
مدينة زويل للعلوم والتكنولوجيا
Zewail City of Science and Technology


Zewail City for Science and Technology
University of Science and Technology
Nanotechnology & Nanoelectronics Engineering Program

# Verification of the Digital Data-Path of DDR5 PHY

A Graduation Project
Submitted in Partial Fulfillment of
B.Sc. Degree Requirements in
Nanotechnology & Nanoelectronics Engineering

Prepared By

| | |
|---|---|
| Abdullah Allam | 201700419 |
| Tarek Abou-El-Khier | 201701167 |
| John Saber | 201701287 |
| Mohamed Abdelall | 201700417 |
| Shehab Bahaa | 201700313 |

Supervised By

Dr. Hassan Mostafa Signature

2021 - 2022

# Acknowledgments

This project would not have been possible without the support of many people. Thanks to our project supervisor, Dr. Hassan Mostafa, who helped us greatly with his expertise and connections. Also thanks to Si-Vision engineers who offered guidance and support.

Thanks to Nanotech. & Nanoelectronics department, especially Dr. Amr Bayoumi, the program director, for facilitating the collaboration with our industrial partners and providing us with his help and support. And finally, thanks to our colleagues, and friends who offer endless support and love.

# Declaration

We hereby declare that the work which is being presented in the thesis, entitled "Verification of the Digital Data-Path of DDR5 PHY", in partial fulfillment of the requirement for the B.Sc. Degree in Nanotechnology & Nanoelectronics Engineering, Zewail City is an original work under the guidance of Dr. Hassan Mostafa, assistant professor at the University of Science and Technology, Zewail City.

The matter embodied in this thesis is original and has not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University.

Abdullah Allam

Tarek Abou-El-Khier

John Saber

Mohamed Abdelall

Shehab Bahaa

June 2022

# Abstract

The current state of the electronics industry shows a strong need for better DRAM performance, which relies heavily on the DDR physical layer (PHY) performance. The latest DDR5 JEDEC JESD79-5A and DFI standards address the industry needs and describe the features of state-of-the-art DDR5 PHYs. Functional verification of such sophisticated design is paramount, and it is the subject of this work. UVM along with SVA was used as a good solution to address the need of a robust and reusable testbench for the digital data path of the DDR5 PHY. In this work, we adhere to the verification flow, starting with the specifications extraction from the standards, developing the verification plan and test plan, then building a UVM verification environment written in SystemVerilog. The coverage-driven random testing methodology is also utilized. As a result of testing, 14 bugs were found and reported. Furthermore, high code and functional coverage were obtained indicating the completeness of the verification effort.

# Table of Contents

# List of Figures

# List of Acronyms/Abbreviations

| | |
|---|---|
| ACT | Activate Command |
| CDC | Clock Domain Crossing |
| CRV | Constrained Random Verification |
| DDR | Double data rate |
| DES | Deselect |
| DFI | DDR-PHY Interface |
| DQ | Data Bus |
| DQS | Data Strobe |
| DRAM | Dynamic random-access memory |
| DUT | Design Under Test |
| DV | Digital Verification |
| EDA | Electronic Design Automation |
| FV | Formal Verification |
| HW | Hardware |
| IEEE | Institute of Electrical and Electronics Engineers |
| IP | Intellectual Property |
| JEDEC | Joint Electron Device Engineering Council |
| MC | Memory Controller |
| ML | Machine Learning |
| MR | Mode register |
| MRR | Mode Register Read |
| MRW | Mode Register Write |
| NoC | Network on Chip |
| OOP | Object-oriented programming |
| OVM | Open Verification Methodology |
| PHY | Physical layer |
| PREab | Precharge all Command |

| | |
|---|---|
| RAM | Random Access Memory |
| RD | Read Command |
| RTL | Register Transfer Level |
| SDRAM | Synchronous DRAM |
| SFC | SystemVerilog Functional Coverage |
| SW | Software |
| SV | SystemVerilog |
| SVA | SystemVerilog Assertions |
| TB | Testbench |
| UVM | Universal Verification Methodology |
| VCS | Verilog Compiler and Simulator |
| VIP | Verification Intellectual Property |

# Chapter 1

# Introduction

## 1.1 General Overview

Modern computing systems are arguably the most advanced machines that humans have ever built. They are the most important pillar that enabled the modern era we live in as it serves as an indispensable tool for virtually any discipline of modern engineering, science, and art. Currently, the prevalent computing architecture (i.e., way of doing computation) is the Von Neuman architecture which was presented around the middle of the 20th century. It relies on the concept of "stored program computing" where the systems mainly consist of a processing unit and a memory element. The idea is that the program (algorithmic instructions to do some task) are fetched from the memory and the processor does the calculations and then returns the result back to a memory or a specific data output method such as a screen. Since then, the processor performance increased exponentially thanks to advances in transistor technology as described by Moore's law.

The ideal memory for such huge processing power should be large, fast, and inexpensive simultaneously. However, the underlying fabrication technologies that are used to build different types of memory cannot provide all the three attributes together. Therefore, a memory hierarchy is utilized, where each level of the hierarchy uses a different memory technology, to provide the best performance possible. An essential part of the memory hierarchy in a very wide range of applications is the Dynamic Random Access Memory (DRAM). DRAM offers relatively high capacity with a relatively fast operation. DRAM is volatile memory; i.e., it needs to be refreshed periodically to hold on to the data. Although DRAM also benefited from the technology scaling and advancements, the memory performance lagged behind processor performance for years leading to the term "the memory bottleneck". This is more evident in the age of big data, cloud computing, and artificial intelligence where a staggering amount of data is being moved and processed especially with the

ever-increasing memory-intensive workloads. Therefore, enhancing modern DRAM systems with higher bandwidth, lower latency, higher capacity, and higher reliability is desperately needed on a global scale given that the DRAM market size is expected to reach USD 221.67 billion by 2030 [1].

Double Data Rate (DDR) memories transmit data on both the positive and negative edges of the clock. DDR memory is twice as fast as Single Data Rate (SDR) memory. To address the current industry demands, DDR5 was introduced. Generally, the DRAM memory system, shown in figure 1, consists of the memory controller (MC), the Physical Layer (PHY), and the DRAM chips themselves. The PHY serves as the mediator which converts the signals of the MC interface, which is described by the DFI standard, to the signals of the DRAM interface, which is described by the JEDEC JESD79-5A standard. The MC is typically part of the processing unit and the DRAM is typically built on a separate chip. Therefore, the basic function of the PHY is to deliver commands and data between the MC and DRAM while handling differences in bus width, protocols, and clock frequency. The overall performance of the DRAM system relies heavily on the performance of the PHY. As mentioned above, the PHY specifications are described by two complex standards and the design must be checked to make sure that it really functions as intended by those standards.

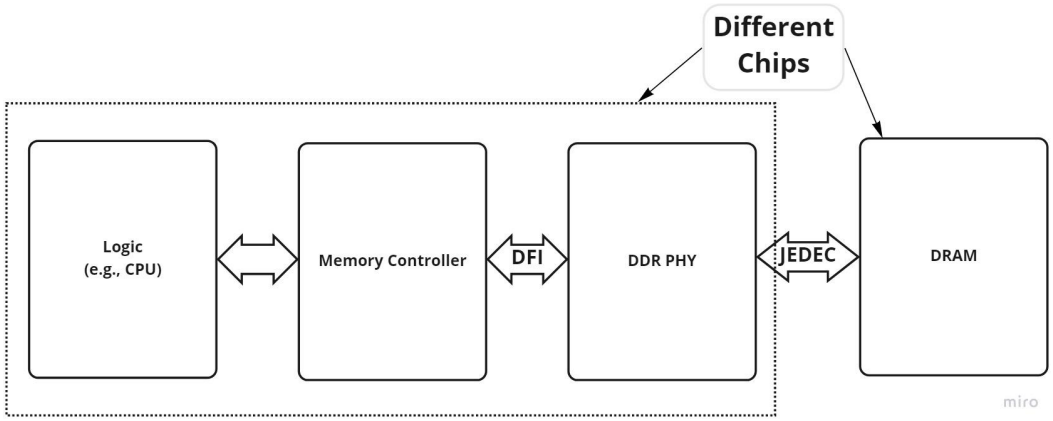


Figure 1. A simplified picture of the DRAM System

Verification of digital circuits is the process of checking the design's functionality against a set of specifications that are agreed upon to be the reference for the design. It has become an essential part of the digital design process because of the high complexity (millions of Boolean gates) of the designs and the very high cost of fixing

bugs after production. In addition, some markets are labeled safety-critical, such as automotive and space applications, where no chance of error is allowed. Furthermore, verification can take most of the product development time in state-of-the-art designs; therefore, it is an extremely critical part of the product cycle. There are two types of verification that are widely used: dynamic functional and formal verification. Dynamic verification implies testing the design against a reference model by applying stimulus to both the design and the reference model and then comparing the output. Formal verification tries to verify the correctness of the design mathematically without simulation (stimuli application). There are two types of testing in dynamic verification: direct and random testing. The current widespread method in complex designs is the random, coverage-driven, testing methodology because of its efficiency in providing a good level of confidence in the design within the allocated time-to-market which is getting tighter. The goal of our project is to develop a reusable Verification IP based on UVM and SVA which is to be leveraged by DDR5 PHY designers, DDR5 DRAM-based SOC developers, and system integrators whose designs adhere to the JEDEC JESD79-5A and DFI standards.

## 1.2 Problem Definition

To verify the correctness of the RTL describing the digital datapath of the DDR5 PHY. This is done by comparing the functionality against the specification defined by the DFI and JEDEC standards in addition to the extra specifications and constraints proposed by the RTL design itself (implementation-specific specification).

## 1.3 Objectives

The objectives are as follows:

A. Extract the design features from the DFI and JEDEC JESD79-5A standards.

B. Develop the verification plan.

C. Develop a test plan.

D. Build a UVM verification environment written in SystemVerilog.

E. Reach the target code and functional coverage.

F. Thoroughly report the bugs.

## 1.4 Functional Requirements

The outputs of the project should adhere to the following requirements:

A. The feature list should be complete; i.e., it should include all the features of the PHY as specified by the DFI and JEDEC JESD79-5A standards in addition to the specification proposed by the RTL design itself.

B. The verification plan should be complete; i.e., it addresses all the extracted features.

C. The SystemVerilog code (UVM environment) should adhere to the principles of object-oriented programming and UVM best practices for reusability and code performance.

D. The test plan, when translated to code, should achieve high code and functional coverage.

E. The assertions should complement the UVM-based verification and address the major and critical design specifications.

F. The testbench should be robust across different simulators.

## 1.5 Thesis Organization

The structure of the thesis is as follows:

**Chapter 2: Standard used**

This chapter includes the various IEEE standards used in this work

**Chapter 3: Market and Literature Review**

This chapter discusses the literature and state-of-the-art methodologies that provide a

foundation for the design, and development of the DDR PHY verification environment. Also, it includes the tools used in this approach.

**Chapter 4: The Digital Verification Process**

This chapter introduces the verification challenges, constraints, and how to track progress. Also, the tasks of this process are explained along with the technologies used.

**Chapter 5: The Universal Verification Methodology**

This chapter introduces the Universal Verification Methodology (UVM). It describes the different classes provided by the UVM library and their usage.

**Chapter 6: The DDR5 PHY**

This chapter describes the DDR5 DRAM system and the PHY digital and analog circuit components. It also explains the DUT features as described in the DFI and JEDEC JESD79-5A standards in addition to the design-specific features.

**Chapter 7: Project Design**

The Verification environment used to test the DDR5 Memory PHY is discussed in this chapter. Its constraints and alternatives and the selected methodology.

**Chapter 8: Project Execution**

This chapter includes the description of the implementation of each component of the testbench (e.g., piece of code).

**Chapter 9: Results**

This chapter shows the results obtained after project execution. It shows the coverage results and analysis in addition to the bugs that were found and their explanation.

**Chapter 10: Cost Analysis**

This chapter describes the financial utility of the work done in this project and mentions how the output of the project can be monetized.

**Chapter 11: Conclusion and Future work**

The chapter consists of the conclusions and details of possible future work.

# Chapter 2

# Standards used

A physical layer facilitates the communication between the memory controller and the DRAM. In order to perform this functionality, it should satisfy both communication protocols between the memory controller and PHY and between PHY and DRAM which are DDR PHY Interface (DFI) standard and JESD209-5A standard respectively. Therefore, both standards will be considered the golden references from which the PHY features and virtual environment will be constructed. Furthermore, the project utilizes simulation-based verification using UVM and the SystemVerilog language. Hence, the testbench development will rely on the IEEE standards of UVM and SystemVerilog too.

## 2.1 DDR PHY Interface (DFI v5.1)

The DDR PHY Interface (DFI) protocol helps in transferring command information and data across the DFI and between the DDR memory controller (MC) and the DDR PHY (PHY). It defines the signals, timing parameters, and programmable parameters that are used to do so. DFI consists of interface groups; each one of them contains its own signals, timing parameters, and programmable parameters. The interface groups are Command, Write Data, Read Data, Update, Status, PHY Master, Disconnect Protocol, Error, 2N Mode, Low Power Control, MC to PHY Message, and WCK Control. Each interface group contains signals that are either required for all DRAMs, required for certain DDR DRAMs or optional for the others. The purpose of the DFI protocol is to make the communication between MC and PHY as easy as possible and nearly compliant with all generations of DRAMs while moving the complexity of each DRR requirement to the JEDEC interface [2].

## 2.2 JEDEC interface Standard (JESD79-5A):

The goal of the JEDEC JESD79-5A standard is to describe the minimum requirements for a JEDEC-compliant SDRAM device. The advancement of DRAM

generation should affect the JEDEC JESD79-5A standard hence, each generation should have its own JEDEC JESD79-5A standard which is not the case for DFI. These requirements could be signal requirements, timing requirements, and so on. These must be satisfied to ensure harmony in the data transfer between MC and DRAM. This standard was published in October 2021 [3].

## 2.3 SystemVerilog IEEE 1800 standard

SystemVerilog is a hardware description and verification language used to design, model, simulate, test, verify and implement electronic systems. It's based on Verilog with additional extensions i.e. Object-Oriented Programming(OOP) capabilities. These capabilities ensure maximum reusability, portability, and scalability of the verification IP developed. The language also comes with verification features to minimize the development time while ensuring full coverage of all the design specifications. Some of these features are constrained randomization, assertions, functional coverage, and code coverage. This standard was published on 6 December 2017

## 2.4 Universal Verification Methodology (UVM) IEEE 1800.2 standard

UVM is a standardized and the most common methodology used to verify integrated circuits. It reduces the complexity of developing a verification framework by providing a well-established, ready-to-use, and universal framework. This paradigm shift lets verification engineers concentrate on test generation and harness and develop modular, reusable, and scalable testbenches that can be deployed across multiple projects. UVM brings in a layer of abstraction where each component in the verification environment has a specific role. For example, the driver class is responsible for driving signals to the design while the monitor will be responsible for monitoring the output of the design and propagating this output to the scoreboard and subscriber for further processing of the data i.e. comparison with the reference model and coverage collection. This standard was published on 4 June 2020 [4].

# Chapter 3

# Market and Literature Review

## 3.1 Overview and Verification Trends

Design verification is no longer an optional endeavor in modern digital circuit design. Verification importance grew with the growing design complexity to the point that the verification effort consumes about half of the design development cost and can reach 80% of the design time. Moreover, the current state of the market, where products are expected to be finished in aggressively short time periods, adds to the importance of solving the verification bottlenecks and using trending methodologies that accelerate the verification process [5].

The target of a verification team is to ensure the device under verification (DUV) is functional given its environment interacts with it properly. The rise of reusable designs or intellectual properties (IP) paved the way for the verification intellectual properties (VIPs) which may include both simulation-based verification testbenches and formal verification materials. During the development of a VIP, two problems arise: the short time available for verification limits the extended usage of simulation-based testbenches, and the lack of formal tool scalability limits the replacement of simulation by formal methods. Therefore, a hybrid system using both techniques will ensure hitting the verification target more efficiently. The simulation-based verification can be further enhanced by using data analytics to generate optimized tests, and the formal verification is reserved for certain features and use cases that are difficult to cover using simulation [5].

Traditional functional verification methods become inefficient in meeting the time-to-market goals with satisfiable coverage closure because of the increase in functional specification of modem hardware designs. Hence, finding new techniques that minimize the development time while maintaining verification requirements is crucial. Machine learning (ML) models proved that they can be used in automating major tasks of the development process while letting engineers concentrate on making

the testbench more robust by adding more coverage points and so on. Some verification areas such as stimulus constraining, test generation, coverage collection, bug detection, and localization show noticeable improvements while they are developed with ML models [6]. For example, deploying an artificial neural network (ANN) in test generation shows a 24.5× speed up in functionally verifying a dual-core RISC processor specification [7]. Another example shows attempts to deploy ML models such as Markov models and inductive logic programming to reach a faster coverage convergence rate [8]. These examples and more show the potential usage of ML models in enhancing functional verification.

Formal methods are a collection of techniques to analyze a description of a digital system (either native code or a model) as a mathematical object. [9] reports the existing tools that utilize formal verification for digital systems. while [10] and [11] explore the use of formal verification in embedded systems and IoT. The studies conclude that formal verification should be considered in these fields and that further systems need to utilize it alongside dynamic verification.

## 3.2 UVM in Modern Verification

Universal Verification Methodology (UVM) is becoming an industry standard that different tools and simulators support due to its reliability, reusability, and flexibility. It's a Verification Framework based on SystemVerilog created by Accellera. UVM is "simulation-oriented" so it can be used alongside other assertion-based methodologies. In Qamar et. al, 27 papers published between 2017-2019 were analyzed to summarize the latest advancements and utilizations of UVM [12]. In these papers, UVM was found to be used in various domains including Analog And Digital circuit design, Computer System Architecture & IoT, Encryption and Cryptography, and even electro-mechanical systems. Some of these papers used the framework with slight modifications such as fault injections while others used the framework without extensions. Various tools and languages were used with UVM including C/C++, SystemVerilog, Verilog, and VHDL. The paper also mentions the benefits of using UVM in these systems which included being standard, portable, reusable, flexible, and including functional and code coverage. The study concludes that UVM has

significantly improved phasing mechanism and provides advanced features which give it an edge over OVM.

