of confidence in the functionality of the DUT.

Enabling and analyzing functional coverage

Let’s say you have a block you need to verify. How do you know that the stimulus you are about to use is exhaustive enough and that you have covered the necessary scenarios/situations to prove it is working correctly? This is where functional coverage comes in. SystemVerilog’s functional coverage constructs allow you to quantify the completeness of your stimulus by recording the values that have occurred on your signals. It is the consequence of Moore's law and present-day market requirement (low cost products having more functionality and smaller size with better battery performance). In this context, designing a baseband with a lot of features results in very complex and sophisticated design, which can be prone to hidden bugs. Before going for the silicon process, one should complete the verification. So that the outcome product can be reliable and bug free. To ensure this, the verification and development goes hand-in- hand. In verification the Functional Coverage tells how much of the design specification(scenarios) has been exercised.

Fig. 3.2 Coverage Convergence



This type of verification perceives the design from a system or user point of view. Above figure shows the feedback loop to analyze the coverage results and decide on which actions to take in order to converge on 100% coverage. The first choice is to run existing tests with more seeds; the second is to build new constraints. Only resort to creating directed tests if absolutely necessary.Back when we exclusively wrote directed tests, the verification planning was limited. If the design specification listed 100 features, all you had to do was write 100 tests. Coverage was

implicit in the tests — the “register move” test moved all combinations of registers back and forth. Measuring progress was easy: if you had completed 50 tests, you were halfway done. With CRT(Constrained Random test), we are freed from hand crafting every line of input stimulus, but now we need to write code that tracks the effectiveness of the test with respect to the verification plan. However we are still more productive, as we are working at a higher level of abstraction. We have moved from tweaking individual bits to describing the interesting design states. Reaching for 100% functional coverage forces us to think more about what we want to observe and how we can direct the design into those states.

To gather the coverage data run the same random testbench over and over, simply by changing the random seed to generate new stimulus. Each individual simulation generates a database of functional coverage information, the trail of footprints from the random walk. You can then merge all this information together to measure your overall progress using functional coverage as shown in Figure 1.9. The analysis of coverage data is done to decide how to modify the tests. If the coverage levels are steadily growing, it just needs to run existing tests with new random seeds, or even just run longer tests. If the coverage growth has started to slow, adding additional constraints to generate more “interesting” stimuli. When it becomes constant after some time, some parts of the design are not being exercised, so more tests need to be created. Lastly, when functional coverage values near 100%, bug rate needs to be checked. If bugs are still being found, true coverage for some areas of your design is skipped. Don’t be in too big of a rush to reach 100% coverage, which just shows that bugs are looked in all the usual places. While verifying a design, take many random walks through the stimulus space; this can create many unanticipated combinations.

Functional coverage is a crucial aspect of the Universal Verification Methodology (UVM) that enables the verification engineers to measure the completeness of their verification environment. Functional coverage is a measure of the degree to which the design functionality has been exercised by the test cases during verification. It helps to ensure that the design meets the functional specifications and requirements. Functional coverage is typically implemented as a set of coverage points that represent the key functional aspects of the design. These coverage points can be defined at different levels of abstraction, such as module, block, or system level, depending on the

complexity of the design and the verification goals. It plays a vital role in the verification process. It provides feedback on the effectiveness of the test suite and helps to identify areas of the design that have not been thoroughly tested. By measuring the completeness of the verification, functional coverage helps to increase the confidence in the correctness of the design. Additionally, it helps to prioritize the verification efforts by focusing on the areas that have low coverage, which in turn saves time and resources.

UVM provides a built-in functional coverage mechanism that allows verification engineers to define coverage models and collect coverage data during simulation. The UVM coverage model is a hierarchical structure that consists of coverage groups, coverage points, and coverage bins. Coverage groups represent the different aspects of the design that need to be covered, such as signals, transactions, or functions. Coverage points are the individual coverage items within a group, such as specific signal transitions or function calls. Coverage bins are the data containers that collect the coverage data for each coverage point. The UVM coverage model is defined in the coverage class, which is derived from the uvm_coverage_base class. The coverage class provides methods for defining coverage groups, coverage points, and coverage bins, as well as for collecting coverage data during simulation. The coverage data can be analyzed and reported using different tools, such as the UVM built-in coverage report, third-party coverage tools, or custom scripts.

In order to use the UVM coverage mechanism, the verification engineer needs to create a coverage model that reflects the design functionality and the verification goals. The coverage model should be defined in a separate file, which is then connected to the testbench using the uvm_config_db mechanism. During simulation, the coverage model is activated using the uvm_report_coverage() method, which enables the collection of coverage data. The coverage data is then analyzed and reported at the end of the simulation using the uvm_coverage_report class. Functional coverage includes :

1. Statement Coverage

Statement coverage is the most basic coverage type that checks whether every statement in the code has been executed at least once. This coverage type is usually used at the module or block level to ensure that all the statements in a design have been exercised.

2. Branch Coverage

Branch coverage checks whether all the possible branches in the code have been executed at least once. This coverage type is useful in identifying untested paths in a design that may lead to unexpected behavior.

3. Condition Coverage

Condition coverage checks whether all the possible Boolean conditions in the code have been exercised at least once. This coverage type is used to ensure that all the decision points in a design have been tested.

4. Toggle Coverage

Toggle coverage is used to ensure that every bit in the design has been toggled at least once. This coverage type is essential in designs where certain bits may be unused and may not be exercised during normal operation.

5. Finite State Machine (FSM) Coverage

FSM coverage is used to ensure that all the states and transitions of an FSM have been exercised at least once. This coverage type is used in designs that contain state machines to ensure that all the possible sequences of states and transitions have been tested.

6. Assertion Coverage

Assertion coverage checks whether all the assertions in the design have been evaluated as true or false during simulation. This coverage type is used to ensure that all the assertions in a design have been tested.

7. Code Coverage

Code coverage combines statement, branch, and condition coverage to ensure that all the code in a design has been executed at least once. This coverage type is used to ensure that all the executable code in a design has been tested.

This project involves two Bus Functional Models (BFMs), namely, the FIFO BFM and the AXI BFM. To enable functional coverage of the AXI interface, several coverage-related signals need to be set, such as toggle coverage and FSM coverage. Meanwhile, for the functional coverage of the FIFO interface, the signals are sampled inside the FIFO monitor using a task from the interface. The collected data is then sent to a functional coverage class, which is included inside the FIFO BFM. This functional coverage class includes all the signals for which coverage is required. The AXI interface requires toggle coverage and FSM coverage to be set. Toggle coverage is a coverage

type that ensures that every bit in the design has been toggled at least once. This coverage type is essential in designs where certain bits may be unused and may not be exercised during normal operation. FSM coverage is used to ensure that all the states and transitions of a finite state machine (FSM) have been exercised at least once. This coverage type is used in designs that contain state machines to ensure that all the possible sequences of states and transitions have been tested.