The adoption of UVM among the top-tier EDA companies also adds to its favor. The largest two EDA companies, Cadence and Synopsys, offer comprehensive VIPs for DDR5-based systems. These VIPs are based on SystemVerilog and UVM as shown in [13], [14], and [15]. These VIPs are also endorsed by the memory manufacturing giants, such as Micron [14], adding to them more credibility. Furthermore, state-of-the-art EDA tools (such as Synopsys VCS, Cadence Xelium, and Mentor Questasim) support the use of UVM. These tools occupy most of the EDA market indicating that indeed UVM is becoming the de-facto library used in digital design verification.

## 3.3 Conclusion

The points presented above support our design choice in this work. A UVM-based verification environment for the digital datapath of a PHY that is built for DDR5 DRAM was developed. Also, the testbench is supplemented with the use of SystemVerilog Assertions. The whole VIP could be easily adjusted and reused for verifying different DDR5 PHY designs.

# Chapter 4

# The Digital Verification Process

This chapter introduces the verification challenges, constraints, and how to track progress. Also, the tasks of this process are explained along with the technologies used.

## 4.1 Challenges of Verification

Nowadays, ASICs and FPGAs are in the order of multi-million gates, and their description can easily take hundreds of thousands of HDL lines. The DV engineers are required to ensure that the design implementation performs as intended in the specifications, and flag bugs when the implementation acts differently. To detect these bugs, complex simulations are run on the design. However, the task is not that simple as there are main challenges, the size explosion of the state space and detecting the incorrect behavior [16].

### 4.1.1 The State-Space Explosion

A typical design contains thousands of flip flops, latches, combinational logic, and RAM arrays. All are used to control the behavior of the circuit. The chip inputs are used to manipulate the internal state by changing the values stored in the flip flops, latches, and RAM arrays. At any specified time, the chip is in one of the huge numbers of possible states. Plus, the next state can be any of these possible states depending on the inputs and current state, so to verify the chip exhaustively, the DV engineer needs to check that all possible current states along with all input combinations provide the correct output and correct next state [16].

The reachable state space of the digital designs has an exponential nature with the number of state bits which makes it one of the core verification challenges. This state-space explosion is independent of the verification methodology used. Also, the exponential complexity exists in both dimensions, time and memory space [16].

To mitigate this challenge, a divide and conquer methodology is followed so that the design is broken down into smaller, more manageable sub-components. Once the sub-components are verified, they are stitched back together. Also, another mitigation mechanism is defining the illegal states of the design based on the specifications. Illegal states reduce the number of states that need to be verified. However, this solution is not always possible because the illegal state cannot be ignored in certain designs [16].

### 4.1.2 Detecting Incorrect Behavior

Another challenge is detecting the violations in the design implementation. With all the possible states and transitions, the DV must be able to decide whether the design is acting correctly or not. The solution to this challenge is validating the logic at the transaction level so that the focus is shifted to the behavior of the design [16].

Briefly, the verification challenges come down to

- driving the input and state transitions scenarios
- flagging the incorrect behavior of the design implementation

To attack these challenges, digital verification engineers use plenty of tools and technologies that increase the level of abstraction and automation and enable parallelism. These technologies are discussed in sections 4.4 – 4.7 [16].

## 4.2 Verification Constraints

The verification team manages the process by balancing the triple constraints: (1) schedule, (2) cost, and (3) quality [16].

### 4.2.1 Schedule

The success of electronic products depends heavily on showing up in the marketplace at the right time as a huge amount of revenue goes to the product coming first in the market, so more than any other industry, delays in introducing products to the market can be deadly [16].

### 4.2.2 Cost

Companies aim to maximize profit from their hardware products. One key mechanism is to minimize the costs of development and manufacturing. Roughly, 40-50% of project resources are used for functional verification. This is evident from the chart in figure 2 which shows the costs for different parts of the product development process. To reduce the development costs, the productivity of functional verification must increase. Another reason for improving productivity is that every year, the complexity of the designs being verified grows higher than the rate of productivity growth, so there is a gap between what needs to be verified and what is getting verified [16], [17].

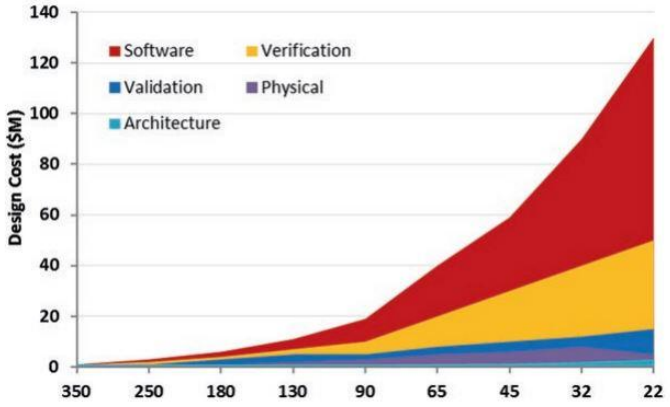


Figure 2. Verification cost as technology node shrinks [17].

Also, as shown in figure 3, the cost of undetected bugs grows over time. If a bug is detected during verification, its cost is just modifying the HDL. On the other hand, a bug detected during testing after fabrication can cost hundreds of thousands of dollars due to the added time-to-market and refabrication. Finally, detecting a bug by the customer is the most costly because, in addition to the warranty or replacement costs, it affects the company's reputation.

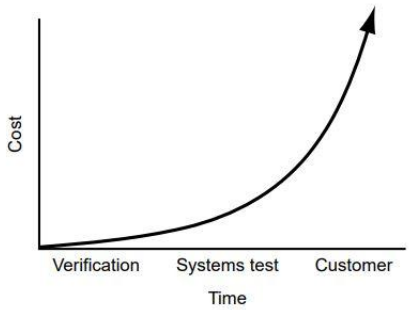


Figure 3. Cost of undetected bugs over time [16].

### 4.2.3 Quality

Electronics consumers expect the products to have high quality, so if a company introduces a product with low quality, it can be devastating to the company's reputation. Another aspect is that components failure loads warranty costs on the company [16].

The verification process affects all these constraints. Chips are fabricated sooner if the fabrication team can detect the bugs fastly and efficiently. Because it is on the critical path of the project, it is convenient to track the verification productivity. It is measuered in terms of the quality of the bugs and time. As shown in figure 4, the more steep the curve of bugs number with respect to time, fewer costs and schedule time is required. The other factor measured, the quality of bugs, is a qualitative one. The more difficult and complex the scenario required to detect a bug, the higher quality is the bug. As time passes, the average complexity of the bugs should grow [16].

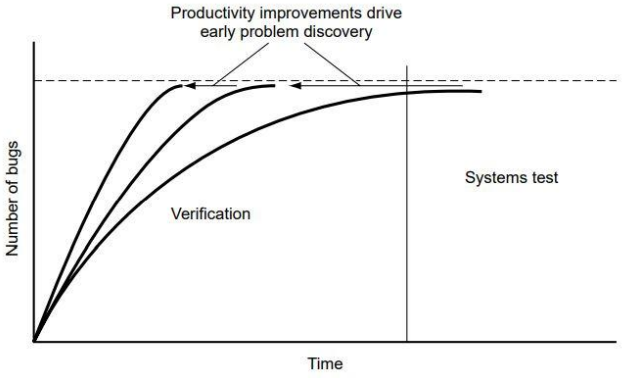


Figure 4. productivity with respect to schedule and cost [16].

## 4.3 Verification Tasks

As discussed previously, the verification process is time-consuming and involves many tasks. These tasks are grouped into phases carried out in a loop throughout the time of the process. Figure 5 shows this functional verification cycle. The first phase in the loop is the development phase. It involves developing the verification plan, testbench architecture and test cases, and creating the environment. The second phase is the simulation. Next, the debugging phase is performed at the transaction level using a reference model and at the signal level using assertions. The last phase is collecting all types of coverage which are code, functional, and assertions coverage. This coverage is used as a feedback for the development stage [17].

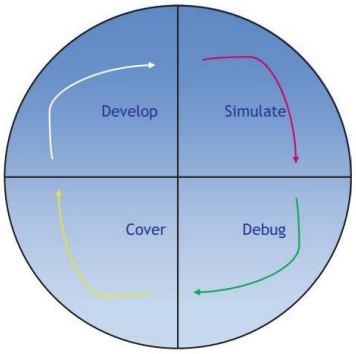


Figure 5. Typical verification process loop [17].

## 4.4 Digital Verification Technologies

To overcome the challenges of verification and increase the productivity of the DV engineers, Automation is one important mechanism that allows autonomous completion of tasks with predictable results. It allows parallelizing the effort which reduces the overall verification time. However, not all processes can be automated due to the variety of protocols, functions, and interfaces that need to be verified, so various tools and technologies have been developed to automate many parts of the verification process [18].

The field of digital verification is exhaustive in its breadth and depth. It requires knowledge of many technologies and methodologies. These technologies are including but are not limited to SystemVerilog, UVM, SystemVerilog Assertions

(SVA), SystemVerilog Functional Coverage (SFC), Constrained Random Verification (CRV), Clock Domain Crossing (CDC), Network on Chip (NoC) verification, low-power verification, static formal verification, and HW/SW co-verification. An EDA tool includes many of these technologies. The job of the DV engineer is to use the essential technologies so that no significant bugs are missed. These subsections present some of the verification technologies [17].

# 4.5 Coverage-Driven Verification

Coverage is the term used to describe measuring the progress towards a complete design verification. As the simulation progresses, the coverage tool collects information, then after post-processing, a coverage report is generated. This report is used to identify the coverage holes, so tests are created or modified to fill the holes. The coverage feedback process will be repeated until a satisfactory coverage level is reached. There are two types of coverage: code coverage and functional coverage. They are two different metrics [19].

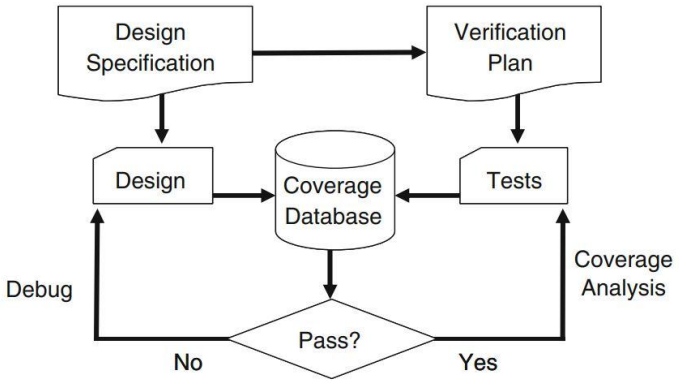


Figure 6. Coverage verification methodology [19].

## 4.5.1 Code Coverage

Code coverage measures the coverage of the design structure such as branches, expressions, state transitions, etc. It is collected automatically by the tool used, i.e., not specified by the user. The code coverage report reflects how much of the design code has been exercised by the tests. However, 100% code coverage does not mean that the design verification is complete because it does not measure how much of the verification plan has been exercised, so if the implementation is missing a feature, the

code coverage will not catch it. This is why the second type of coverage is needed [19].

### 4.5.2 Functional Coverage

Functional coverage is specified by the user based on the design specifications. It reflects how much of the design intent has been exercised. In other words, it is used to check that the testbench has covered all corner cases, key design features, and possible failure modes as described in the specifications. However, gathering data for functional coverage may increase the simulation time and memory space, so the data used for the analysis and improving the tests should be the only data measured. Similar to code coverage, achieving 100% functional coverage does not mean that the design verification is complete because if the code coverage is low, this means that the testbench is not exercising the entire design, so more design features and corner cases need to be added [19].

Based on the previous discussion, it is clear that the verification goal is achieving both high functional and code coverage. Figure 7 shows the coverage comparison that should be continuously performed, so either the design or the testbench is modified to achieve good coverage [19].

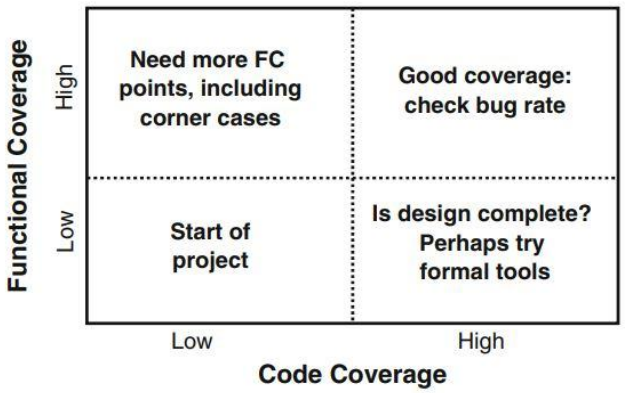


Figure 7. Coverage comparison [19].

### 4.5.3 Bug rate

Along with the coverage metrics, the bug rate should be checked continuously. Over the project life, how many bugs are found each week should be measured, so when the bug rate drops, then different approaches to create corner cases should be used.

The bug rate depends on many factors such as new design changes, project phases, and recently integrated blocks. Unexpected changes in the bug rate signal a possible problem [19].

### 4.5.4 Coverage Components of SystemVerilog

As shown in figure 8, the functional coverage uses coverpoints and covergroups to specify the function needed to be covered. They allow measuring transition and cross coverage. These components are mainly used at the transaction level in the subscriber. Another component is the “cover” statement associated with the assertions. This component is mainly used at the pin level. Every property that uses an “assert” statement should also use a “cover” statement. An “assert” statement checks the behavior of the design, and a “cover” statement checks if the property got excercised, so both statements complement each other [17].

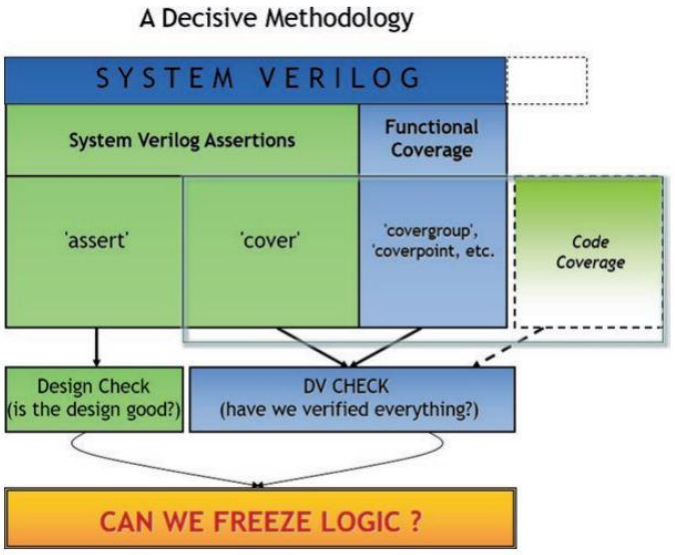


Figure 8. functional coverage methodology [17].

## 4.6 Simulation-Based vs Formal Verification

Simulation-based verification traverses the reachable state space of the DUT, and at each state, checkers and assertions test whether the current state is legal and whether the output results are correct as shown in figure 9. The main advantage of the simulation-based verification is that as the size of the DUT increases, the decrease in the simulation speed and increase in the model size are linear. However, this approach

has a severe limitation that the state space size does not allow exhaustive verification; The simulation-based verification checks the design in a case-by-case mannar with no guarantee that the design behaves correctly in the unchecked states, so in essence, simulations can only show the existence of bugs but can never show their absence. On the other hand, formal verification provides stronger verification for the DUT or at least part of it. It checks the properties for all the DUT states as it processes the DUT model and attempts to prove mathematically the validity of a propery or give a counterexample. Although the formal verification algorithms run freely over the state space, the checking is done exhaustively, so due to the challenge of state space explosion and limited resources, it is not used at the full SoC level. To solve this problem, certain blocks are specified for formal verification per run. This technique is called case splitting. Also, some constraints can be applied to the inputs and sequences as shown in figure 10 [16].

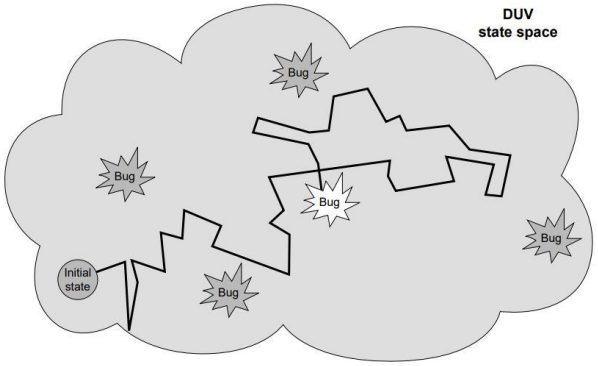


Figure 9. Path of one simulation through the reachable state space of a DUT [16].

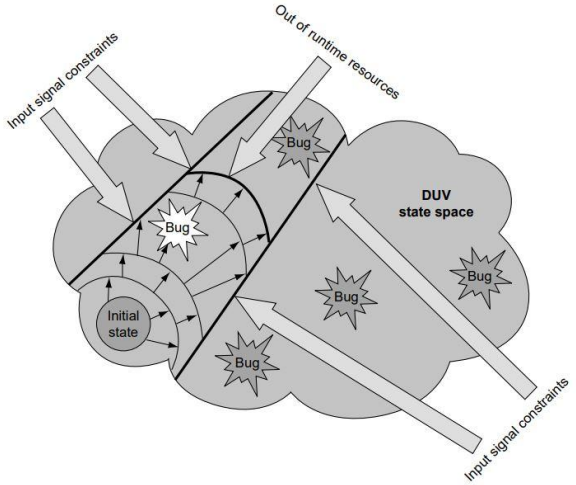


Figure 10. practical limitations on formal verification (runtime resources) [16].