On the other hand, for the functional coverage of the FIFO interface, the signals are sampled inside the FIFO monitor using a task from the interface. The collected data is then sent to a functional coverage class, which is included inside the FIFO BFM. This functional coverage class includes all the signals for which coverage is required. Inside the functional coverage class, different cover groups are defined which include coverpoints for all the signals. Inside the coverpoints, bins, cross, transition, and toggle coverage points are present. These coverpoints capture the various aspects of the signal behavior during simulation, such as the number of times a particular signal value occurs, the relationship between two or more signals, the transition of a signal from one value to another, and the number of times each bit in the signal has changed its value. The bins capture the number of times a particular signal value occurs during simulation. Cross coverage captures the relationship between two or more signals, while transition coverage captures the transition of a signal from one value to another. Toggle coverage captures the number of times each bit in the signal has changed its value. To ensure proper functional coverage and to check every corner case some points to be followed:

1. Identify all the design features: Review the design specifications to identify all the interfaces, inputs, outputs, and other important features.
2. Define coverpoints and cross coverage: Define coverpoints for each signal in the design and define cross coverage to capture the relationships between signals.
3. Define bins and sampling intervals: Define bins to capture the different values that a signal can take, and determine how frequently the signal is sampled during simulation.
4. Use randomization and constraint randomization: Generate random scenarios during simulation by using randomization and constraint randomization.
5. Use scenario-based testing: Define a set of scenarios based on real-world use cases that the design should be able to handle.

6. Use code coverage tools: Use code coverage tools to ensure that all the code in the design has been exercised during simulation.

Methodology Work done

This section explains about the methodology followed in the course of the development of this project. It has been divided into AXI Protocol understanding and functional coverage test planning, Functional Coverage coding and compile clean. Simulation and checking mechanism both for Single flow test and Multi test regression followed by Results and Future work .

1) **AXI Protocol understanding and functional coverage test planning** :

First a basic understanding of AXI protocol is required to further go ahead in this project. Handshaking was to be understood clearly so that source and destination are in sync. After gaining understanding of AXI functional coverage test planning was done which included deciding on the functional coverage region as shown in figure 3.3. Here the region between bridge and fabric is chosen. The red dots here represent the ports which we are tapping for functional coverage.

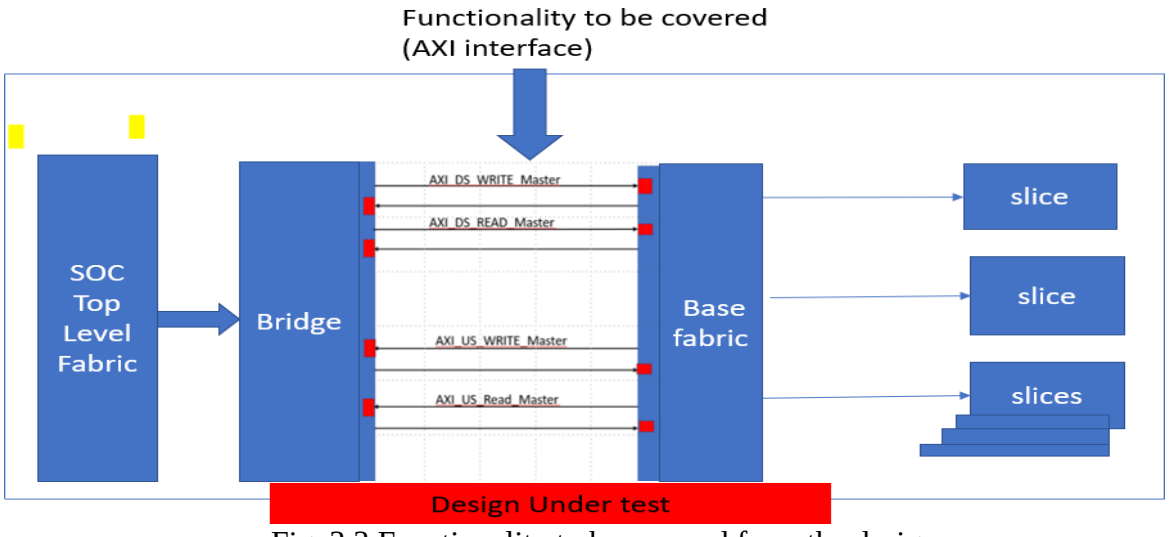


Fig. 3.3 Functionality to be covered from the design

Due to the fact that it accepts transactions from the generator and sends them to the AXI interface, BFM is crucial in the verification environment. The AXI interface that connects the master and the slave is in charge of ensuring that the necessary handshaking occurs during communication [4]. The AXI interface will be used for all outputs as it contains all transaction-related signals. The

signals handshaking and valid transactions are continuously monitored. The monitor is made in accordance with the fact that memory is where most transactions are concentrated.

Based on this tapping four covergroups are decided to be created which carries the downstream and upstream transaction. For downstream two covergroups which cover the transaction from the bridge(master) to fabric(slave) and are named as master2slave_ds sampled on the basis of slave clock slave2master_rsp_read_ds sampled on the basis of fabric clock. Similarly for upstream two covergroups which cover the transaction from the fabric(slave) to bridge(master) and are named as slave2master_us sampled on the basis of fabric clock and master2slave_rsp_read_us sampled on the basis of bridge clock. Post this coverpoints and bins for each covergroup are decided based on the test plan and the input of the design team and collected.

```
covergroup master2slave_ds @(posedge clk_at_fabric);
ds_awaddr: coverpoint ds_awaddr{
                        bins A=   {'h80000000xxxxx};
                        bins B=   {'h80000000xxxxx};
                        bins C=   {'h80000000xxxxx};
                        bins D=   {'h80000000xxxxx};
                        bins E=   {'h80000000xxxxx};
                        bins F=   {'h80000000xxxxx};
                        bins 0 =  {'h80000000xxxxx};
                        bins 1 =  {'h80000000xxxxx};
                        bins 2 =  {'h80000000xxxxx};
                        bins G =   {'h80000000xxxxx};
                        bins H =   {'h80000000xxxxx};
                        bins I =   {'h80000000xxxxx};
                        bins J =   {'h80000000xxxxx};
                        bins K =   {'h80000000xxxxx};
                        bins I =  {'h800000xxxxxxx};
                        }

 ds_awid:coverpoint ds_awid{
                       bins awid = {x,z};
                       }
ds_awsize:coverpoint ds_awsize{
                         bins awsize_a ={3'hx};
                         bins awsize_b ={3'hx};
                         bins awsize_c ={3'hx};
                         }
                       endgroup:master2slave_ds
```

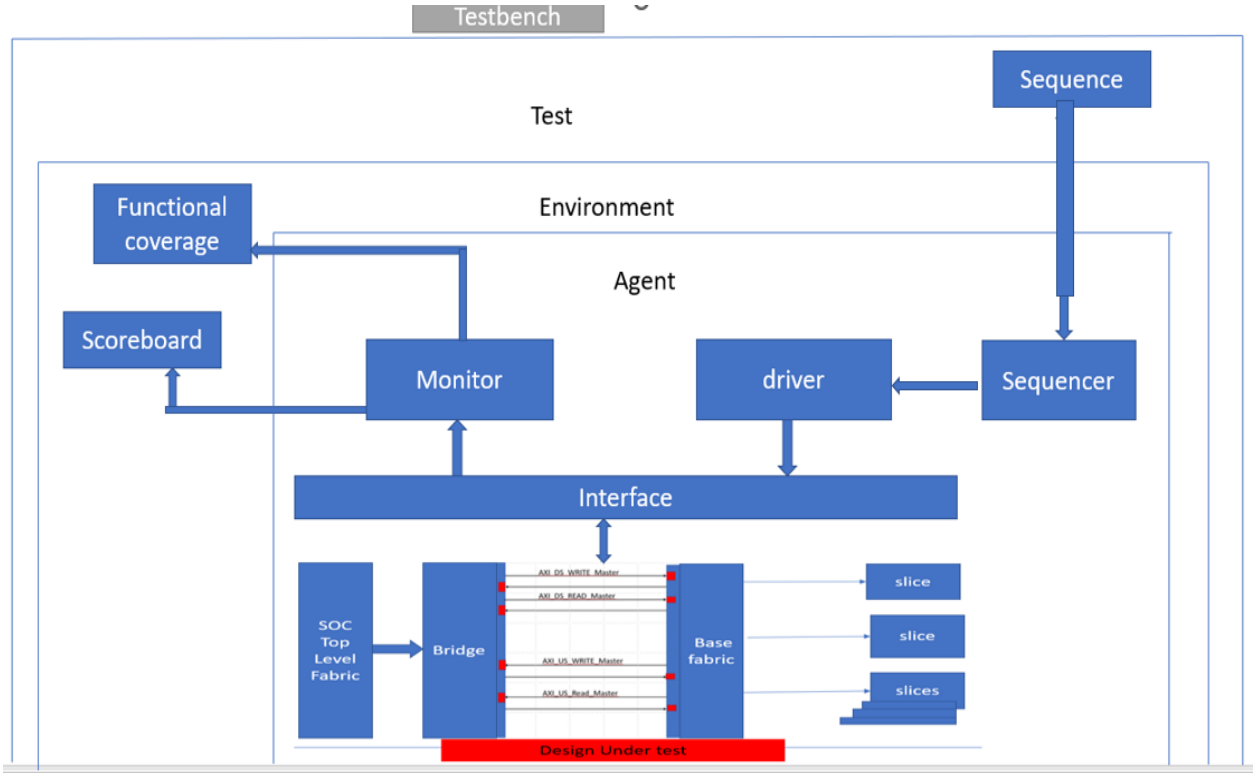
2) Functional Coverage Coding and compile clean:

Fig. 3.4 A snippet of coding of covergroups.

Total of four covergroups are coded with first covergroup 8 coverpoints 86 bins , second covergroup with 6 coverpoints and 84 bins, third covergroup with 9 coverpoints and 200 and fourth covergroup with 6 coverpoint and 200 bins. After coding was done then the code was compiled and checked for errors. Initially some syntax errors were seen such as missing colons, ending statements etc. After minor changes the code was successfully compiled.

3) Simulation and checking mechanism:

Single flow test check : After successful compilation it was required to run the code and see its result. A single test was run on the DUT through a typical UVM environment as shown in figure 3.4 which involves Sequencer which carries the sequences to the driver class. These sequences are then moved to the DUT where they are processed and their result is obtained which is picked

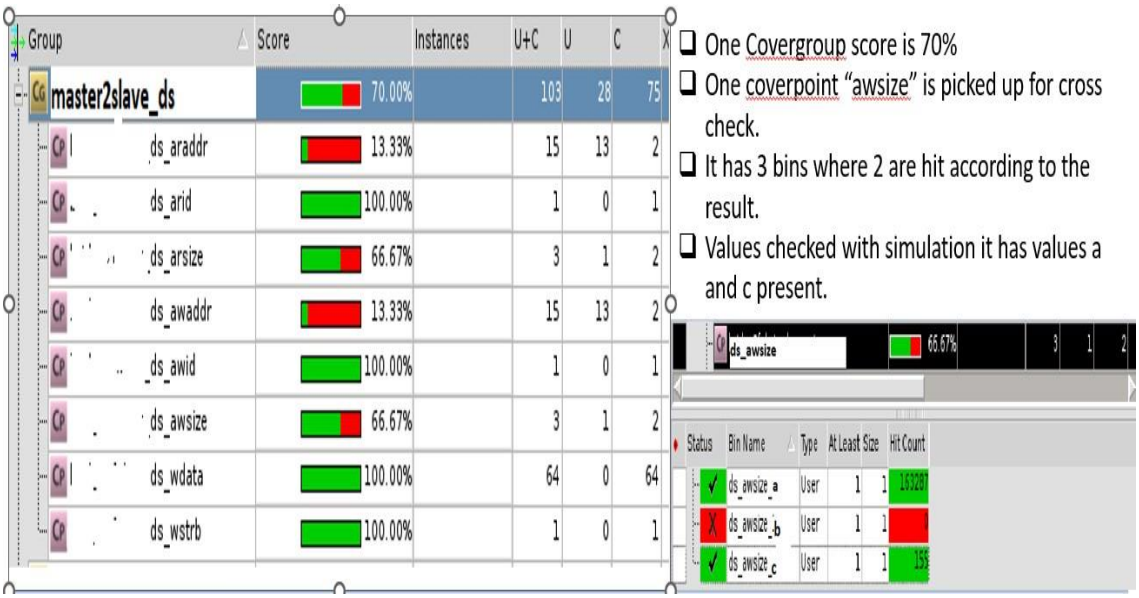


up by the monitor. Now the monitor's job is to transport this information to the scoreboard and functional coverage database. Here functional coverage is the database which collects all the data of bins hit or not % of both functional coverage/coverpoint.

Fig. 3.5 Running of a single test in UVM Environment

Now the results of the single test obtained are checked to check whatever data present in the functional coverage database is the same as data in simulation. As represented in figure 3.5 total coverage data is seen and a particular coverpoint “ds_awsize” is picked up.

Fig. 3.6 Result of single test



Now the value of ds_awsize is cross checked in the simulation which is found to be similar as the bin values hit. Only for those values present in simulation the bins are hit. After this cross checking we came across that a major number of bins are still unhit. To gain more confidence in the code a sanity check to ensure a minimum number of bins are hit.

a) Sanity Check : Before enablement of the coverage code a sanity check is carried out to gain some confidence in the code. Once a minimal number of bins were hit the coverage was enabled for normal regression run. Here minimal number refers to some bins of each coverpoint being targeted such as toggling of bits,some address bins, some bins containing length,etc based on the simulation result. The result of simulation is cross checked with that of bins hit in the coverage. After this confidence only it was enabled for regression run and the data after run was collected effectively as shown in figure 3.7. As most number of bins were found to be hit the code was processed to be used for regression run.

Fig. 3.7 Result of sanity checks

b) Multi test regression: After the sanity check the code was turned in to be used for the run of regression runs . To be sure that some uncovered bins in the sanity are covered. As the main goal of coverage is to reach 100%. Regular regression runs are being carried out so that the coverage improves.

4) Results:

As shown in figure 3.7 regression run of two days the coverage result has shown slight increase in the percentage. It can be further increased by running test cases with multiple seeds. If still no change further analysis will be done on the result obtained some values will be randomly picked and cross checked. Error in the code and coverage plan will try to observe.

S.No	Covergroups	15 December Score	16 December Score
1	master2slave_ds	96.12	96.12
2	master2slave_ds_read_rsp	100	100
3	slave2master_us	82.28	82.8
4	slave2master_us_read_rsp	100	100
	Total	378.4	378.92
	Percentage	94.6	94.73

Fig. 3.8 Regression runs result

No delays modeled in the handshake signals on the FIFO interface

In the current implementation of the FIFO interface, there are no delays modeled in the handshake signals. This means that all the handshakes are immediately honored, without any delay. However, the actual RTL implementation of the interface might behave differently, and may introduce delays in the handshake signals. This can lead to unexpected behavior in the design and can potentially cause issues in the system. To ensure that the simulation accurately models the behavior of the actual RTL implementation, it is important to implement the delays in the handshake signals. This can be achieved by introducing timing models in the simulation, which accurately represent the timing behavior of the actual RTL implementation. This will ensure that the simulation accurately reflects the behavior of the system, and will help to identify any issues that might arise due to timing constraints in the actual implementation.