## 4.7 Assertion Based Verification

Most of today's testbenches use coverage-driven constraints random verification mindset along with Transaction-Level Modeling (TLM) in order to avoid dealing with signals and increase the level of abstraction to facilitate functional verification tasks. This can be achieved using UVM_driver and UVM_monitor which convert from transaction level to pin level and vice versa respectively. TLM also makes it easy to compare two transactions one from monitor and the other from a reference mode. However, we still need to probe the signals themselves, assert the timing dependence between them, make sure that the driver is driving the signals properly and assert the correctness of its behavior. Assertions can be created using System Verilog Assertion (SVA) and can be divided into two categories: immediate assertions and concurrent assertions.

### 4.7.1 Immediate assertion

Immediate assertions check whether or not an expression is evaluated true when the statement is executed. The procedural code of the testbench can check the value of certain signals and show error messages if they are not as expected. The following example from System Verilog For Verification illustrates the idea clearly:

```
arbif.cb.request <= 2'b01;
repeat (2) @arbif.cb;
if  (arbif.cb.grant !=  2'b01)
    $display("Error, grant != 2'b01");
```

In this example, the bus request is asserted, hence, the grant signal must be asserted two clock cycles after the assertion of the request signal or an error message will appear. A better and more compact way to do this is by using the assert keyword as shown below:

```
arbif.cb.request <= 2'b01;
repeat (2) @arbif.cb;
a1: assert  (arbif.cb.grant ==  2'b01);
```

Note that the logic is reversed since assert means the signal behaves as expected. On the other hand, you should give a meaningful name for the assertion in order to speed

up debugging process instead of using a1. Moreover, else statement may be used after the assert statement if you want to omit the default message and use your own message.

### 4.7.2 Concurrent assertions

On the other hand, there are concurrent assertions that run continuously throughout the simulation time and check the signals for the entire simulation. They are written inside a module that will be instantiated later in the testbench. Concurrent assertions are continuous in essence so, they cannot be written inside a procedural statement like initial or always block. Moreover, a sampling clock must be specified inside the assertion statement. Another example from the previous book is used here to clarify the idea:

```
interface arb_if(input bit clk);
    logic [1:0] grant, request;
    bit rst;
property request_2state;
    @(posedge clk)  disable iff (rst)
        $isunknown(request) == 0;   // Make sure no Z or X found
    endproperty
assert_request_2state: assert property (request_2state);
```

In this example, the desired behavior is written inside a property statement (request signal is not Z or X except at the reset condition). Then, the property is asserted using assert property keyword. Moreover, assertions can be put inside the interface to check the correctness of the signal against the standards used. This use case is one of the most powerful advantages of assertions.

# Chapter 5

# The Universal Verification Methodology

## 5.1 Overview

The Universal Verification Methodology (UVM) is an open-source standard methodology used by verification engineers to develop testbenches for design units. There were different verification methodologies that preceded the UVM, such as Open Verification Methodology (OVM) which is developed by Mentor Graphics, Universal Reuse Methodology (URM) from Cadence, and Synopsys also developed its own Verification Methodology Manual (VMM). These methodologies are based on System Verilog hardware verification language (HVL) [17].

The UVM was developed by Accellera based on the OVM [20]. Since then, the UVM has been adopted by all EDA companies which makes it "universal" and used extensively in the field of IC design verification. In 2014, the UVM was standardized by IEEE, project number 1800.2 [4]. The figure 11 below shows the development of verification methodologies over the years.

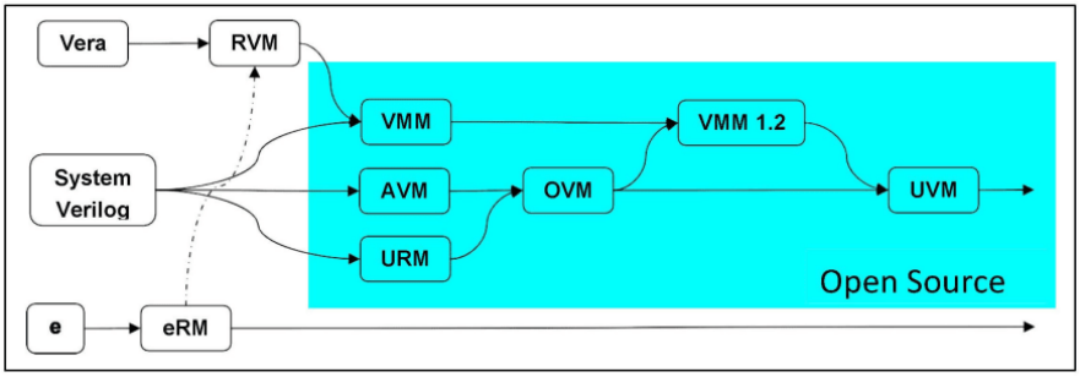


Figure 11. UVM development history [21]

UVM's main goal is to develop reusable, robust, and configurable testbenches. It is, essentially, a class library that exploits the features of object-oriented programming (OOP), such as inheritance and polymorphism. Also, it provides a broad spectrum of utilities, making it a perfect tool to implement almost any structure for a testbench.

## 5.2 UVM Testbench Architecture

UVM TB hierarchy is depicted, in a generic form, in the figure below. It shows that a testbench module instantiates the DUT, interfaces, and the UVM tests, in which each of the test classes, in turn, has an instance of the whole UVM environment. The communication between the UVM components is based on the TLM, which provides means of communication methods for receiving/sending transactions between UVM components [20].

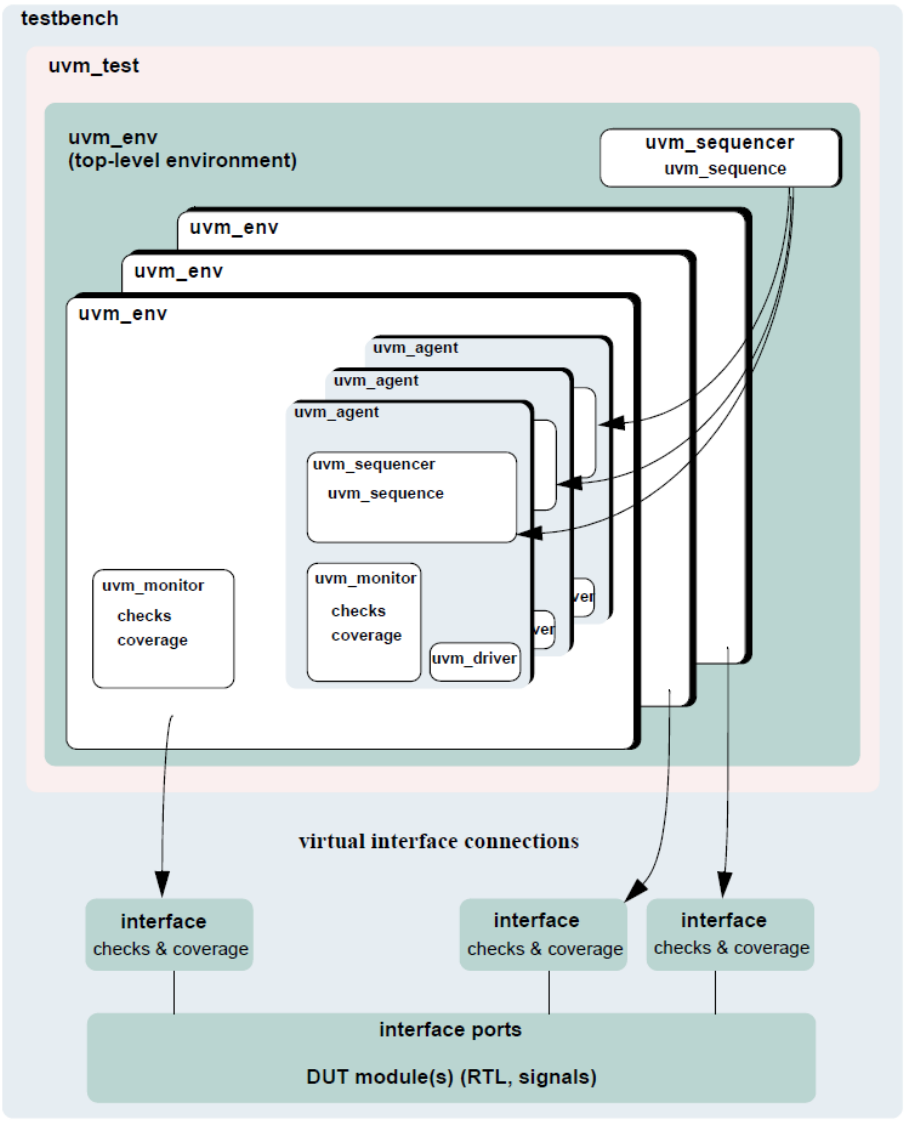


Figure 12. UVM Testbench Architecture [20]

A test bench can include more than an environment, for a SoC verification as an example, where each block in a SoC can have its own environment. Also, an

environment can have multiple agents as each agent is responsible for an interface stimulus and sampling using the UVM driver and monitor (employing TLM). Each one of these components is illustrated in the subsequent sections.

## 5.3 UVM Class Hierarchy

Accelera's UVM user guide illustrates the hierarchy of the UVM base classes as depicted in the figure below.

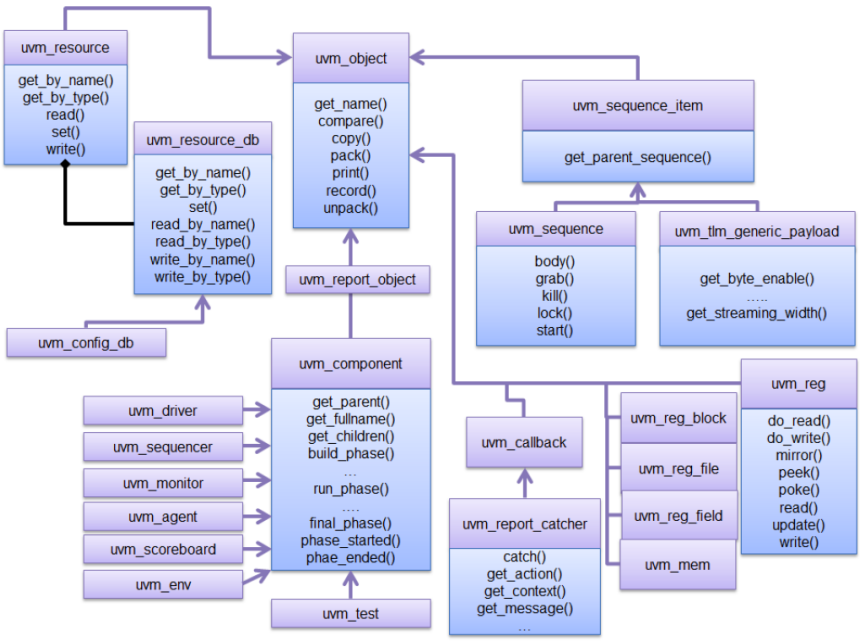


Figure 13. UVM Class Hierarchy [20]

The uvm_object is the parent of all UVM classes each child/layer extended or added to the hierarchy inherits its parent's functionality and adds new functionalities. The uvm_component class is the base class of the testbench components, it inherits the uvm_object and adds the phasing functionality, thereby, the testbench components are built, connected, and configured properly.

## 5.4 UVM Test

The UVM Test class is a top-level component class, it instantiates and configures the UVM environment. Its purpose is to apply the sequences and test scenarios onto the DUT through the UVM environment. This is done by invoking the sequences of

transactions defined by UVM sequence base class. Typically, a base test class is created extending the uvm_test class. It includes all environment configurations. It is then extended further to implement specific test cases and run selected sequences [17].

## 5.5 UVM Environment

The UVM Environment base class typically instantiates other components like UVM Agents, Scoreboards, and Subscribers. Its purpose is to implement and configure verification IP components, facilitate transaction communication between them, and also to boost reusability. The agents are connected to the scoreboard and the subscriber through a TLM analysis port for broadcasting transactions. The test can configure the UVM environment default configuration to achieve a specific verification goal [17].

## 5.6 UVM Agent

UVM Agent is an essential part of the UVM environment. It is the component that interacts directly with the DUT. It includes UVM Driver and Monitor which are responsible for the conversion from the transaction level to pin-level activities and vice versa. It also includes a UVM sequencer, which regulates and arbitrates sequences flow to the driver. Typically, each agent in an environment is responsible for one interface for driving and monitoring [20]

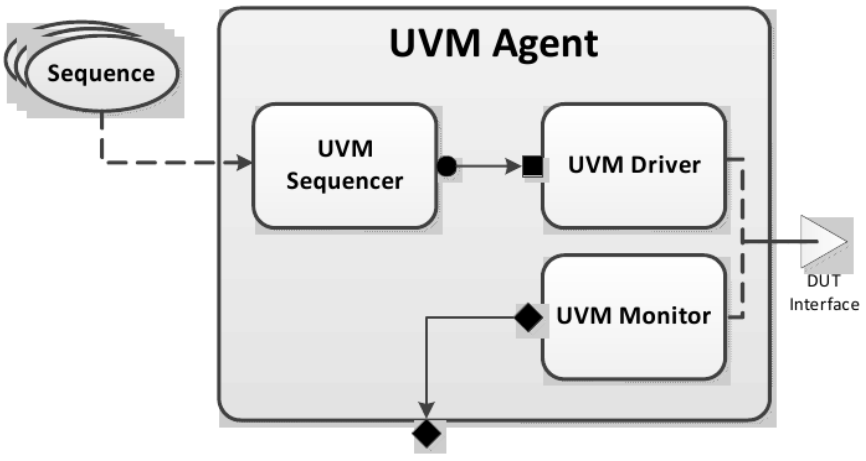


Figure 14. UVM Agent [20]

An agent can be active or passive, wherein the active mode it has the utility to generate a stimulus to the DUT through the driver. On the other hand, in the passive mode, the driver and the sequencer are disabled and there is only a monitor operation [20].

## 5.7 UVM Sequence Items

At the basic level, a sequence item, or transaction, represents the abstraction of signals that are used to communicate with the DUT. Sequence items are the basis on which different sequences will be created. Things that influence sequence item content include the information required by the driver to complete a pin-level transaction and the ease with which stimulus content can be generated (often with randomization in mind) [22].

## 5.8 UVM Sequencers & Sequences

The sequencer is the component that manages the delivery of stimuli, thereby providing transactions to a driver. Tests containing complex sequences are facilitated by using standard UVM sequencers with predefined arbitration functions. A sequencer can be viewed as an arbiter at its most basic level. It controls who contacts the driver; this is equivalent to who contacts the interface [22].

One or more sequence items, which are sent to a driver afterward, are represented by a sequence. A collection of sequences can be used in constrained-random testing to allow for a level of randomization beyond that achievable by randomizing the transactions only. Sequences can be used to represent successive or parallel, dependent or independent stimuli on one or multiple interfaces. UVM provides extensive support for sequence definition and application [22].

## 5.9 UVM Driver

The UVM Driver is responsible for converting the abstracted high-level transactions/sequence items coming from the UVM sequencer into lower-level pin-level activity driven onto the DUT. The sequencer communicates with the driver

through a TLM handshake mechanism, to make sure that the received transaction is consumed by the driver and it is time to get a new one from the sequencer. Moreover, there are some essential methods defined in UVM Driver and are used to communicate transactions with the sequencer [17]:

- `get_next_item()`: This method is used to get sequence_item (REQ) from the sequencer and it blocks until the REQ item is available.
- `try_next_item()`: This is similar to the previous method but it is non-blocking. It returns null if no sequence items are available.
- `item_done()`: This is a non-blocking method that completes the driver-sequencer handshake, called after `get_next_item()`. Also, it returns the response sequence item (RSP) that is observed by the driver back to the sequencer and it turns to the sequence class. This is needed when the sequence generation (in the sequence class) is dependent on the response of the DUT.

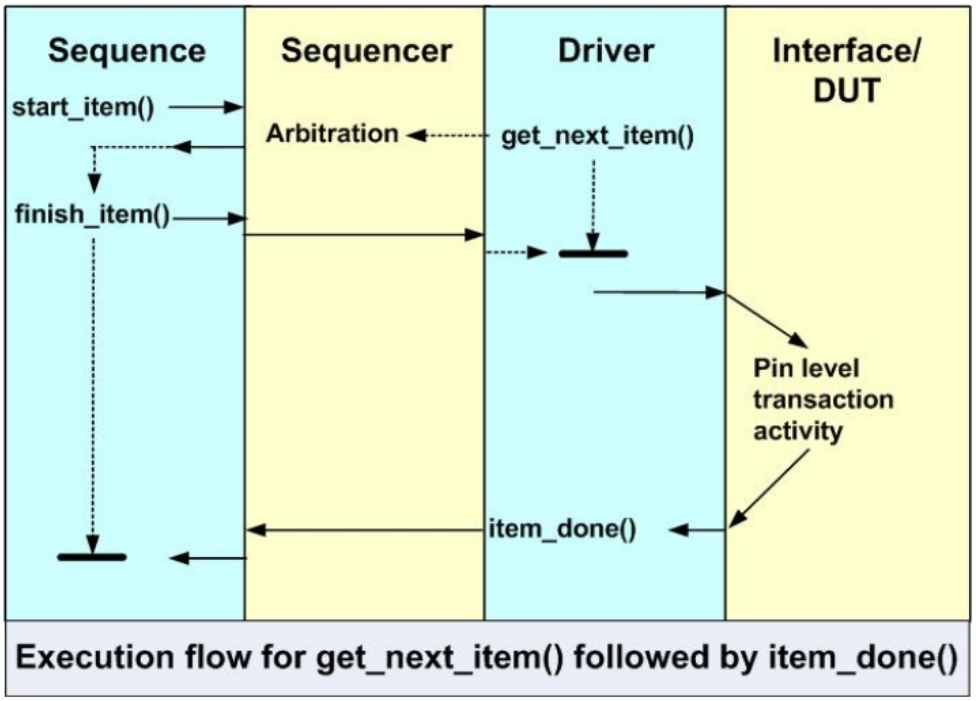


Figure 15. UVM Driver flow of execution [22]

## 5.10 UVM Monitor

A monitor is a component class that connects to the DUT via a virtual interface to observe the DUT's response to stimuli. The monitor's job is to collect the pin-level activity from the interface and construct appropriate transactions that are at a higher level of abstraction. Normally, the monitor is passive and does not apply stimuli to the DUT in any way; this is essential to accurately and reliably broadcast the DUT behavior to the analysis components of the testbench without intervention (i.e., scoreboard and subscriber). The monitor must contain knowledge about the specification that the DUT implements so that it recognizes certain patterns and then collects them in a meaningful transaction; this also includes the knowledge of when to sample signals, when not to sample, and when a transaction is complete and can be broadcasted. Normally, the run phase of the monitor will contain a loop; on each iteration of this loop, a transaction is created and broadcasted to the scoreboard and subscriber through the analysis port [22].

## 5.11 UVM Scoreboard and Reference Model

The scoreboard and reference model constitute the analysis section of a testbench. They determine whether the DUT behavior is as predicted by the design specification, assuring that the design implemented the specifications as intended. Monitors observing the DUT's input ports broadcast the constructed transactions to the scoreboard and reference model [22].

An automatic technique to check the result when random stimulation is fed into the DUT is needed as an alternative to manual checking. Thus, the reference model is a verification component and is considered a "golden predictor" of the DUT behavior. The same input stimuli that are sent to the DUT are applied to the reference model; it then generates the expected response that is correct by definition. A reference model will generate predicted output, which is compared to the actual DUT output by the scoreboard to determine if the DUT passed or failed. Typically, reference models are developed at an abstract level and can be written in high-level languages like C, C++, Python, or SystemC [22].

Scoreboards are UVM components that receive transactions sent by a monitor and compare to see if the design is working as it should. There are two ways of comparing the expected and actual transactions depending on the DUT: in-order and out-of-order. The in-order comparison anticipates that the expected and actual transactions will appear in the same order; the transactions' arrival is independent, but they arrive in order. The out-of-order comparison does not assume that transactions will appear on the expected and actual sides in order. Therefore, unmatched transactions must be kept until a corresponding transaction shows up in the opposite stream. In addition a way of matching corresponding expected and actual transactions must be implemented. The evaluation result is used by the scoreboard to report and record failures. In most cases, successful evaluations are not reported or they are allowed only while debugging, but they can be documented for later summary reports. On the other hand, error messages are typically always operational throughout the testing endeavor [22].

## 5.12 UVM Subscriber

A subscriber is typically a component that receives transactions through an analysis port. For the basic example of a subscriber, a UVM base class is provided. For more complex subscribers that receive transactions from multiple analysis ports, various macros and classes can be used. By that definition, the majority of analytical components are subscribers. To perform analysis transaction processing, the uvm_subscriber component class encapsulates a uvm_analysis_export and its related virtual write method. Collecting coverage information is paramount to measuring the progress of the verification process. The UVM subscriber class can be used to build a coverage collector. This class samples the transactions observed and broadcasted by the monitor and links them to functional coverage groups that are defined within it [22].

## 5.13 Resource Database

UVM has a resource database that contains any piece of information that can be shared between UVM components and objects. This database can be accessed using

uvm_config_db. Two methods can be used for that purpose: the set and get method. uvm_config_db::set method is used to put information into the database while the uvm_config_db::get method is used to retrieve the information from the database. The uvm_config_db is a type-parametrized class that accepts any type of parameters such as a class, a uvm_object, a built-in type like a byte, bit, or a virtual interface. As mentioned before, a resource database can be used to share information among UVM components, however, it has two typical use cases: the first is to pass the virtual interface from the DUT domain to the test while the second is the pass the configuration objects through the testbench hierarchy.

### The set method:

The general declaration of any set method is

```
void uvm_config_db #( type T = int )::set( uvm_component cntxt ,
string inst_name , string field_name , T value );
```

Where:

- T is the resource type.
- Cntxt and inst_name are used to locate the needed resource within the database.
- Field_name is the resource name.
- Value: is the value that needed to be stored in the database

### The get method

For the get method there is a general declaration too which is

```
bit uvm_config_db #( type T = int )::get( uvm_component cntxt ,
string inst_name , string field_name , ref T value );
```

Where:

- T is the resource type.

- Cntxt and inst_name are used to locate the needed resource within the database.
- Field_name is the resource name.
- Value is the value that needed to be stored in the database.

The return type of the get method is “bit” in order to check whether the data is retrieved or not [22].

## 5.14 TLM

Transaction level Modeling TLM is used for connecting environment components with each other. UVM library is equipped with TLM library that contains analysis port, analysis export, imp port, and imp export. These elements are essential for sending and receiving transactions from one component to the other. The use of TLM facilitates the verification task by letting the designer concentrate on debugging the transaction itself rather than getting immersed with each signal in the DUT. UVM TLM consists of TLM1, TLM2, Sequencer_Port, and Analysis:

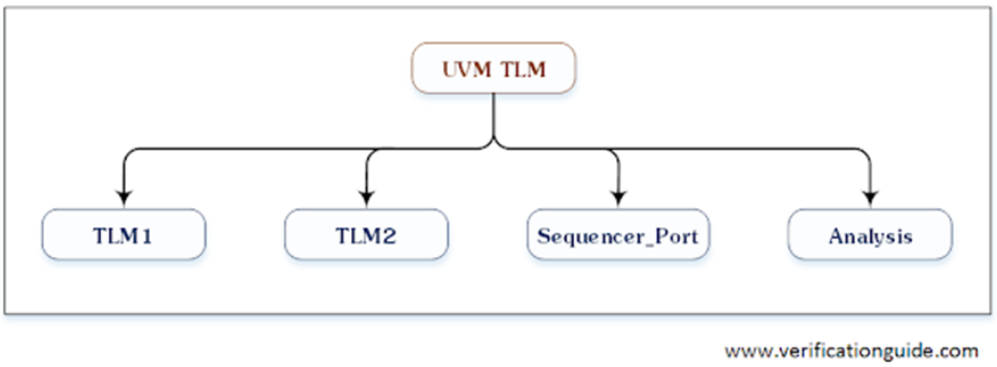


Figure 16. UVM TLM types [23]

- TLM1: it provides blocking and non-blocking transaction-level interfaces that are passed by value.
- TLM2: it provides sockets with blocking and non-blocking transaction-level interfaces.
- Sequencer_Port: it’s a pull/push port

- Analysis: it's used for the non-blocking broadcast of the transactions between UVM components.

Blocking means that the methods will not return until the transaction has been sent or received successfully while non-blocking means that the methods will convey the transaction without consuming simulation time [23].

## 5.15 UVM Factory

In UVM, the factory provides a way of overriding components and objects, on-demand, without manually changing the testbench code. The features of the UVM factory add to the flexibility and reusability of the testbench. This "overriding" happens by changing the object/component class, during its construction, by another class that is derived from the original. Overriding using the factory can be done either by instance name or by class type (which changes all instances within the scope of this override). For the override to be successful, the derived class must be extended from the original class; the factory utilizes polymorphism to override the original class and keep the whole testbench functional without changing other classes of the environment. To use the UVM factory, three steps are systematically done: registration, construction, and creation. To register a component or object in the UVM factory, the registration macros must be added to the class definition. These macros are `uvm_object_utils for objects, and `uvm_component_utils for components. Also, the class definition must include a default constructor method so that the factory calls it while creating the class. The last thing is creation; components and objects that are registered in the factory are produced by a "create" method instead of calling "new". This is where the factory does its work before returning the required class (i.e., the original or a derived one). Components are created in the build phase of the higher component in the testbench hierarchy. On the other hand, objects are created when needed during the build or the run phase [22].

## 5.16 UVM Phases

The UVM phases make the testbench flow systematic by dividing the simulation into three main parts: build phases, runtime phases, and cleanup phases. There are virtual

methods in the uvm_component class definition that corresponds to different UVM phases. The testbench developer overrides these methods appropriately in each component (e.g., the overridden phasing methods in the environment class will typically differ from the driver class) [22].

The flow of the testbench starts with the build phases, which are functions (consume no simulation time). The build phases are: the build, connect, and end_of_elaboration phases. The build phase is used for constructing the components, usually using the factory to configure the construction per need, making the testbench hierarchy in a top-down way. The connect phase, which works in a bottom-up fashion, is used to handle TLM ports/exports connections. The end_of_elaboration phase can be used as a final way to make modifications, but it is usually not necessary [22].

Following the build phases are the run-time phases which are used for stimulus application and DUT response monitoring and evaluation. They start with the start_of_simulation phase which is followed by the run phase and the other phases (from pre_reset to post_shutdown phase) which execute in parallel with the run phase. The start_of_simulation can be used for displaying information about the testbench such as its hierarchy; it executes within components in a bottom-up way. The run phase, typically essential for transactors (driver, monitor), is a task where stimuli are generated and DUT response is monitored [22].

The last part of the phasing is the cleanup phases. These phases are implemented as functions and are composed of: extract, check, report, and final phase. They are used for processing and presenting the output of the testbench (information generated by the scoreboard and subscriber) [22].

# Chapter 6

# The DDR5 PHY

## 6.1 Overview of the DRAM system and how it works

DRAM, an acronym for Dynamic Random Access Memory, is the main memory for our PCs. It's an external module by definition hence, additional care should be given to implementation effects such as clocking and synchronization, signal integrity, packing, and Pins. A DRAM cell consists of a single transistor and capacitor. It's called dynamic because the capacitor is leaky and must be refreshed periodically. DRAM chip consists of arrays of DRAM cells where each cell can be accessed by a specific row address and column address. DRAM chips can be characterized by their organization for example there are x2, x4, x8, x16, and x32 organizations. The number that follows "x" represents the number of arrays; which means the number of read/write bits for a single read/write command. Moreover, DRAM ships consist of a number of banks where the bank represents a set of memory arrays that operate independently of other sets. The main usage of banks was to double or quadruple DRAM bandwidth without needing to increase the bandwidth of the device itself. This can be done by using the concept of interleaving; if a single bank needs 10ns to transmit the data, 2 banks will need only 5ns. DRAM manufacturers further extend the level of parallelism by connecting multiple DIMMs to the memory controller where each DIMM consists of one or more ranks; a rank is a group of DRAM chips that operate in unison. This technique of abstraction further increases the bandwidth and makes it possible to pipeline the access process. On the other hand, DRAM buses must satisfy JEDEC JESD79-5A standards hence, they should have data, control, address, and chip select. The address bus is responsible for delivering the row and column addresses to the DRAM; nowadays width of the address bus is 15 bits. Control signals consist of data strobes, enable signals and clocks. While these signals are connected to all DRAM chips, there is a signal called chip select that has a width equal to the number of DRAM ranks. The purpose of this signal is to select the intended rank that's responsible for handling the commands sent. The internal

memory structure consists of rows and columns where the intersection between a row and a column represents a unit cell.

The microprocessor connects to DRAM chips through a memory controller block and a physical layer block; these blocks must communicate with each other with well-known communication protocols like JEDEC JESD79-5A and DFI (explained later). The transaction life cycle between the microprocessor and a DRAM chip consists of several steps: first, the transaction is being sent from the microprocessor to the memory controller (it may be delayed) in a queue. Second, the transaction is translated to DRAM language by the controller. Then, the command is sent to the DRAM itself, and according to the command type, its interaction with the DRAM will vary. Let's try to decompose a read command in order to fully understand how DRAM operates. As discussed before, the intersection between a bit line and a word line represents a DRAM cell that's controlled with a transistor which is a switch. In order to increase DRAM response, all bit lines must be charged to half of the supply by the sense amplifier. When the word line goes high that's the switch is turned on, and the capacitor starts to change the voltage on the bit line slightly. Accordingly, the sense amplifier senses this slight change and brings the bit line to either one or zero depending on the stored capacitor voltage level. This process also charges the capacitor again that's why the read operation is a non-destructive operation. With the aid of the figure below, the read process can be better explained in detail.

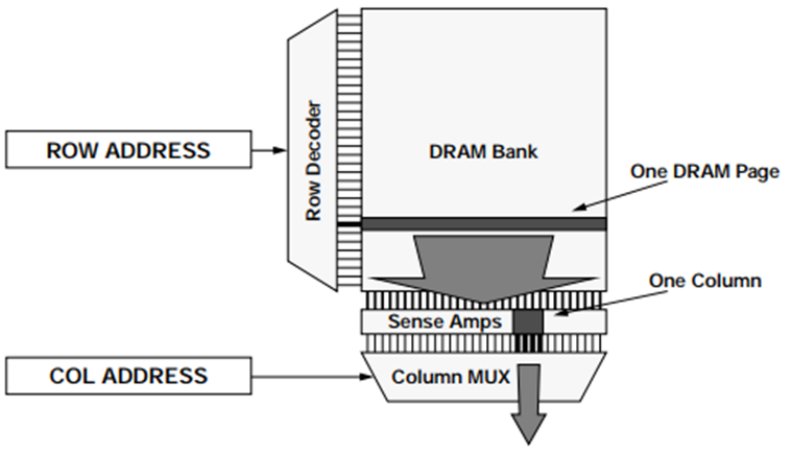


Figure 17. Accessing protocol of DRAM columns and rows [24]

At the very beginning of the process (assuming the bit line is already precharged to half of the supply voltage), the memory controller sends an activate command which specifies the chip, the bank group, the bank, and the row from which the microprocessor needs to fetch the data. As a result of the previous command, the DRAM sends the entire row that may consist of thousands of bits to the sense amplifier. This takes us to the last step which is the read command itself. This command specifies the column from which the data will be fetched. The number of output data depends on whether x2, x4, or x8 DRAM device is used as shown in the figure below

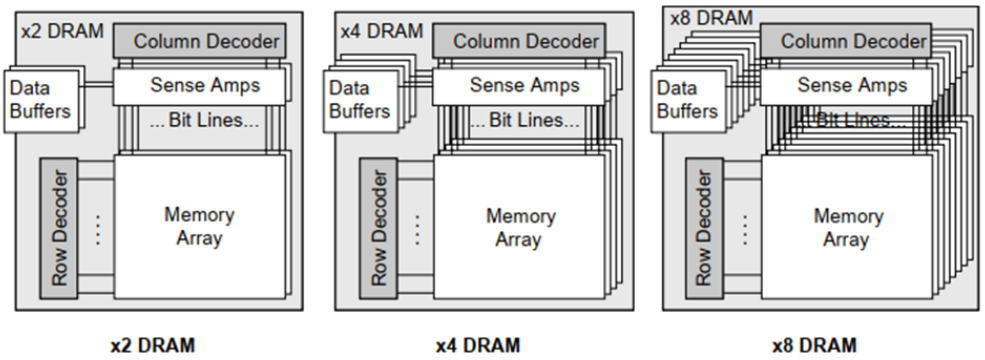


Figure 18. Logical organization of wide data-out DRAMs[24]

One of the most critical problems that DRAM manufacturers face is charge leakage. When the transistor is open there is a non-zero current value that leaks from the transistor making the capacitor loses its charge. One of the solutions that handle this problem is periodic refresh; the memory controller ensures that the leaky cell is being read and written before it totally loses its content. [24]

### 6.1.1 DRAM architecture

The memory controller contains several channels where each channel has a command/address bus that's 64-bit wide in default. We can connect only one DRAM module to a single channel. The single rank consists of several devices to satisfy the bus width. For example, we need a rank that has 8 chips with a "x8" configuration to make the output 64-bit wide. All DRAM devices inside a single rank work in lockstep that's they cannot be addressed separately. At this moment, we can talk about DRAM

generations and flavors and how each generation tries to increase the data rate without corrupting the data.

### 6.1.2 DRAM standards: Regular DDR

Synchronous DRAM (SDRAM) starts with single data rate (SDR) architectures in which IO clock frequency is the same as the internal DRAM frequency. On the other hand, double data rate architectures are then introduced to data rate independent of internal clocks. DDR as the name implies transfers the data on both the rising edge and the falling edge of the clock. Memory designers have enabled this architecture by introducing prefetching design methodology in which multiple data words are prefetched for a single read. The concept of DDR and prefetching for DDR1, DDR2, and DDR3 is better explained using the below figure:

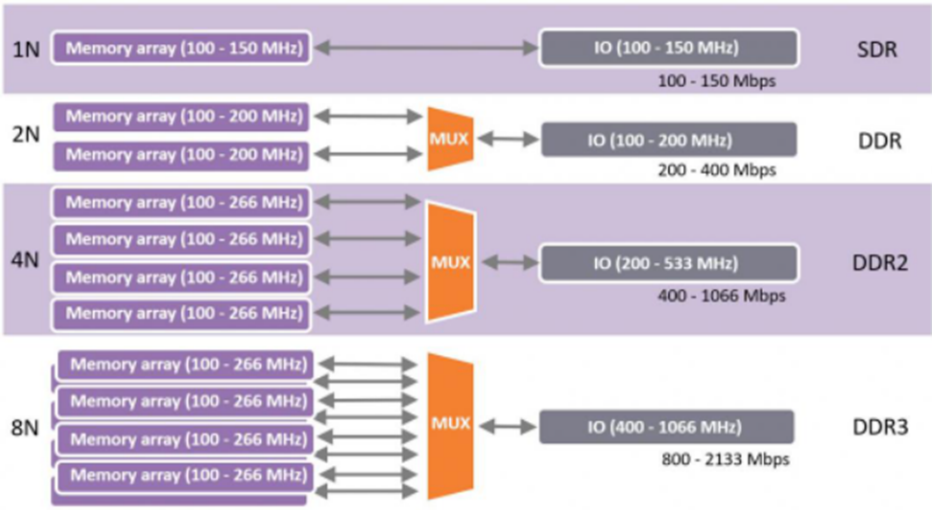


Figure 19. DDR internal and external clock frequency [25]

As explained before, for SDR, IO clock frequency is equal to that of internal DRAM frequency which is 100-150 MHz, however, DDR is sending the data on the rising and falling edge of the IO clock so it requires a prefetch buffer which size is twice of the data word. Regarding DDR2, the prefetch buffer size is 4 times the word size hence, we need IO clock frequency that's double the internal DRAM clock. For, DDR3, the same concept is applied, the IO clock frequency is guardable of the internal clock while buffer size is 8 times the word size.