Overall, it is important to model the behavior of the actual RTL implementation as accurately as possible in the simulation. This will help to ensure that any issues or potential problems are identified and resolved early in the verification process, before they become more difficult and costly to fix. By introducing delays in the handshake signals, we can ensure that the simulation accurately reflects the behavior of the actual implementation , and can help to ensure the reliability and functionality of the system.This module includes an AXI interface, a FIFO interface, and the Design Under Test (DUT). The AXI interface and the FIFO interface communicate with each other through the DUT. In the context of this module, when the AXI BFM initiates transactions and reads or writes data to or from the FIFO BFM, this is referred to as a downstream transaction.

Conversely, when the FIFO BFM initiates transactions and reads or writes data to or from the AXI BFM, this is referred to as an upstream transaction. When a downstream transaction takes place, the AXI BFM initiates the transaction by asserting the appropriate signals. The FIFO BFM then responds according to the read/write data. However, in the current implementation of the module, there is no delay modeled in the handshake signals of the FIFO interface. As a result, the FIFO signals are honored immediately without any clock delay. This can lead to unexpected behavior in the design, as the actual RTL implementation of the FIFO interface may behave differently.
To address this issue, two clock delays are added inside the FIFO driver to introduce a delay in the handshake signals. This ensures that the simulation accurately models the behavior of the actual RTL implementation, and helps to identify any issues that might arise due to timing constraints in the actual implementation. In summary, the module includes an AXI interface, a FIFO interface, and the DUT. Downstream transactions occur when the AXI BFM initiates transactions and reads or writes data to or from the FIFO BFM. Upstream transactions occur when the FIFO BFM initiates transactions and reads or writes data to or from the AXI BFM. The FIFO signals are currently honored immediately without any clock delay, which can lead to unexpected behavior in the design. To address this issue, two clock delays are added inside the FIFO driver to accurately model the behavior of the actual RTL implementation. By doing so, the simulation accurately reflects the behavior of the system and helps to ensure the reliability and functionality of the design.

Multiple transactions on a single interface

In the context of digital design verification, a testbench is used to apply a series of test stimuli to the Design Under Test (DUT) to ensure its proper functionality. One of the key aspects of a testbench is its ability to simulate the behavior of the real-world application of the DUT. In the current testbench, multiple transactions on a single interface are not enabled, which can affect the accuracy and effectiveness of the verification process. The DUT has the capability of issuing multiple transactions before the previous transactions are completed (upstream), but in the current testbench, the transactions are serialized and sent one after another. This can cause issues such as missed coverage or incorrect functionality. To address this issue, multiple transactions on a single

interface need to be enabled in the testbench. This can be achieved by modifying the testbench code to allow multiple transactions to be issued on a single interface without waiting for the previous transaction to complete. This modification can be done in a number of ways, including using multiple threads or using a pipelining approach.

One possible approach to enable multiple transactions is to use multiple threads. This approach involves creating multiple threads in the testbench, each of which is responsible for issuing and monitoring a transaction on a single interface. Each thread operates independently of the others and can issue a new transaction on the same interface as soon as the previous transaction has been initiated. This approach can help to reduce the serialization of transactions and enable the DUT to issue multiple transactions without waiting for the previous transaction to complete. Another possible approach to enable multiple transactions is to use a pipelining approach. This approach involves breaking up the transaction into multiple stages, with each stage being processed independently of the others. The stages can be processed in parallel, enabling multiple transactions to be issued on the same interface without waiting for the previous transaction to complete. This approach can be useful for interfaces that require a high throughput or that have a large number of transactions.

In the run phase of the test case, multiple transactions are issued on a single interface. To simulate the parallel execution of transactions, the sequences are started using the fork join_any construct. The fork join_any construct is a parallel execution construct that starts multiple sequences in parallel and waits for any one of them to complete before continuing. During the simulation, the waveform is checked to verify whether the data is routed correctly or not. The waveform is a graphical representation of the signals in the design and their behavior over time. It allows designers to visualize the behavior of the design and identify any issues that may arise during simulation. The simulation is run for multiple cycles to ensure that the behavior of the system is consistent and correct over time. If any issues are identified during the simulation, the test case is modified to address the issue, and the simulation is run again. Enabling multiple transactions on a single interface in the testbench is important for accurately simulating the behavior of the DUT in real-world applications. This can help to ensure that the DUT behaves as expected when multiple transactions are issued on a single interface. By enabling multiple transactions, the testbench can provide a more accurate and comprehensive verification of the DUT's functionality, which can

increase the confidence in its operation in real-world scenarios.

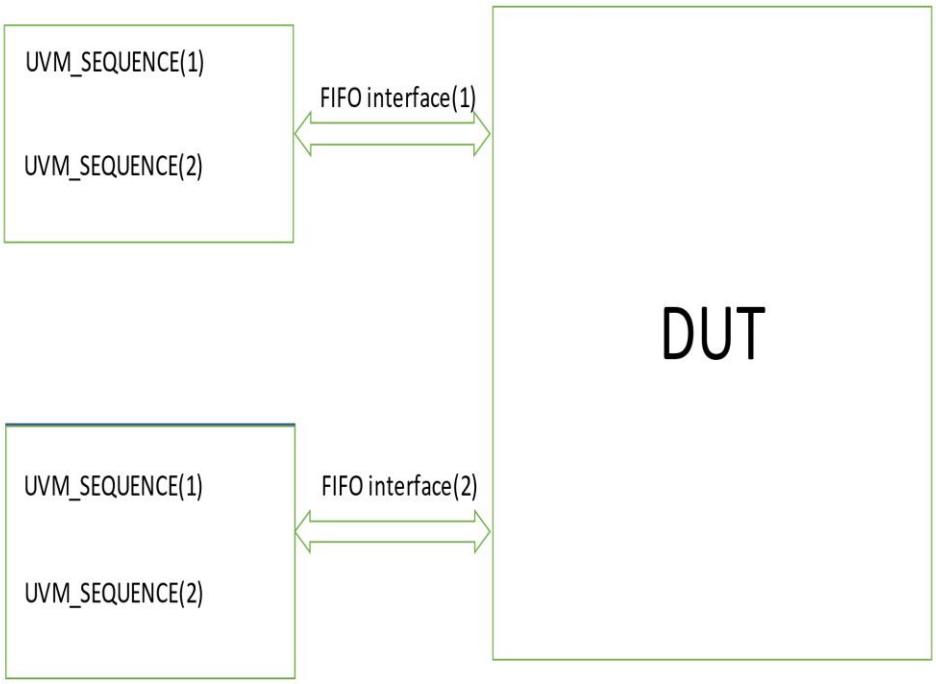


Fig. 3.9 Multiple sequences started on a single interface parallely

Downstream pipelining

In the current testbench for Downstream Transactions, there is no pipelining realized. Downstream transactions are put on the interface one at a time when all the phases of a previous transaction are over. In an actual scenario, the agent may issue another transaction while the current transaction is in progress. The DUT has ways to backpressure this second transaction until the processing of the first transaction is over. So in existing test cases no single test case is there for pipelining. Downstream pipelining is a technique used in computer systems to improve the efficiency of data transfers between components. In downstream pipelining, multiple transactions can be in progress simultaneously, with each transaction being processed by different stages of the pipeline. To implement downstream pipelining, multiple instances of the same sequence can be created, and these instances can be run in parallel using the fork-join_any construct. To cover the downstream pipelining for read and write transactions, two test cases can be added. The first test case can cover the read transactions, while the second test case can cover the write transactions.