DDR4 uses another concept to avoid the traditional doubling of prefetch buffer and the IO clock frequency which will transfer 16 times 64 bits that's the double basic unit of data used in processor caches. The traditional technique will waste a lot of energy and time if the second data unit isn't needed. Instead, DDR4 used the bank grouping concept as explained in detail in the following figure

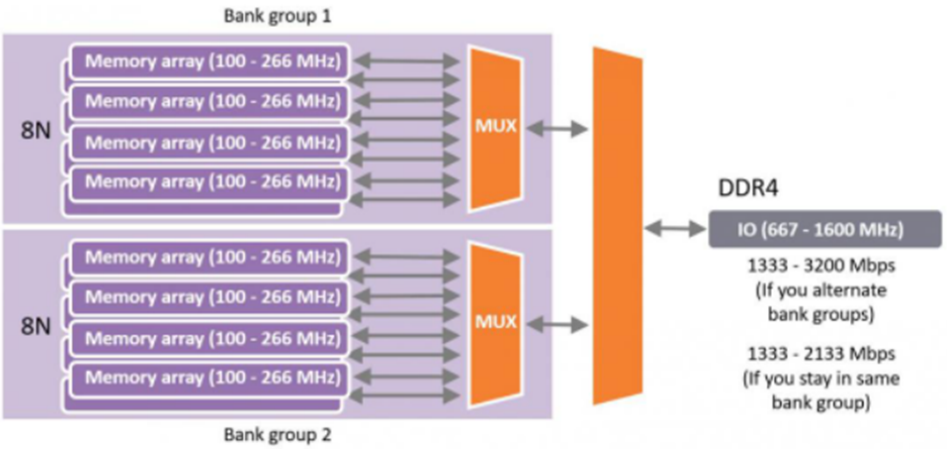


Figure 20. Bank grouping in DDR4 [25]

Instead of using a 16n buffer, DRAM manufacturers have decided to introduce multiple groups of banks with each group having an 8n buffer size with a multiplexer to select the right group. IO clock is further increased to 8 times if the memory requests are interleaved that's bank groups are accessed sequentially.

DDR5 uses another technique that's different from what is used in DDR4 which is channel splitting. In this technique, the 64bit bus is divided into two independent 32bit buses accordingly, we can increase the prefetching to 16n which is 64 bytes in reality. This technique allows further doubling of the IO clock frequency to 2133-3200 MHz. On the other hand, increasing the IO clock comes with its own drawbacks such as data integrity, power consumption, and noise performance. These drawbacks can be mitigated by using on-die termination to enhance matching between DRAM and the outside world, using differential clocking to enhance signal integrity, and finally, DRAM generations are integrated closely with the processor. [25]

The following table summarizes the comparison between SDR and all DDR generations

| Specification | DDR | DDR2 | DDR3 | DDR4 | DDR5 |
|---|---|---|---|---|---|
| Vdd | 2.5V | 1.8V | 1.5V (1.35 DDR3L) | 1.2V | 1.1V |
| Vpp | Internal | Internal | Internal | 2.5V | |
| Internal clock (MHz) | 100 - 200 | 100 - 266 (OC) | 133 – 300 (OC) | 133-300 (OC) | 133 – 200 - ... |
| IO clock (MHz) | 100 - 200 | 200 - 533 | 533 – 1200 | 1066 - 2400 | 2133 to 3200 - ... |
| Prefetch buffer size | 2n | **4n** | **8n** | 8n | 16n |
| Max transfer rate (MT/s) | 200 – 400 | 400 – 1066 | 1066 – 2400 | 2133 - 4800 | 4266 – 6400 - ... |
| Max data rate per DIMM ( GB/s) | 1.6 – 3.2 | 3.2 – 8.5 | 6.4 – 19.2 | 19.2 – 38.4 | 34.1 – 51.2 - ... |
| Number of banks | 4 | 8 | 8 | **16 in 4 groups** | 32, in x groups |
| Chip density | 256Mb – 1Gb | 512Mb - 4Gb | 1 Gb – 8Gb | 4Gb - 32Gb | 16Gb – 32Gb - .... |
| Typical module density | 1GB | 4GB | 8GB | 16GB | 32GB |
| DIMM pins | 184 | 240 | 240 | 288 | 288 |
| CMD/address bus | | | | 24bit SDR without ODT | **2x7bit DDR with ODT** |
| Channel width | 64 | 64 | 64 | 64 | **2x32** |

Figure 21. Summarizing the comparison between SDR and DDR generations [25]

## 6.2 General Circuit architecture and components of the PHY

The PHY acts, in a sense, as a bridge between the inherently analog DRAM and the inherently digital memory controller. Also, it is typically the case that the memory controller and the DRAM interfaces operate at different frequencies. For the PHY to do its job of translating the commands from the memory controller language to the DRAM language and delivering data between the two sides, it must include digital and analog circuits. According to [26], the general analog blocks of the PHY may include the following:

- Transmitter and receiver circuits: for high-speed data movement from and to the DRAM chip
- Clock domain crossing circuits: to handle working with two interfaces of different frequencies.
- Clock oscillator, clock dividers, Phase-Locked Loop, Delay Locked Loop: to produce and maintain the different clocks used by the different clock domains.
- Differential serializer: to handle the conversion from DDR to SDR.

On the other hand, the digital blocks include the following:

- Serializer/deserializers: to handle the different frequency ratio operation which leads to different bus widths.
- Command/Data datapath: to handle commands and data (FIFOs, Muxes, …)
- Embedded microcontroller: for firmware-based training in multi-standard PHYs.

Although the whole PHY, both Analog and Digital, can be modeled and simulated using VerilogA and SystemVerilog together, our work focuses on verifying the digital datapath only of the PHY.

## 6.3 DUT Properties as described in the DFI Standard

The DFI standard categorizes different signals and their corresponding timing parameters and functionality into different interface groups. Certain signals are only applicable to specific DRAM types and some are optional. The corresponding parameters must be used by the implemented DFI signals. Compliance in Signal widths, interconnect timing, timing parameters, frequency ratio, and functionality must be guaranteed to ensure correct operation between the MC and the PHY. It is noted that the DFI specification does not specify absolute latencies or a strict range of values that must be handled. Fixed values, maximum values, and constants dependent on other system settings are the different parameters types for DFI timing [2].

There are three defined clock domains: the control clock domain, the command clock domain, and the data clock domain. These clock domains use the same frequency in a matched frequency system. The system is classified as a frequency ratio system if the memory clock is higher than the MC by a factor of 2 or 4. The control clock domain of a frequency ratio system with a single memory clock, which is the case for DDR5, works at the DFI clock frequency, while the command and data clock domains operate at the higher clock ratio; this is defined as the DFI PHY clock frequency. Each clock domain's timing parameters are also defined in relation to its clock signal. The DFI clock is the MC clock whereas the DFI PHY Clock is the DRAM’s clock [2].

The DFI interfaces used by the DUT and their corresponding signals are as follows:

- Status Interface: dfi_freq_ratio

- Command Interface: dfi_reset_n, dfi_cs, dfi_address
- Read Data Interface: dfi_rddata, dfi_rddata_en, dfi_rddata_valid

The DFI read and write data bus width is twice that of the DRAM data bus in a matched frequency system. To allow the MC and PHY to transfer all of the DRAM-required data in a single DFI clock cycle in a frequency ratio system, the read and data bus will be widened proportionally to the frequency ratio; i.e., the width is multiplied by a factor of 2 in a 1:2 frequency ratio system. A single DFI command is issued per DFI clock in a matched frequency system, which is mapped to a single memory clock of the DRAM command. A DFI command is issued per phase in a frequency ratio system, where the number of command phases is equal to the clock ratio [2].

Apart from the general overall description stated above, the command and read interfaces are discussed in more detail below. First, the transmission of signals required to drive the address and command signals to the memory devices is controlled by the command interface. The signals are meant to be sent to the memory devices in such a way that the timing relationship between the signals is maintained as driven by the MC on the DFI interface. The tctrl delay timing parameter specifies the delay added between the DFI and DRAM interfaces; it is essentially the command translation delay, which must be consistent across signals of the command interface. The command interface buses/signals are duplicated into phase-specific signals which define the signal value for each phase of the DFI PHY clock in frequency ratio systems. A command sent on any of the phases must be interpreted correctly by the PHY. Furthermore, the dfi_address bus must be mapped (one to one) to the CA bus of the DDR5 DRAM; both should have the same width and the bit ordering. For the chip select (dfi_cs) signal, its polarity should follow the polarity of the corresponding memory signal (CS_n). All the timing parameters (specific to the signals used by the DUT) must be followed as described in Table 9 in the DFI standard [2].

The read interface is responsible for the data capture across the DFI interface. It handles the read transaction as follows; after exactly trddata_en cycles from issuing a read command on the command interface, the MC asserts the dfi_rddata_en signal to point out to the PHY that a read operation is underway in the memory. The duration of asserting the dfi_rddata_en signal reflects the number of contiguous cycles

expected for the read data on the dfi_rddata bus. Within tphy_rdlat cycles from asserting the dfi_rddata_en signal, the valid data transfer starts from the PHY to the MC. The PHY asserts the dfi_rddata_valid signal during the valid data transfer on the dfi_rddata bus. The dfi_rddata_valid assertion clocks have a one-to-one correspondence with dfi_rddata_en assertion clocks. The width of the dfi_rddata_en and dfi_rddata_valid signals equal the number of PHY data slices as there is a one-to-one correspondence between these signals and data slices. The timing parameters of the read interface are described in Table 15 in the DFI standard [2].

## 6.4 DUT properties as described in the JEDEC

The JEDEC JESD79-5A standard mainly describes the specs of the DRAM itself. We are only concerned with the sections that describe the DRAM's input commands and output data.

### 6.4.1 Mode Register Definition: MRR & MRW

Mode registers are used by the dram to store some configurations that affect the DRAM's operation. These registers should have default values set in the reset and initialization procedure. The values of the registers can be read using the MRR command which reads the values stored inside the mode register and outputs them on the data bus. To change the values of the registers an MRW command needs to be issued. the command itself would include the values to be written on the CA bus [3].

### 6.4.2 USED Mode Registers

There are 256 mode registers some of which are writable and others are read-only[3]. However, we are only concerned with MR0, MR8, MR40, and MR50. These are the registers that include the configuration for burst length, preamble pattern, postamble patterns, DQS offset, and CRC enable. These are the reconfigurations that would affect the operation of our DUT.

### 6.4.3 Command Truth Table & 2-Cycle Command Cancel

The JEDEC JESD79-5A standard also specifies all the possible commands and their address encoding. The commands we used were RD, MRW, MRR, DES, PREab, and

ACT. Only the PREab is a single-cycle command; all other commands are two-cycle commands [3].

Two-cycle commands require the CS_n to be low for the first cycle and high for the second cycle. If the second cycle is low the command is canceled or is the non-target rank for ODT which means the command shouldn't be executed in this rank [3].

### 6.4.4 Burst Length, Type, and Order

Burst length configuration can be changed in MR0. There are 4 different burst lengths BL16, BC8 OTF, BL32, and BL32 OTF. However, only the first three were implemented in the DUT. Burst length defines the length of data in a read operation. BL32 reads require another Dummy read operation to be sent exactly 8 cycles after.

### 6.4.5 Programmable Preamble & Postamble

Preamble and postamble configuration options are in MR8. A preamble is a pattern that appears on the DQS signal indicating the beginning of a data burst. and a postamble is a pattern indicating the end of a data burst.

### 6.4.6 Interamble

When two data bursts are near enough (back-to-back reads) there could be not enough space in between the two bursts for a full preamble and postamble patterns. In this case, the post amble is prioritized and the next burst could start with no preamble or a partial one.

### 6.4.7 Read Operation

Read operation retrieves data from the dram array. The read command includes the column to read from and there is an option on whether to use the default burst length or the burst length specified in the mode register.

## 6.5 DUT properties that are not part of the standards

The DFI standard states the following: "The DFI protocol does not encompass all of the features of the MC or the PHY, nor does the protocol put any restrictions on how the MC or the PHY interface to other aspects of the system" [2]. Therefore, any design can possess features that are not outlined in the DFI standard. Among those

features is an input enable signal that must be asserted before the PHY can operate, and an input CRC_enable signal (phycrc_mode_i) that should be asserted if CRC is implemented by the PHY and be de-asserted otherwise. An extra output signal indicates the validity of the driven values on the CA bus (CA_VALID_DA_o). Another notable difference between the standard and the DUT is the use of the dfi_freq_ratio signal. In the standard, it is used in the initialization and frequency-change protocols during which it indicates a certain frequency ratio; however, the signal conveys the same message to the DUT without following the protocols outlined in the DFI standard.

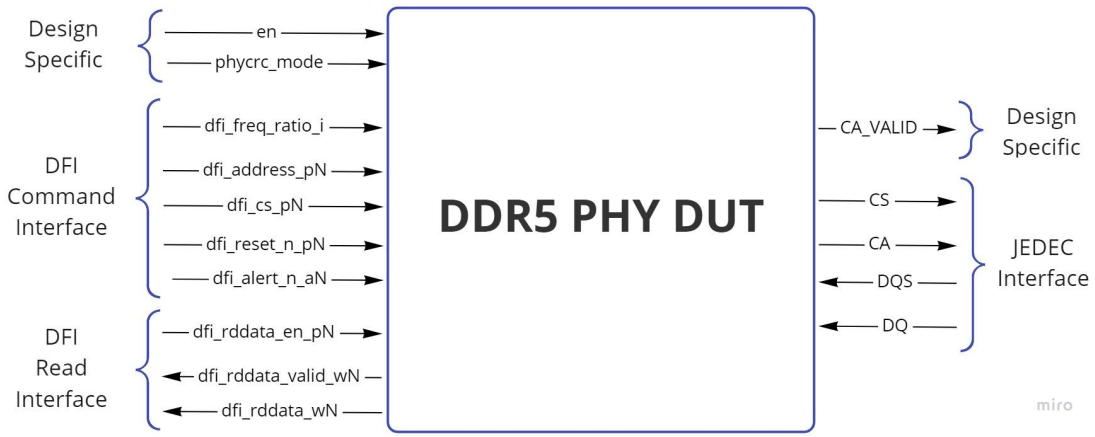


Figure 22. Black-box diagram of the DDR5 PHY DUT

# Chapter 7

# Project Design

## 7.1 Project purpose and constraints

The **project purpose** is to verify a DDR5 PHY design following the full digital verification flow of

- *feature extraction*:
  This step involves extracting the features specified in the JEDEC JESD79-5A and DFI standards, so in the subsequent steps, the design is verified to support them correctly.

- *Verification plan*:
  For each feature extracted, the following is specified:

  - ***Test generation plan***: to describe how this feature would be tested
  - ***Checking plan***: identifying what needs to be checked with respect to the reference model developed.
  - ***Assertion plan***: identifying the timing parameters between the signals that must be preserved.
  - ***Coverage plan***: defines the cover bins and groups that this feature would hit when tested. It abstracts the feature into the coverage plan to easily monitor if this feature is tested or not when random testing is used.

  Noting that it is not obligatory to fill all plans for each feature; only the relevant plans for that feature are required.

- **Test plan**:

In this step, the test scenarios and constraints are specified to ensure that all features are tested.

- **Testbench development**:
  During this phase, the testbench and test scenarios codes are written with reference to the verification and test plan.

**Project constraints**

Due to the complexity of the design and the high cost of fixing the bugs after design, the verification process is extremely time-consuming, so the first project constraint is time. In our case, the team is constrained to finish the verification process by the end of the academic year.

As the goal of the verification is to build confidence that the design implementation functions as intended in the specifications, the second constraint is to achieve high functional and code coverage (> 95%).

The third constraint is using the different industry-standard tools and techniques correctly during the verification process such as checkers, assertions, and coverage collection.

## 7.2 Project technical specifications

For each step in the digital verification flow, there are specifications to ensure that the following steps are done correctly and without any missing information. These specifications are as follows:

- For the features extraction: all the PHY features specified in the JEDEC JESD79-5A and DFI standards should be included along with the implementation-specific specifications.
- For the verification plan: all the extracted features should be addressed correctly using checkers, assertions, and coverage.
- For the test plan: the test scenarios should be sufficient to achieve high functional and code coverage.

- For the testbench development: the codes should follow the OOP principles and UVM best practices for reusability and code performance.
- For the bug report: it should be clear and a way to reproduce the bug must be accurately given.
- For the overall verification result: the level of confidence in the design working features should be high, and the outlined bugs are actually flaws in the design, not the testbench.

## 7.3 Design alternatives and justification

There are many alternatives to verify the design; these alternatives differ in performance during different stages and differ with respect to important parameters such as scalability, reusability, flexibility, and ease of maintenance. Considering testing methodologies, there are two options: directed testing and random testing. The main difference between the two methodologies is that in direct testing, tests are written for the bugs expected to be in the design while random testing can find unexpected bugs in the design. The figure below shows the coverage of each testing methodology over total design space and features [19].