In the first test case for read transactions, multiple instances of the same sequence can be created for the read transactions. These instances can be run in parallel using the fork-join_any construct, with each instance representing a stage in the pipeline. Each instance of the sequence can issue a read transaction and wait for the response from the DUT. Once the response is received, the instance can pass the data to the next instance of the sequence, which can process the data further. This process can continue until all the instances of the sequence have completed processing the data.

The second test case can cover the write transactions. In this test case, multiple instances of the same sequence can be created for the write transactions. These instances can be run in parallel using the fork-join_any construct, with each instance representing a stage in the pipeline. Each instance of the sequence can issue a write transaction and wait for the response from the DUT. Once the response is received, the instance can pass the data to the next instance of the sequence, which can process the data further. This process can continue until all the instances of the sequence have completed processing the data.

To cover the writeread case, a combination of the above two test cases can be used. Multiple instances of the sequence can be created to issue write transactions, followed by read transactions. These instances can be run in parallel using the fork-join_any construct, with each instance

representing a stage in the pipeline. The write transactions can be processed in the first set of instances, while the read transactions can be processed in the second set of instances. Once the data has been processed by all the instances of the sequence, the results can be checked to ensure that the data was transferred correctly. In conclusion, to cover downstream pipelining for read and write transactions, multiple instances of the same sequence can be created and run in parallel using the fork-join_any construct. Two test cases can be added to cover the read and write transactions separately, while a combination of these test cases can be used to cover the writeread case. These test cases can help to ensure that data is transferred efficiently and accurately between components in computer systems. This stimulus needs to be modified to cover such scenarios.

Testplusargs READ_DOWN WRITE_DOWN enhancement

A testbench is a set of modules, interfaces, and tasks that are used to verify the functionality of a design under test (DUT). It is a simulation environment that enables the testing of a DUT in a controlled manner. Testbench components include stimulus generators, checkers, monitors, scoreboards, and coverage collectors. One important aspect of testbench development is the use of command-line arguments, such as Testplusargs WRITE_DOWN and READ_DOWN, which are often used in connecting and building testbench components, as well as driving transactions on the DUT interface. However, this approach can lead to issues with code complexity and maintainability. To address this issue, signal-based approaches can be used to drive downstream transactions, while removing test-specific simulation arguments from the construction and connection of testbench components. The Testplusargs command-line arguments are used to specify test-specific parameters, such as the test name or the number of iterations, when running a simulation. In the context of testbench development, Testplusargs WRITE_DOWN and READ_DOWN are used to drive transactions from the driver to the DUT interface. This approach is often used in downstream transactions, where transactions are initiated by the driver and transmitted to the DUT through a specific interface. However, the use of Testplusargs in connecting and building testbench components can lead to code complexity and maintainability issues. This is because the components become dependent on specific simulation arguments, making it difficult to modify or reuse them for different test cases. In addition, Testplusargs may not be suitable for

testing complex systems, where many components and interfaces are involved.

To address these issues, signal-based approaches can be used to drive downstream transactions. This involves using signals that are responsible for downstream transactions as a condition inside the driver to drive transactions to the DUT. This approach removes the dependency on Testplusargs and makes the driver more generic, allowing it to be reused across different test cases. For example, instead of using Testplusargs to specify the interface through which a transaction should be transmitted, the driver can check the signal responsible for the downstream transaction and transmit the transaction through the appropriate interface. This approach reduces code complexity and makes it easier to modify the driver for different test cases. The construction and connection of testbench components, such as the driver, monitor, and configuration, should not depend on the test-specific simulation arguments. These components should be constructed and connected based on their intended functionality, without being affected by the specific test case.

In addition, agent-specific tasks, such as the communication between the driver and the monitor, should also be designed to be independent of the specific test case. This allows the tasks to be reused across different test cases, making the testbench more modular and easier to maintain. In conclusion, the use of Testplusargs in connecting and building testbench components can lead to code complexity and maintainability issues. Signal-based approaches can be used to drive downstream transactions, while removing the dependency on Testplusargs and making the testbench more generic. The construction and connection of testbench components, as well as agent-specific tasks, should be designed to be independent of the specific test case, making the testbench more modular and easier to maintain.

Some signals always set to1

In the world of digital design and verification, transactions play a crucial role in ensuring the proper functioning of a design. Transactions represent a specific unit of data transfer between different modules in a digital system. Transactions typically involve a set of signals that need to be set to specific values to allow the transfer of data. Some of these signals are always set to a specific value, such as wr_sel, rd_sel for downstream transactions, and wr_cavail, rd_cavail.

To verify the functionality of a design, sequences are defined that involve these signals. These

sequences are typically constrained based on the type of transaction, whether it is an upstream or downstream transaction. However, in the actual Register Transfer Level (RTL) implementation, these signals may toggle between 0 and 1, depending on the state of the design. To overcome this issue, signals that are fixed to a value of 1 or 0 inside sequences are toggled to 0 or 1 in the driver after they have been driven. This ensures that the signals are in the correct state when downstream transactions arrive. In this way, the verification process can accurately model the behavior of the design and ensure that it meets the required functionality. It is important to understand the behavior of transactions and the signals involved in them to ensure that the design is functioning correctly. For example -

```
class my_sequence extends uvm_sequence #(my_transaction);

`uvm_object_utils(my_sequence)

function new(string name = "my_sequence");
super.new(name);
endfunction

virtual task body();
my_transaction tr;
// Randomize fields using uvm_rand_send_with
`uvm_rand_send_with(tr, {
tr.field1==1,
tr.field2==0,
tr.field3==1  });
// Additional customizations to the transaction if needed
tr.field5 = "Hello, World!";
// Send the transaction to the sequencer
if (seq_item_port.try_put(tr)) begin
```

```
`uvm_info(get_type_name(), $sformatf("Sent transaction: %s", tr.sprint()),
UVM_MEDIUM)
end else begin
`uvm_error(get_type_name(), "Failed to send transaction")
end
endtask

endclass

class my_driver extends uvm_driver #(my_transaction);

`uvm_component_utils(my_driver)

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

virtual task run_phase(uvm_phase phase);
my_transaction tr;
forever begin
// Wait for a transaction from the sequencer
seq_item_port.get_next_item(tr);
// Drive the signals of the transaction
@(posedge clk) begin
signal1 <= tr.field1;
tr.field2<=1;
signal3 <= tr.field3;
end
// Do additional processing on the transaction if needed
tr.field6 = signal4;
```

```
// Finish the transaction and notify the sequencer
seq_item_port.item_done();
end
endtask

endclass
```

In above example in sequence class the value of field2 signal is set to 0 and to toggle this value , after allowing required transaction it toggled to value 1.

The most important component of the VLSI configuration stream is verification. In order to prevent it from turning out to be dangerous later on in the design process, it aims to find the problems in the RTL (Register Transfer Level) plan early on. During the verification process, distractions occur about 70% of the time. It is the most time-consuming process in this approach. The number of transistors in integrated circuits (ICs) has increased, and this has resulted in a reduction in highlight cost as well as improved outline tools [5]. This increases the chance that the outline will contain bugs. Consequently, the demand for the IC's confirmation became essential.

CHAPTER 4

SIMULATION AND CONCLUSION

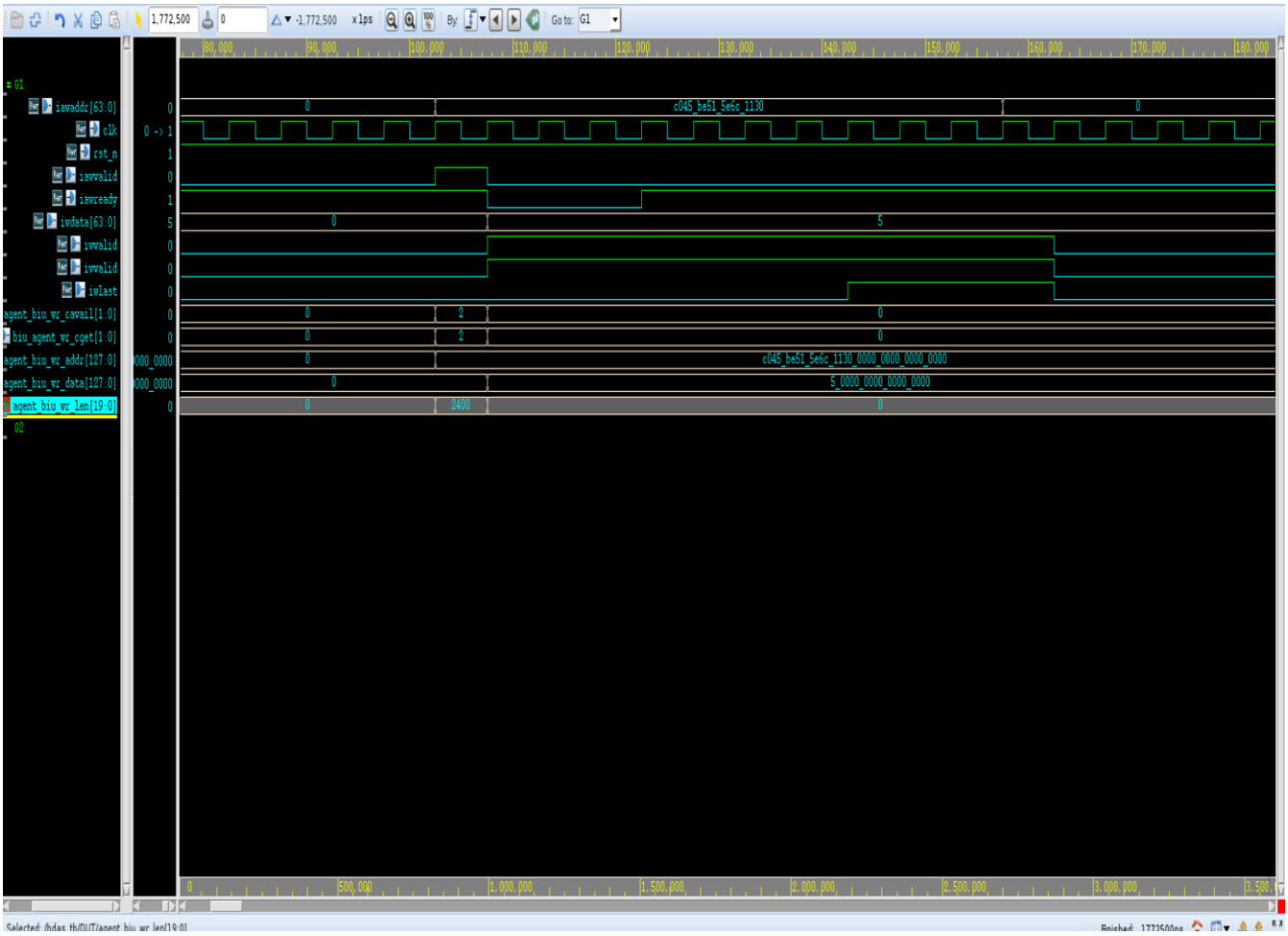


Fig. 4.1 Upstream write

In above waveform as we can see for upstream write , fifo bfm is sending write transaction signals to DUT with the help of valid and ready handshake signals, then after handshake between DUT and AXI bfm occurs when both valid and ready signals asserted and signals are mapped to axi transactions by DUT and data is written into AXI slave memory.

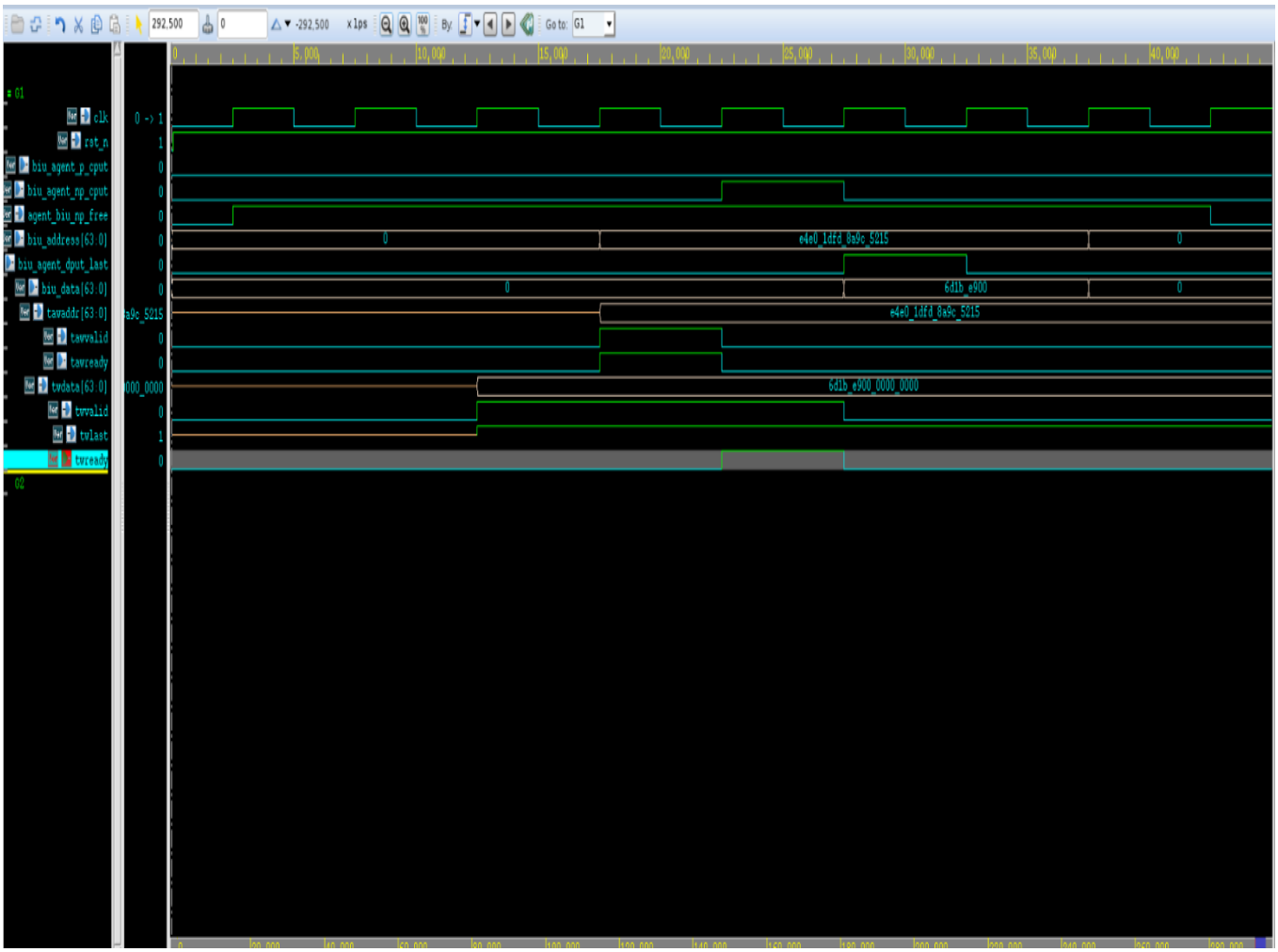


Fig. 4.2 Downstream write

In above waveform AXI bfm send the write transaction to the DUT then FIFO bfm acknowledge the downstream write by sending np_free signal to DUT to start the Transaction in downstream. From DUT fifo BFM will receive dput_last, fifo interface will assign cp_cavail =1 till cp_cget goes high then FIFO BFM will disable cp_cavail.