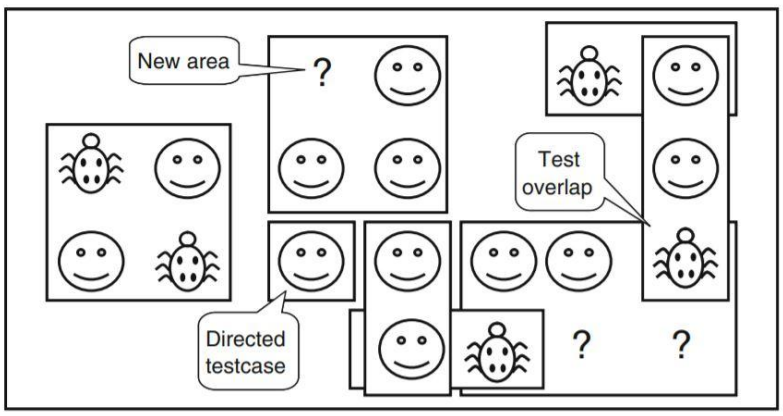


Figure 23. random vs direct testing coverage [19]

Regarding random testing, the automatically generated stimuli require developing a reference model to automatically predict the results. This additional infrastructure takes a huge amount of time during the testbench development while the directed testing requires little infrastructure. After developing the testbench, directed testing

provides immediate, steady progress but with slow progress while random testing is much faster to achieve 100% coverage. The figure below illustrates these two differences [19].

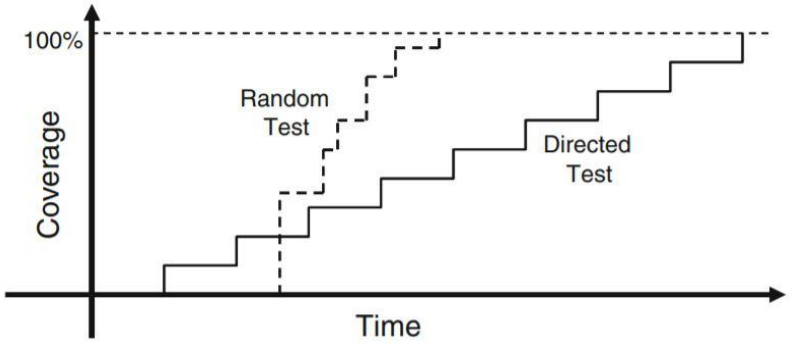


Figure 24. random vs direct testing progress over time [19]

Another two alternatives appear when considering the scalability and reusability of the testbench. The two alternatives are building Systemverilog testbench or UVM testbench. In a SystemVerilog testbench, all classes are built manually from scratch while a base class library (BCL) is used in a UVM testbench. This UVM testbench has many advantages over the SystemVerilog testbench such as separating tests from testbenches, simplifying the configuration of objects, modularity, and reusability.

An alternative to SystemVerilog altogether is cocotb, which is a testbench environment that implements the concepts found in UVM, but it is written in Python. Although Python promises ease of code development, it is not as well-established as UVM. Furthermore, UVM has the advantage of being written in the same language as the design, which makes it easier to debug the testbench and design together.

## 7.4 Description of the selected design

Our selected testbench would mainly depend on employing the coverage-driven random testing methodology to rapidly achieve high coverage. As the ability to achieve coverage diminishes over time, directed testing would be used to achieve 100% coverage by hitting the holes left by random tests. The figure below shows the coverage-driven constrained random verification methodology used [19].

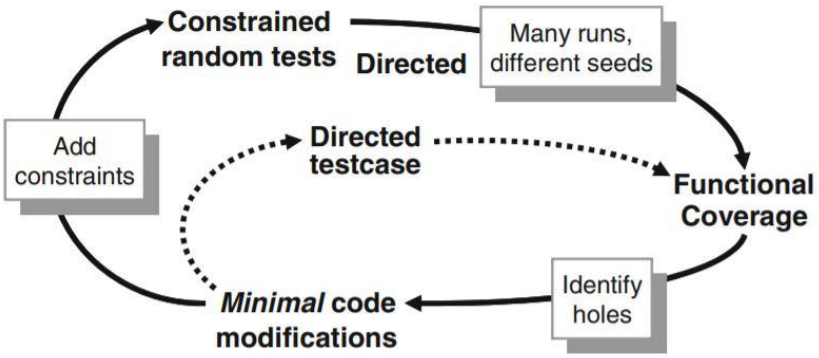


Figure 25. coverage convergence flow [19]

For the testbench code development, a UVM testbench would be built to exploit its many advantages that were mentioned before. In addition, a SystemVerilog assertions module is built to implement signal-level checkers that are described in the standard but are not captured in the transaction-level checkers in the UVM scoreboard.

## 7.5 Block diagram and functions of the subsystems

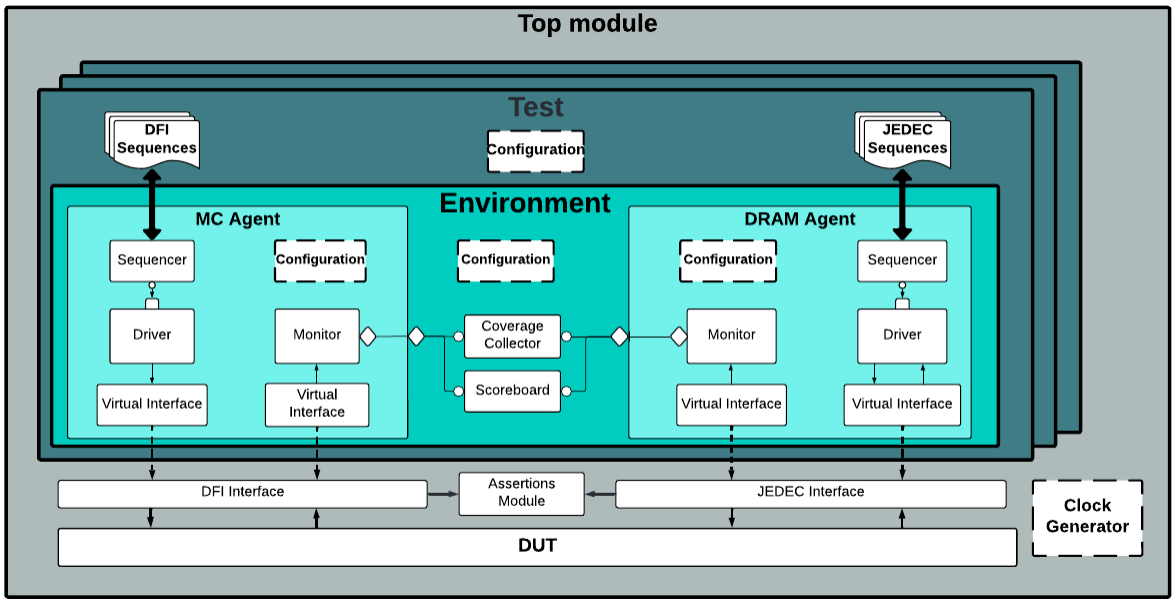


Figure 26. testbench architecture for the DDR5 PHY

The figure above shows the block diagram of the selected testbench. In it, there are two interfaces: DFI and JEDEC, and there is an agent for each interface. Depending on which interface is being tested in a specific test case, the configuration object

would be used to specify the parameters needed by the environment to act according to the standards. The following is a brief description of each class:

1- **Sequence item**: maps the pin-level signal into a higher level of abstraction. It encapsulates the information needed for the stimulus and response on the DFI and JEDEC interfaces.

2- **Sequence**: Defines the sequence of generating the sequence_items. Different sequence classes are built for both interfaces implementing different scenarios.

3- **Sequencer**: Deliver the sequence_item generated in sequence to the driver.

4- **DRAM Driver**: Drives the sequence_item from the sequencer to the DUT's JEDEC interface at the pin level.

5- **MC Driver**: Drives the sequence_item from the sequencer to the DUT's DFI interface at the pin level.

6- **MC Monito**r: Monitors the pin-level activity of the DUT's DFI interface and converts it to a packet-level activity.

7- **DRAM Monito**r: Monitors the pin-level activity of the DUT's JEDEC interface and converts it to a packet-level activity.

8- **DRAM Agent**: Contains the sequencer, driver, and monitor related to the JEDEC interface.

9- **MC Agent**: Contains the sequencer, driver, and monitor related to the DFI interface.

10- **Scoreboard**: Compares the outputs received from the monitor with the expected values.

11- **Coverage collector**: defines and measures the functional coverage.

12- **Environment**: Contains the agents, coverage collector, and scoreboard.

13- **Test**: contains the sequences of the test case and environment.

14- **Clock generator**: generates the clock.

15- **Configuration**: specifies the attributes of the classes such as the burst length of read operations that are done by the DRAM driver

16- **DDR Assertions module:** contains the SVA properties and assert and cover statements for the DFI and JEDEC signals.

17- **Top module**: connects the testbench, DUT, and assertions module.

# 7.6 Verification Plan

The verification plan specifies what must be verified in a hardware design, as well as the assertions and coverage criteria that must be established and met in order to progress to the next step in the design flow.

The verification plan consists of four important aspects:

- Design requirements and features.
- Stimulus generation.
- Coverage measurements.
- Response checking and assertion

## Design requirements and features

This document contains DFI and JEDEC design requirements along with the associated scope/interface, the feature name, associated signals, feature description, and the feature page in the standard. A snippet from our verification plan is shown in the below figure:

| ID | Scope/Interface | Feature Name | ssociated Signals (Parameters | From/Defined by | Design Requirement/Specs. Description (From Standard) | Reference (Section & Page) |
|---|---|---|---|---|---|---|
| DFI_DR_1 | Command Interface | Min clks MC holds CA | dfi_address_pN - dfi_cs_pN | MC | For DDR5, the MC must hold the CA signals for at least 1 DFI PHY clock cycle after the CS is active. There | S3.1 - P33 - T8 |
| DFI_DR_2 | Command Interface | DFI address bus width | dfi_address_pN | MC | DFI address bus. These signals define the address information. For DDR5, the CA bus is an SDR (single da | T8 & P33 |
| DFI_DR_3 | Command Interface | DFI address bus bit ordering | dfi_address_pN | MC | The PHY must preserve the bit ordering of the dfi_address signals when it sends this data to the DRAM dev | T8 - P34 |
| DFI_DR_4 | Command Interface | Error indicator | dfi_alert_n_aN | PHY | This signal is driven when a CRC or command parity error is detected in the memory system. The PHY is re | T8 - P34 |
| DFI_DR_5 | Command Interface | Error indicator | dfi_alert_n_aN | PHY | The PHY holds the current state until the PHY error input transitions to a new value | T8 - P34 |
| DFI_DR_6 | Command Interface | Error indicator pulse width. | dfi_alert_n_aN | PHY | The pulse width of the dfi_alert_n signal matches the pulse width of the DRAM subsystem error signal, plus | T8 - P34 |
| DFI_DR_7 | Command Interface | DFI chip select. | dfi_cs_pN | MC | This signal defines the chip select. The polarity of this signal is defined by the polarity of the corresponding | T8 - P35 |
| DFI_DR_8 | Command Interface | DFI reset bus. | dfi_reset_n_pN | MC | These signals define the RESET. | T8 - P35 |

Figure 27. A snippet of design requirement and feature section of the verification plan

## Coverage Measurement

The verification scopes are given in the coverage measurement part of the verification plan. It is the most important section because it lets the verification designers know how much functionality the testbench covers. It also gives us information about what's remaining by means of coverage analysis. This section describes the functional coverage and what must be done to fully cover all DUT functionality. A snippet from the coverage plan is shown in the below figure:

| ID | Scope/Interface | Feature Name | Associated Signals (Parameters) | om/Defined | Design Requirement/Specs. Description | Generation Plan |
|---|---|---|---|---|---|---|
| DFI_DR_1 | Command Interface | Min clks MC holds CA | dfi_address - dfi_cs | MC | For DDR5, the MC must hold the CA signals for at least 1 DFI PHY clock cyc | Drive dfi_address and dfi_cs |
| DFI_DR_2 | Command Interface | DFI address bus width | dfi_address | MC | DFI address bus. These signals define the address information. For DDR5, the | Drive dfi_address |
| DFI_DR_3 | Command Interface | DFI address bus bit ordering | dfi_address | MC | The PHY must preserve the bit ordering of the dfi_address signals when it sen | Drive dfi_address |
| DFI_DR_4 | Command Interface | Error indicator | dfi_alert_n | PHY | This signal is driven when a CRC or command parity error is detected in the m | ~~Send erroneus data in Write command~~ |
| DFI_DR_5 | Command Interface | Error indicator | dfi_alert_n | PHY | The PHY holds the current state until the PHY error input transitions to a new | ~~Send erroneus data in Write command~~ |
| DFI_DR_6 | Command Interface | Error indicator pulse width. | dfi_alert_n | PHY | The pulse width of the dfi_alert_n signal matches the pulse width of the DRAM | ~~Send erroneus data in Write command~~ |
| DFI_DR_7 | Command Interface | DFI chip select. | dfi_cs | MC | This signal defines the chip select. The polarity of this signal is defined by the | Drive dfi_cs (active low) |
| DFI_DR_8 | Command Interface | DFI reset bus. | dfi_reset_n | MC | These signals define the RESET. | Drive dfi_reset_n (active low) |
| DFI_DR_9 | Command Interface | DFI reset bus bit ordering | dfi_reset_n | MC | The PHY must preserve the bit ordering of the dfi_reset_n signals when it sen | Drive dfi_reset_n and dfi_address |
| DFI_DR_10 | Command Interface | DFI reset bus. | dfi_reset_n | MC | The MC is expected to drive the dfi_reset_n signal with appropriate value as p | Drive dfi_reset_n |
| DFI_DR_11 | Command Interface | MC Commands issuing | dfi_address - dfi_cs | MC | The MC may issue commands on any phase to communicate with the PHY. F | Drive dfi_address_pN and dfi_cs_pN |
| DFI_DR_12 | Command Interface | Ctrl delay Timing Parameter | (tctrl_delay) | PHY | Specifies the number of DFI clock cycles from the time that any command su | Drive the command interface signals |

Figure 28. Snippet of coverage measurement section of verification plan

## Stimulus generation:

The stimulus generation section is in charge of generating the input test vector needed to completely exercise the DUV and display all of its possible behaviors. This entails not only creating valid test vectors to demonstrate that the device is functioning as planned, but also creating faulty test vectors to force the device into unexpected situations. The goal of stimulus creation is to provide test vectors that allow for a high level of coverage. A snippet from the code can be is shown in the below figure:

| ID | Scope/Interface | Feature Name | ed Signals (Par | Defin | Design Requirement/Specs. Description | Checking/Assertion Plan |
|---|---|---|---|---|---|---|
| DFI_DR_1 | Command Interfac | Min clks MC holds CA | dfi_address - dfi_cs | MC | For DDR5, the MC must hold the CA signals for at least 1 DFI PHY clock cycle after the | # Assert that the command is driven for at least 1 DFI PHY clock cycle after the CS is active. |
| DFI_DR_2 | Command Interfac | DFI address bus width | dfi_address | MC | DFI address bus. These signals define the address information. | # Check the CA singal bit by bit compared to dfi_address bus.<br># Assertion: tcmd_lat specifies the number of DFI clocks after the dfi_cs signal is |
| DFI_DR_3 | Command Interfac | FI address bus bit orderir | dfi_address | MC | The PHY must preserve the bit ordering of the dfi_address signals when it sends this data to | # Check that CA bits are in correct order |
| DFI_DR_4 | Command Interfac | Error indicator | dfi_alert_n | PHY | This signal is driven when a CRC or command parity error is detected in the memory<br>The PHY holds the current state until the | # Check that PHY asserts dfi_alert_n for errors |
| DFI_DR_5 | Command Interfac | Error indicator | dfi_alert_n | PHY | PHY error input transitions to a new value<br>The pulse width of the dfi_alert_n signal matches the pulse width of the DRAM subsystem error signal, plus or minus | # Assert that dfi_alert_n always reflects the DRAM status (DRAM error flag) + or - synchronization cycles<br># Assert the width of dfi_alert_n always reflects the DRAM status (DRAM error flag) + or - synchronization cycles |
| DFI_DR_6 | Command Interfac | DFI chip select. | dfi_cs | MC | This signal defines the chip select. The polarity of this signal is defined by the polarity of the corresponding memory signal. [Active low: CS_n] | # Assert the correct behaviour of CS_n on the DRAM interface |
| DFI_DR_7 | Command Interfac | DFI reset bus. | dfi_reset_n | MC | These signals define the RESET. | # Assert the reset_n signal is drived accordingly |

Figure 29. A snippet of stimulus generation section of verification plan

## Response checking:

The response checking section is responsible for making sure that the DUT responses meet the correct behavior according to standard specification. This can be done by an approach called reference model check. A snippet from the response checking and assertions is shown in the below figure:

| ID | :ope/Interfa | Feature Name | Signals (if | Defir | Design Requirement/Specs. Description | Coverage plan |
|---|---|---|---|---|---|---|
| DFI_DR_1 | mmand Interf | n clks MC holds | address - dfi | MC | For DDR5, the MC must hold the CA signals for at least 1 DFI PHY clock cycle after the CS is active. There is no | # Assertion coverage: dfi_cs (1=>0=> 1) & dfi_address valid |
| DFI_DR_2 | mmand Interf | address bus wid | dfi_address | MC | DFI address bus. These signals define the address information. | # Assertion coverage: dfi_cs (1=>0=> 1) & dfi_address valid |
| DFI_DR_3 | mmand Interf | ddress bus bit or | dfi_address | MC | The PHY must preserve the bit ordering of the dfi_address signals when it sends this data to the DRAM devices. | # Cover point: cover all valid values of dfi_address; i.e., all commands [a checker will make sure dfi_address = CA] |
| DFI_DR_4 | mmand Interf | Error indicator | dfi_alert_n | PHY | This signal is driven when a CRC or command parity error is detected in the memory system. The PHY is not required to distinguish between a CRC and command parity error. | # Coverpoint: cover all values of dfi_alert_n (bins: err , no_err) |
| DFI_DR_5 | mmand Interf | Error indicator | dfi_alert_n | PHY | The PHY holds the current state until the PHY error input transitions to a new value<br>The pulse width of the dfi_alert_n signal matches the pulse width of the DRAM subsystem error signal, plus or minus synchronization cycles. | # Assertion coverage |
| DFI_DR_6 | mmand Interf | DFI chip select. | dfi_cs | MC | This signal defines the chip select. The polarity of this signal is defined by the polarity of the corresponding memory signal. [Active low: CS_n] | # Assertion coverage: asserting that CS_n follows dfi_cs behaviour |
| DFI_DR_7 | mmand Interf | DFI reset bus. | dfi_reset_n | MC | These signals define the RESET. | # Cover both values of dfi_reset_n |

Figure 30. Snippet of response checking section of verification plan

# Chapter 8

# Project Execution

In this chapter, a thorough explanation of the testbench components is presented with proper figures and code snippets for illustration. As for the DDR5 PHY, it has two different interfaces; DFI and JEDEC, that interact with the Memory Controller and the DRAM respectively. The purpose of the verification environment is to properly stimulate the DUT interfaces, monitor its behavior, check for erroneous behavior and collect coverage information.