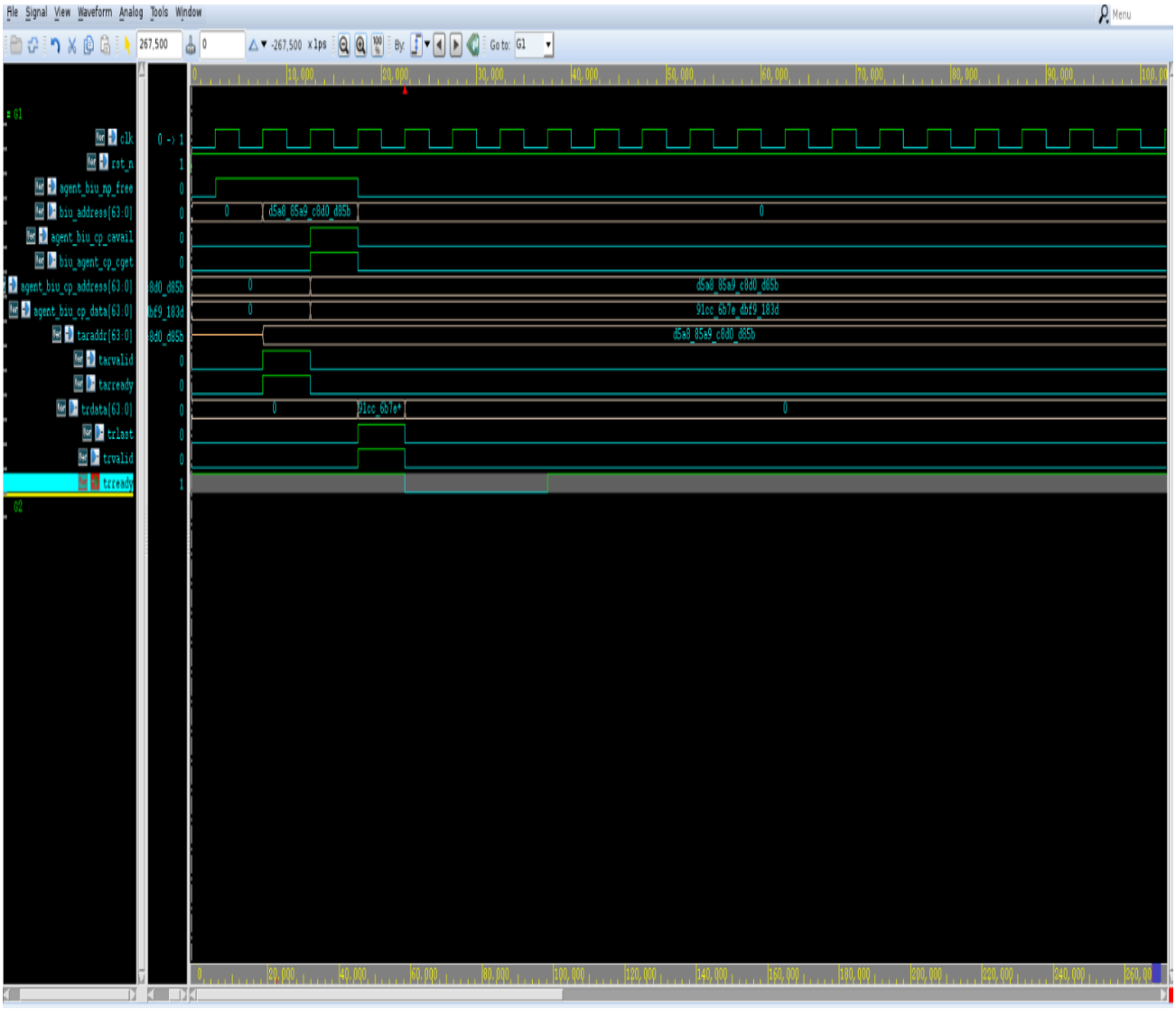


Fig. 4.3 Downstream read

In above waveform AXI bfm send the read transaction to the DUT then FIFO bfm acknowledge the downstream read by sending np_free signal to DUT to start the Transaction in downstream. From DUTfifo BFM will receive np_cput, fifo interface will assign cp_cavail =1 till cp_cget goes high then FIFO BFM will disable cp_cavail.