Exploiting the concepts of OOP like inheritance and polymorphism ensures the verification components' reusability and configurability, thereby, reducing the amount of effort needed to create a whole new testbench. In order to make tb scope available to the build process, a package "`tb_pkg.sv`" is created to include all tb classes and enum declarations.

## 8.1 Top Module

The top-level module "`top_testbench.sv`" is responsible for the integration of the DUT with the testbench architecture classes. It instantiates the DUT, ddr_assertions module and the interfaces then connect them properly. It uses `uvm_config_db` to register the interfaces into the config database as virtual interfaces which are used by the drivers and the monitors in the environment. Additionally, the top module calls `run_test()` method to run the UVM test via +UVM_TESTNAME command line option.

The top module is responsible for importing the uvm and tb packages as well as generating the required dut clocks: `dfi_clk_i`, `dfi_phy_clk_i`. It resolves their values in compilation time depending on the frequency ratio defined.

```
module top_testbench;
        import uvm_pkg::*;
        import tb_pkg::*;
...
        parameter DFI_CLK_PERIOD = 800;
```

```
        `ifdef ratio_1_to_1
                parameter PHY_CLK_PERIOD = DFI_CLK_PERIOD;
        `elsif ratio_1_to_2
                parameter PHY_CLK_PERIOD = DFI_CLK_PERIOD/2;
        `elsif ratio_1_to_4
                parameter PHY_CLK_PERIOD = DFI_CLK_PERIOD/4;
        ...
        always #(DFI_CLK_PERIOD/2) dfi_clk_i              = ~dfi_clk_i;
        always #(PHY_CLK_PERIOD/2) dfi_phy_clk_i          = ~dfi_phy_clk_i
...
endmodule : top_testbench
```

# 8.2 Interfaces

As mentioned earlier, the interfaces are registered as virtual interfaces in the config database. This is because the virtual interfaces are pointers to the actual ones and can be used with the OOP/dynamic nature of UVM. This means that any UVM component that has access to the virtual interface (e.g. driver/monitor) can create a handle for it and have access to the static interface signals.

The DUT has two standard interfaces with different signals and clock domains; therefore, two interface blocks were implemented, namely, dfi_intf.sv & jedec_intf.sv; they declare all the signals defined by the DUT and basically by the DFI & JEDEC JESD79-5A standards and they have dfi_clk, dfi_phy_clk respectively as inputs. Furthermore, each interface contains a clocking block to regulate applying stimuli to the DUT and sampling the DUT's response in a consistent way across the environment while avoiding race conditions.

```
interface dfi_intf(input logic dfi_clk);

        logic                           reset_n_i;
        logic [1:0]                     dfi_freq_ratio_i;
        logic [13:0]                    dfi_address_p0;
...
       clocking cb_D @(posedge dfi_clk);
                //Drive on negedge -- Sample at #1step
                default input #1step output negedge;
                input dfi_rddata_w0,dfi_rddata_w1, ...;
                output reset_n_i, dfi_freq_ratio_i, dfi_address_p0, ...;
        endclocking
endinterface
```

```
interface jedec_intf(input logic dfi_phy_clk);
        logic [13:0] CA_DA_o;
```

```
        logic DQS_AD_i;
...
        clocking cb_J @(posedge dfi_phy_clk);
                default input #1step output negedge;
                input CA_DA_o, ...;
                output DQ_AD_i, ...;
        endclocking
endinterface
```

The guidelines mentioned in [27] were followed to ensure a robust testbench clocking scheme. Generally, there are three things that differ in how they are handled using the clocking block: applying the DUT stimulus, sampling the DUT inputs (applied by the driver), and sampling the DUT outputs (its response). First, the DUT stimuli should be applied on the negative edge. This avoids the activity at the positive edge and makes debugging using waveforms easier. Second, the DUT input signals are sampled at the positive edge because the monitor should sample exactly what is seen by the DUT. Third, the DUT's output is sampled when all activity in the current time step is finished. This avoids the positive edge, where outputs change, and the negative, where inputs change, producing consistent results. This is done by the SystemVerilog 1step delay in the clocking block. An illustration of the used driving and sampling scheme is shown below.

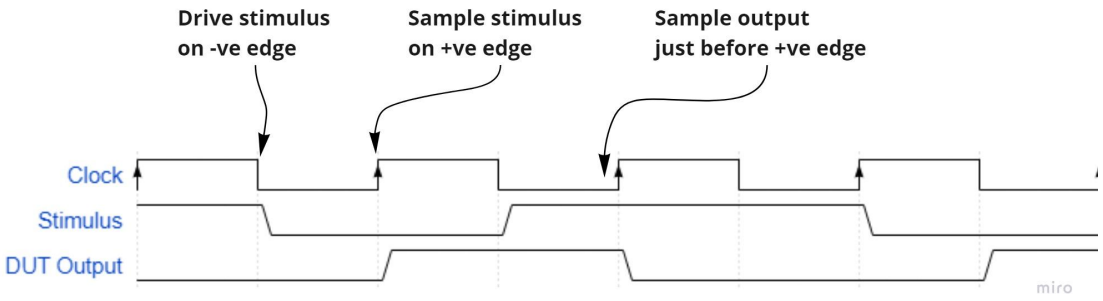


Figure 31. Driving and Sampling Scheme

## 8.3 Tests

A base test class "`base_test.sv`" is extended from `uvm_test` which is a top_level UVM component class. The base test instantiates the environment class, thus, the whole testbench components are involved in a container called environment and it can be configured or tweaked for each test. In other words, we no longer need to copy,

edit or connect the verification components in each test, but instead, the environment class takes care of all of it.

The base test, also, prints test topology, it runs the `reset` and the `base` sequences. Thus, new tests extending the base test can build targeted scenarios, and specific test cases and start different sequences.

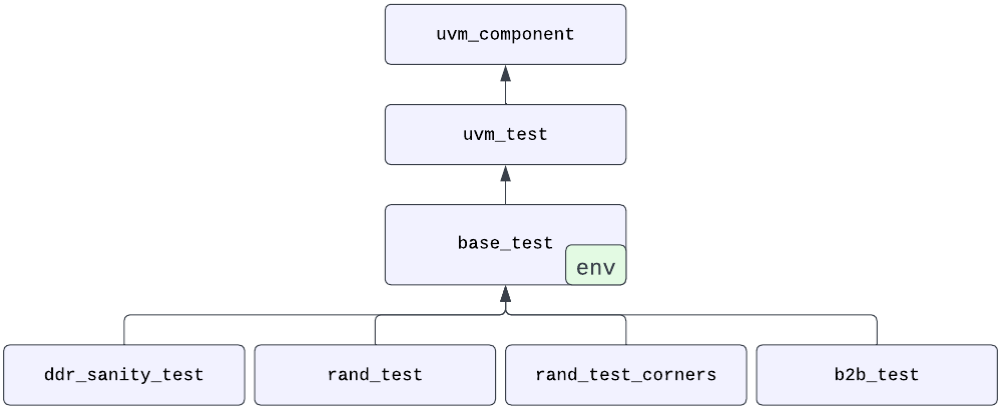


Figure 32. Different testcases inherited from base_test

As shown in figure 32, there are four independent tests inherited from the base test class, namely, ddr_sanity_test, rand_test, rand_test_corners, and the b2b (back to back) test, and each one of them fulfills different test scenarios. They are discussed in section 8.15.

# 8.4 Environment

The environment class "`env.sv`" constitutes the main architecture of the uvm testbench. It instantiates the agents (MC & DRAM), the scoreboard, and the subscriber. It uses the analysis port/imp/export of the TLM to connect these components together. The `analysis_port` of the mc_agent and the dram_agent are connected to the `analysis_imp` of both the scoreboard and the subscriber. These connections enable the monitors in both agents to broadcast their sampled transactions to the scoreboard (check for errors) and the subscriber (collect coverage).

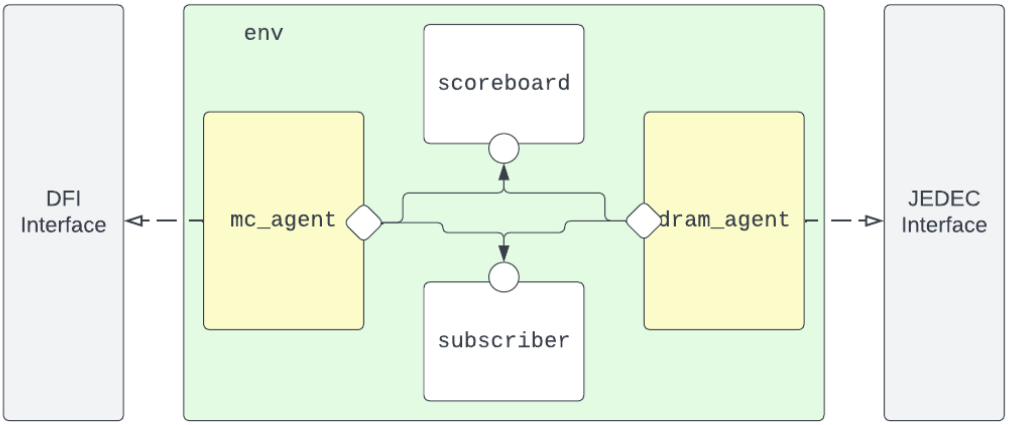


Figure 33. The env.sv class and sub-components. connections

# 8.5 Agents

There is an agent for each interface group; one for DFI (`mc_agent`) and another agent for the JEDEC (`dram_agent`). Each agent has its own driver (`mc_driver`, `dram_driver`), monitor (`mc_monitor`, `dram_monitor`) and sequencer (`mc_sequencer`, `dram_sequencer`). The agents are responsible for connecting the drivers' `seq_item_port` to the `seq_item_imp` of the sequencers. They also connect the monitors' `analysis_port` to the agents' `analysis_port`.

```
class mc_agent extends uvm_agent;
 ...
    function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        //Connect driver port to sequencer imp/export
        mc_driver1.seq_item_port.connect(mc_sequencer1.mc_seq_item_imp);

        //Connect monitor analysis port to agent analysis port
        mc_monitor1.mc_analysis_port.connect(mc_analysis_port);
    endfunction
 ...
endclass : mc_agent
```

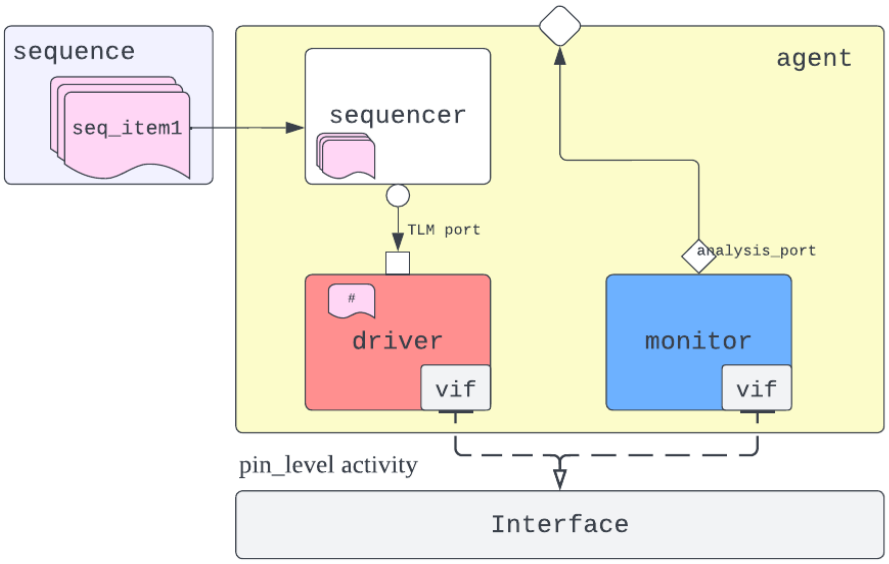


Figure 34. MC/DRAM Agent components and connections

# 8.6 Sequence Item

In order to generate stimulus to the DUT and receive its responses, an object class called `ddr_sequence_item` is created as an abstracted transaction level model. This is because the data flows within the uvm testbench components in the form of sequence_items or transactions. The class `ddr_sequence_item` extends the `uvm_seq_item` class (a `uvm_object` class), thus inheriting `uvm_object` methods, like `print(), clone(),` …etc. This class acts as a placeholder for data property members by which the driver/monitor can stimulate/interpret the pin_level activities. The `ddr_seq_item` data fields represent:

- Control information, such as, `dfi_freq_ratio, reset_n_i, en_i.`
- Payload information, such as, `CMD, data.`
- Configuration information, such as, `BL_mod, AP, OP, read_pre_amble.`
- Analysis information, such as, `MR, dfi_rddata_queue, is_data_only.`

Notice that, sequence items are incorporated in both request and response transactions. All data fields/variables used for request/stimulus to the DUT are declared as rand or randc variables, which is necessary for randomized transaction stimulus generation. On the other hand, the analysis data fields, i.e. response data

fields, obviously are not allowed to be random variables because they hold DUT responses.

The `ddr_sequence_item` defines constraints to the random variables to ensure the values of the transaction are within the legal range or desired bounds. These constraints can be extended, but not overwritten/violated, when randomized with in-line constraints (in sequence classes) to help generate specific sequences.

```
class ddr_sequence_item extends  uvm_sequence_item;
        `uvm_object_utils(ddr_sequence_item)
        //==================== Data abstraction ====================//
        rand bit [2*device_width-1:0]   data ;                  //data
        byte                            MR [50:0] ;             //MRs
...
        //==================== dfi_address/CA abstraction ===========//
        rand  command_t                 CMD;
        randc bit               [1:0]   BA;           //Bank Address
        randc bit               [2:0]   BG;           //Bank Group
        randc bit               [3:0]   CID;
        randc bit               [17:0]  ROW;          //Row Address
...
        //=========================================================//
        //                       Constraints                       //
        //=========================================================//
        constraint c_MRA {MRA inside {8'h00, 8'h08, 8'h32};}
        constraint c_command_cancel {command_cancel dist {0:/95, 1:/5};}
...

endclass : ddr_sequence_item
```

# 8.7 Sequences/Sequencers

The sequences are object classes derived from the `uvm_sequence` class. Their main purpose is to generate sequence_item objects or start other sequences and send them to the driver through the sequencer component. Conversely, when the DUT generates a response, the driver creates a response sequence item and passes it back to the the same sequence object, again through the sequencer. A base sequence (`base_seq`) is created extending `uvm_sequence` class with transaction type `ddr_seq_item`.In this work, the base sequence contains empty virtual body methods which are overwritten in other children sequences.

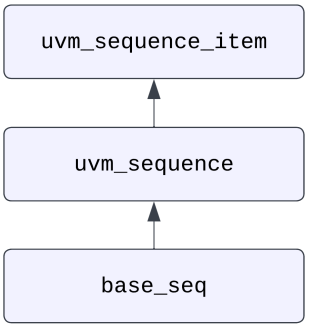


Figure 35. The base_seq inherits uvm_seq, and it is the parent of all other sequences

```
`include "base_seq.sv"
`include "reset_seq.sv"
`include "ddr_sanity_seq.sv"
`include "ACT_seq.sv"
`include "RD_seq.sv"
`include "MRW_seq.sv"
`include "MRR_seq.sv"
`include "PREab_seq.sv"
`include "DES_seq.sv"
`include "dram_resp_seq.sv"
`include "rand_seq.sv"
`include "rand_seq_corners.sv"
`include "b2b_seq.sv"
```

A Snippet from `sequence_lib.sv` file listing all sequences

The base sequence is essential is a particular set of features or configurations need to be inherited in the extending sequences. In our case, we just need to define common data types that are used to build the other random sequences.

The fact that sequences and sequence items are object classes means that they can be randomized and easily manipulated in order to derive meaningful test scenarios. The only difference between a sequence and a sequence item is that the sequence has body methods (i.e. `pre_body, body, post_body tasks`) which are used to create and execute sequence items or/and other sequences objects.

A typical sequence is executed by the run phase of a test targeting a particular sequencer. The sequencer is responsible for sending the sequence_items/transactions inside the sequence `body` task to the driver.There are two sequencers in this testbench, `mc_sequencer` and `dram_sequencer`, one for each agent and they are

connected to the corresponding driver by a TLM port in a one-to-one handshake mechanism.

```
class base_test extends uvm_test;
      virtual task run_phase(uvm_phase phase);
           reset_seq_inst = reset_seq::type_id::create("reset_seq_inst");
           base_seq_inst = base_seq::type_id::create("base_seq_inst");
                super.run_phase(phase);
                phase.raise_objection(this);
                     reset_seq_inst.start(env1.mc_agent1.mc_sequencer1);
                     base_seq_inst.start(env1.mc_agent1.mc_sequencer1);
                phase.drop_objection(this);
        endtask

endclass : base_test;
```

For example, the base test creates and starts the `reset_seq` and the `base_seq`, passing the handle of the `mc_sequencer` as an argument to the `start` task. By calling reset sequence `start()` method, the test executes the `pre_body`, `body` and `post_body` sequence tasks in that order. In the code snippet below, the `pre_body` task creates a `ddr_seq_item` object called `item1`, and in turn it is passed as an augment to the `start_item` and `finish_item` methods in the `body` task.

```
class reset_seq extends base_seq;
        `uvm_object_utils(reset_seq)
        task pre_body();
                item1 = ddr_sequence_item::type_id::create ("item1");
        endtask
        task body();
                start_item (item1);
                        item1.reset_n_i = 0;
                        item1.en_i      = 0;
                finish_item (item1);

                start_item (item1);
                        item1.reset_n_i = 1;
                        item1.en_i      = 1;
                finish_item (item1);
        endtask

endclass : reset_seq
```

To sum up, we want to send two transactions (as shown in the above code snippet) to the `mc_driver` through the `mc_sequencer`. In the reset sequence case we don't need randomization, the transactions are very simple, the first one activates the reset signal (active low) and the consecutive one deactivates it. Thus if the driver requests a transaction from the sequencer every clock cycle, this sequence will be translated to a

reset signal asserted low for a clock cycle to perform DUT reset. There are other sequences that perform curated test scenarios explained in section 8.15.