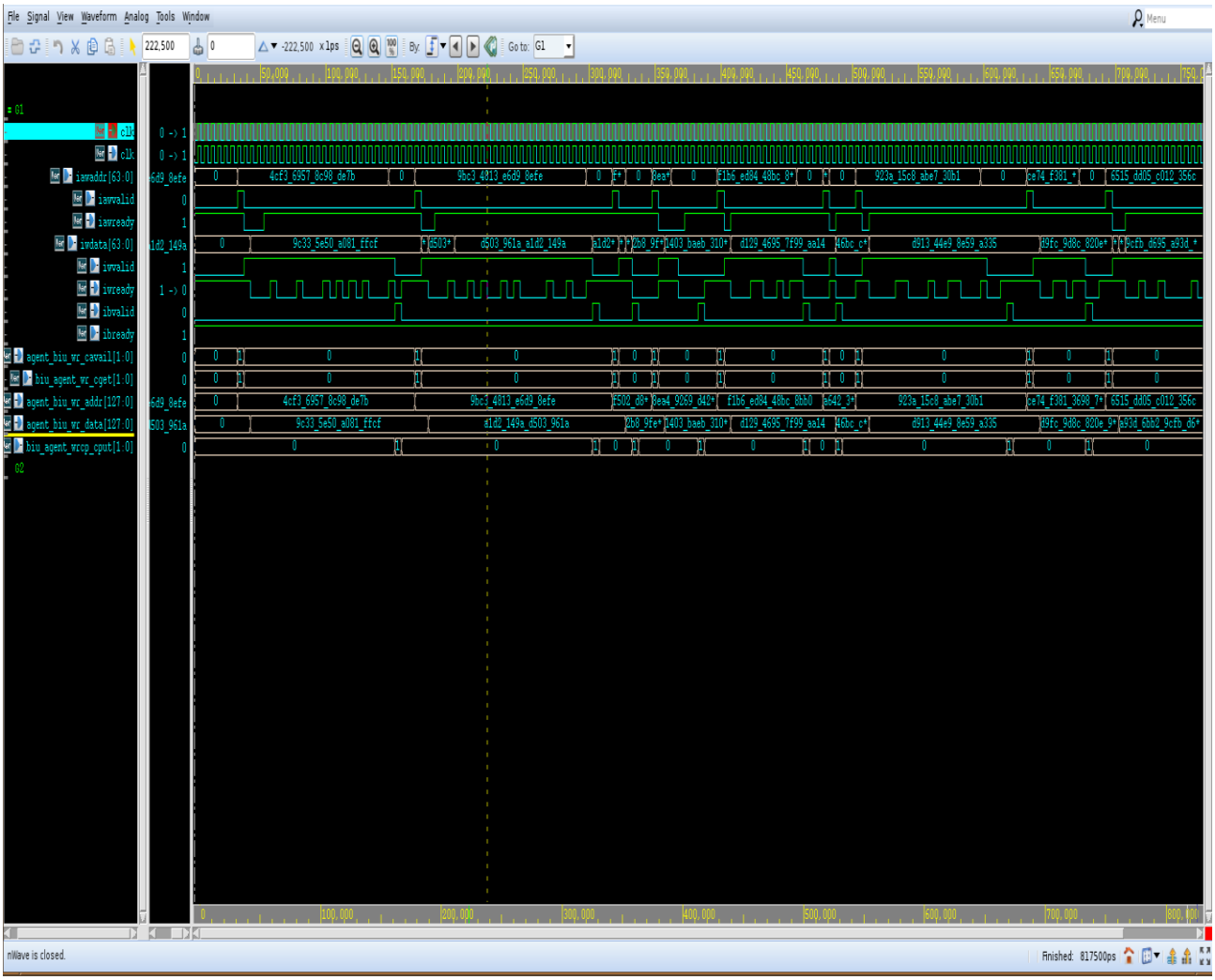


Fig. 4.4 Single interface multiple write

In the above waveform upstream write transaction is taking place. To enable multiple write transactions on a single interface ,transactions are put on interface parallel and parallel transactions are arbitrated by the DUT.

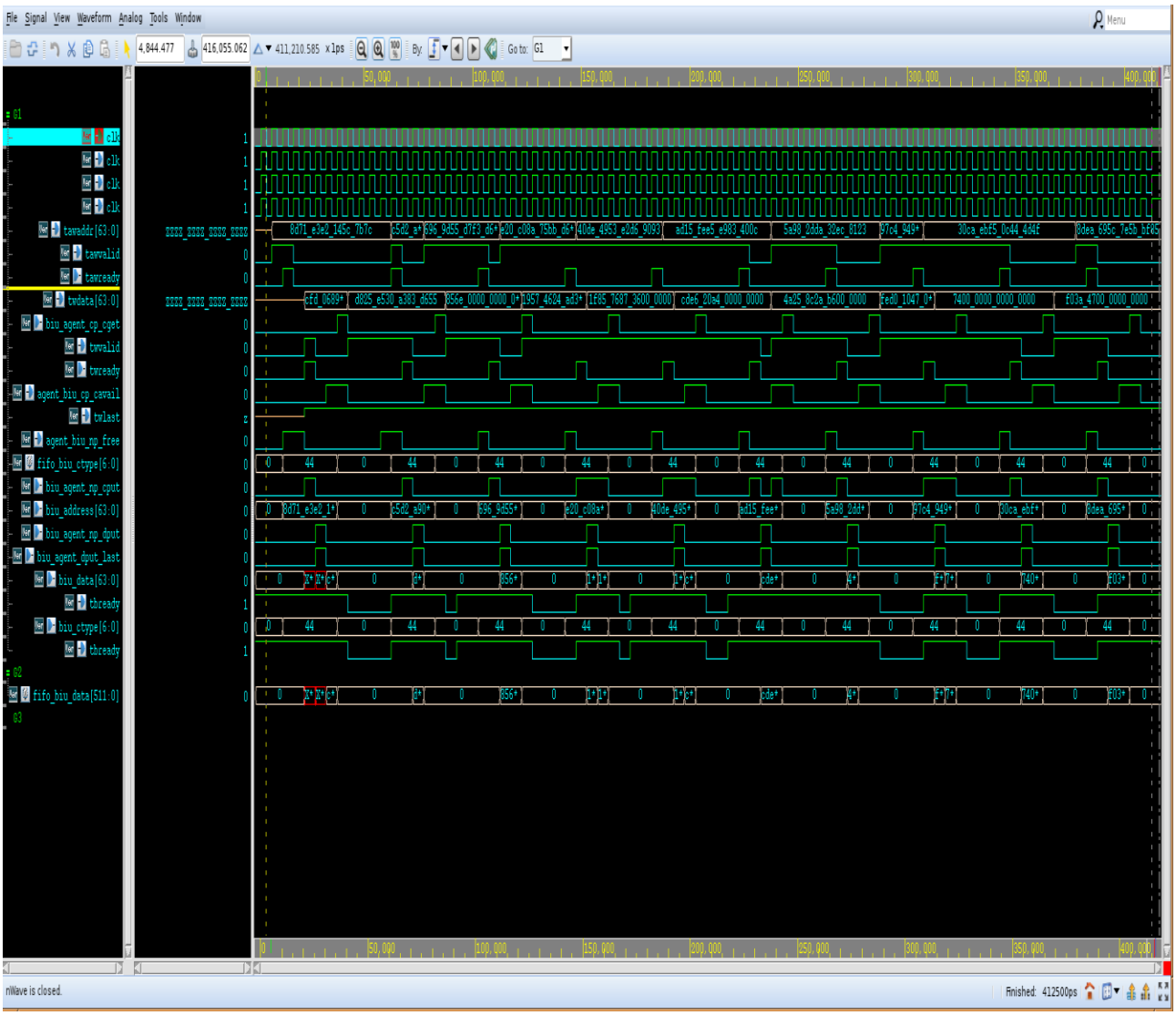


Fig. 4.5 Downstream pipelining write

In the above waveform downstream write transaction is taking place. To enable downstream pipelining sequences started parallel using fork join_any construct and parallel transactions are arbitrated by the DUT. In above waveform it can be seen that, next address is sent by the DUT to FIFO before completing the previous transaction .i.e before sending the cp_cavail ,next address is sent by the DUT.

CONCLUSION

The objective of this work was to implement verification of AXI interface using UVM. The project includes several testcases to check the overall functionality of the design under test. Test cases include upstream and downstream read and write respectively, then written testcase for back to back read and write transaction including write read reset. Further there are some bugs in this design as observed for example upstream read data was always coming zero. Functional coverage will be implemented in other testcases generated to check the overall coverage of the design. And other enhancements were done according to performance and bugs in the testbench to check all other possibilities.

These enhancements include :

1. AXI BFM Functional Coverage was not enabled. There was no local coverage present as well (for AXI interface).
2. There were delays all over the place in the test stimulus. These delays removed and look at the functional coverage to check if back to back request scenarios were properly covered.
3. There were no delays modeled in the handshake signals on the FIFO interface. All the handshakes were honored immediately. Actual RTL might behave differently.
4. Multiple transactions on a single interface were not enabled. The AXI IP has the capability of issuing multiple transactions before the previous transactions were completed(upstream). In the existing testbench, the transactions serialized and sent one after another.

REFERENCES

[1]M. Chen, Z. Zhang and H. Ren, "Design and Verification of High Performance Memory Interface Based on AXI Bus," 2021 IEEE 21st International Conference on Communication Technology(ICCT),Tianjin,China,2021,pp.695699,doi:10.1109/ICCT52962.2021.9658046.

[2]N.Gaikwad and V. N. Patil, "Verification of AMBA AXI On-Chip Communication Protocol," 2018 Fourth International Conference on Computing Communication Control and Automation (ICCUBEA), Pune, India, 2018, pp. 1-5, doi: 10.1109/ICCUBEA.2018.8697587.

[3]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, doi: 10.1109/TENSYMP54529.2022.9864552.

[4]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.08600864,doi:10.1109/ICCSP.2015.7322617.

[5]P. Kumar M.P. and S. K. Panda, "Design and Verification of DDR SDRAM Memory Controller Using SystemVerilog For Higher Coverage," 2019 International Conference on Intelligent Computing and Control Systems (ICCS), Madurai, India, 2019, pp. 689-694, doi: 10.1109/ICCS45141.2019.9065407.