## 8.8 MC Driver

The MC (Memory Controller) Driver is responsible for converting the dfi transactions /sequence_items, coming from the sequencer, into pin-level activities at the dfi through the dfi virtual interface "`dfi_vif`". MC Driver is an abstract way to mimic the behavior of memory controller stimulus. For example, if the driver receives a READ command transaction, it will perform the stimulus protocol in which the driver controls the `dfi_address` bus, `dfi_cs` and `dfi_rddata_en` with proper stimulus, frequency ratio/phasing and timing according to the DFI & JEDEC JESD79-5A standards. Thus, the DUT receives the pin-level activity of a read command.

The `mc_driver` extends `uvm_driver` class, a uvm component. The driver receives transactions from the `mc_sequencer` which is accomplished through the special driver-sequencer one-to-one TLM handshake mechanism, see section 5.9. In the pre_reset phase, the driver set the initial values to the signals, such that, all input signals to the interface has a known initial value at the beginning of the simulation. In the run phase [Fig. 37], a forever loop calls the API methods `get_next_item()` to request a new transaction from the sequencer and `item_done()` to indicate that the transaction is consumed and the driver is ready to get a new one if available. Typically, the driver core logic is executed between calling of these two methods, a task/function or a group of tasks/functions are called to execute the protocol of converting the transaction into pin-level activity driven to the dut. In the case of MC Driver, task `drive_dfi()` is the main task and it calls other tasks within itself.

```
class mc_driver extends  uvm_driver#(ddr_sequence_item);
        `uvm_component_utils(mc_driver)
        static logic [13:0]                             CMD_queue [$];
        static logic [Physical_Rank_No-1:0]             CS_queue [$];

        static logic                                    rddata_en_queue [$];
...
        task run_phase(uvm_phase phase);
                super.run_phase(phase);
                forever begin
                        seq_item_port.get_next_item(dfi_item1);
```

```
                                drive_dfi();
                            seq_item_port.item_done();
                    end
            endtask
            //=============================================================//
            //                      MC Tasks Prototypes                    //
            //=============================================================//
            extern task drive_dfi ();
            extern task num_of_words();
            extern task decode_cmd();
            extern task ACT_cmd ();
            extern task RD_cmd ();
            extern task MRW_cmd ();
            extern task MRR_cmd ();
            extern task DES_cmd ();
            extern task PREab_CMD ();

endclass : mc_driver
```

As shown in the code snippet above, there are nine implemented tasks inside the mc driver in which `drive_dfi()` is the main task. `num_of_words()` task deals with the burst length setting, indicating how many cycles the read data enable signal should remain asserted high for a RD (read) or MRR cmd. The other seven deal with commands transaction-to-signal level realization (`ACT_cmd`, `RD_cmd`, `...` etc). The following code snippets show a glance at how the driver interpret the received transactions and extract useful information. For example, the Burst length configuration is acquired from an MRW transaction (MR0, OP[1:0]), this is used to determine the number of data words later on when a read cmd is received. The no of words is saved in a static int variable called `n_words`, and then used to fill in the `rddata_en_queue` information inside `drive_dfi()` task.

```
task mc_driver::num_of_words();
        if ((dfi_item1.CMD == MRW) && (dfi_item1.MRA == 0) &&
           (dfi_item1.command_cancel != 1)) begin
                case (dfi_item1.OP[1:0])
                        'b00    : dfi_item1.num_of_words=BL16;
                        'b01    : dfi_item1.num_of_words=BC8_OTF;
                        ...
                endcase
        end
        if ((dfi_item1.rddata_en) && (dfi_item1.command_cancel != 1)) begin
                if (dfi_item1.CMD == RD) begin
                ...
                    case (dfi_item1.num_of_words)
                          //Return (0.5*BL) words (2*Device size)
                          BL16    : n_words = n_words + 8;
                          BC8_OTF : n_words = n_words + 4;
                          ...
                    endcase
                ...
endtask : num_of_words
```

```
task  mc_driver::decode_cmd();
        case (dfi_item1.CMD)
                ACT :   begin
                        ACT_cmd();
                end
                RD :    begin
                        RD_cmd();
                end
              ...
        endcase
endtask : decode_cmd
```

```
task automatic  mc_driver::RD_cmd ();    //READ command_encoding (2-cycle cmd)
        CMD0 = {
                dfi_item1.CID[2:0],
                dfi_item1.BG[2:0],
                dfi_item1.BA[1:0],

                dfi_item1.BL_mod,              //Bl_mod bit
                5'b11101
        };
        CMD1 = {
                dfi_item1.CID[3],
                2'b00,
                dfi_item1.AP,                   //Auto precHARGE Active
                ...
        };
        else begin
                CS0 = 0;
                CS1 = 1;
        end

endtask : RD_cmd
```

```
task automatic mc_driver::PREab_CMD (); // Precharge All command_encoding
        CMD0 = {                        // (1-cycle cmd)
                dfi_item1.CID[2:0],
                5'b00000,
                ...
        };
        CS0 = 0;
endtask : PREab_CMD
```

As for commands transaction to pin-livel conversion, the driver uses `decode_cmd()` to decide which of the cmd encoding tasks to execute and perform the proper command encoding according to the JEDEC JESD79-5A standard. As shown in the code snippets above, there are two types of commands; two-cycle commands (like read, MRW, MRR, ACT) and one-cycle command (like prechargeAll). They save the encoding information in a placeholder static logic variable called `CMD0` for first cycle command address information, and `CMD1` for the second cycle if the command is a

2-cycle one. The next step in the `drive_dfi` task is to append these command cycle(s) information to a static logic queue called `CMD_queue`. The same thing goes for `cs_queue`. The following figure shows a flow chart illustrating the `drive_dfi()` task and how it works.

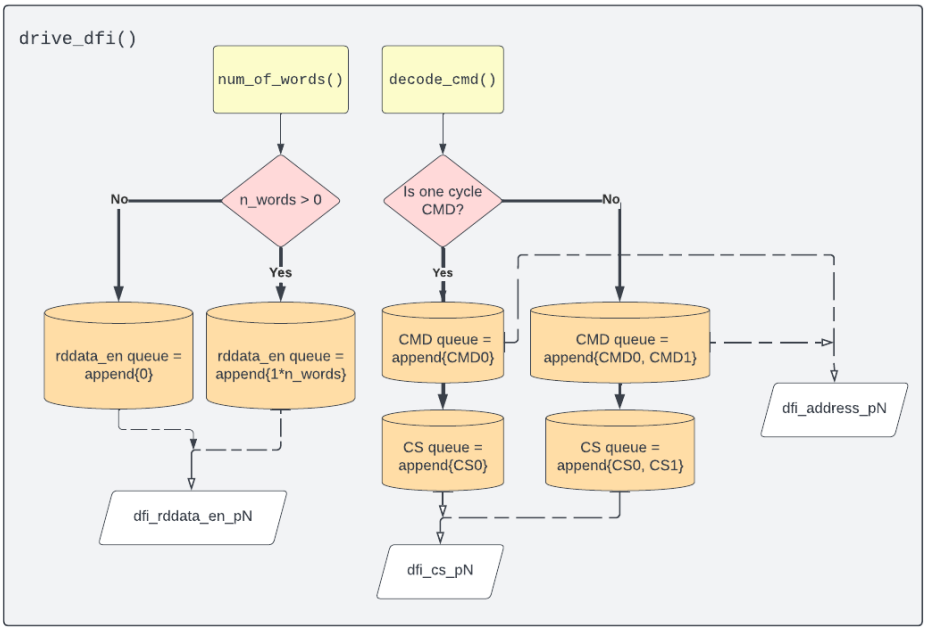


Figure 36. A Flow Chart Diagram illustrating the drive_dfi () task

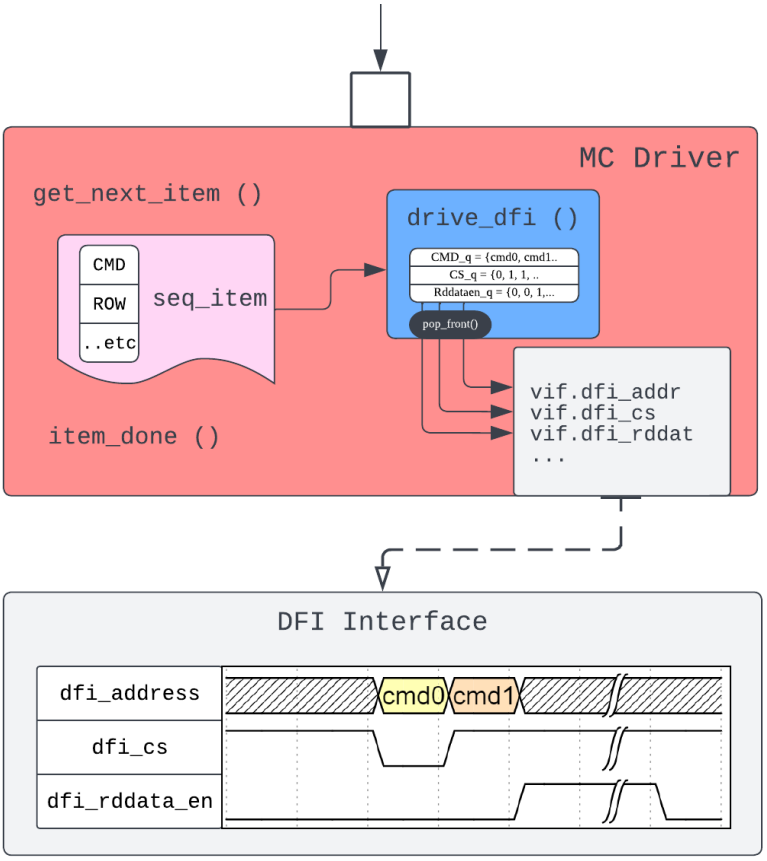


Figure 37. The driver's run phase illustration

## 8.9 MC Monitor

The role of the MC monitor is to monitor the signals on the DFI interface whether the signal is driven by the driver or is an output of the PHY DUT then read these signals and translate them to a higher level transaction that is then sent to the scoreboard and the subscriber for further analysis. Due to the nature of the monitor - explained in section 6.4 - two separate parallel threads were made to monitor commands and data separately.

### 8.9.1 Command Thread

The command thread monitored the phases of dfi_address and dfi_cs which are driven by the MC driver. the thread figures out the type of the command, whether or not it's canceled and decodes the address into a sequence item, and writes it. For MRW command the thread sends the BL, RL, postamble, and preamble patterns data stored into the mode registers. Furthermore, for MRR and RD commands the thread also keeps track of the BL of each data packet in order which can vary from the one stored in the mode register when doing an MRR or setting BL_mode bit to 0 during an RD command. The Command thread ignores DES commands and the Dummy RD command in BL32 mode.

### 8.9.2 Data Thread

The data thread monitors phases of rddata and rddata_valid signals coming from the PHY. It calculates words for each data transaction based on the BL, counts valid data bits up to these data words then writes the transaction.

### 8.9.3 Data Rotation

As mentioned in the DFI standard when operating the PHY at 1:2 or 1:4 frequency ratio, each data transition has to start at the phase after the one on which the last data transition ended regardless of the space between the two data packets. The data thread has to keep a pointer that keeps track of which phase it should start reading from.

### 8.9.4 Communication between the Two Threads

Since the command thread needs to send the BL of each data transaction to the data thread, a mailbox is used as a buffer to communicate the transaction BL between the two threads properly.

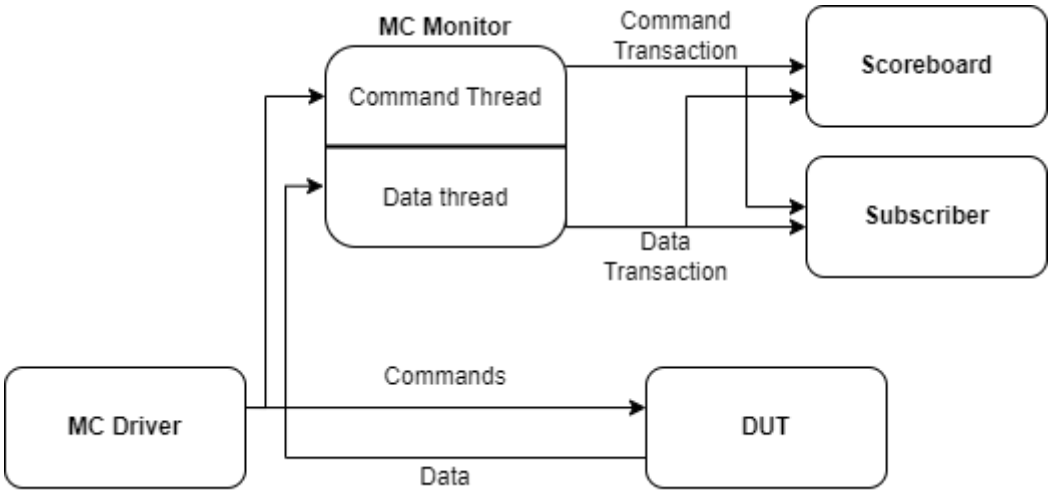


Figure 38. MC monitor communication and threads

# 8.10 DRAM Driver

The DRAM in the memory systems is a slave agent in its interaction with the memory controller through the DDR PHY. The memory controller, the master agent, is responsible for initiating the communication with the DRAM, the communication is based on commands sent to the DRAM by the MC [24]. These commands could be written, read, activate, precharge, …etc, and the DRAM ought to fulfill these commands and respond properly according to the JEDEC JESD79-5A standard. The response may be in the form of reading data, MRR, change the configuration in MRs through MRW …etc. Thus, in order to build a driver to drive the stimulus-response of the DRAM on the JEDEC interface, the driver has to perform a reactive stimulus [28].

The `dram_driver` extends `uvm_driver` class. There are 2 threads run in parallel in the run phase as depicted in figure 39. The first thread is used for translating the pin-level activity at the JEDEC interface into a transaction level (`translation()` task), the thread then fills out the values of the written mode registers through

`fill_mode_regiser()` task. The other thread is responsible for parsing the mode registers and interpreting the saved settings in order to provide proper data stimulus on the JEDEC interface of the DUT, then the thread calls `drive()` task.

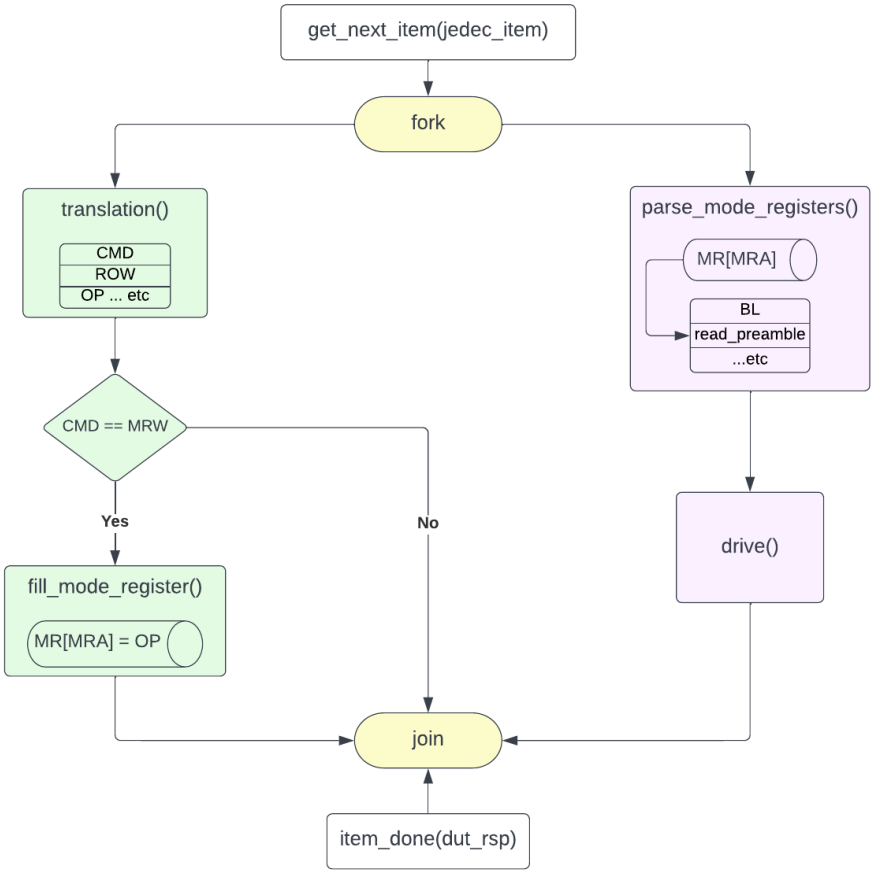


Figure 39. DRAM Driver run phase

The `drive()` task takes the translated and then parsed sequence item as an input (`jedec_seq_item_1`) and outputs the response sequence item (`dut_rsp`). Firstly, the task clones the translated/parsed `jedec_seq_item_1` into the response item `dut_rsp` and then drives the DQS signal and the data bus to the DUT. The output `dut_rsp` is then passed as an argument to the `item_done()` as shown in figure 39. Accordingly, the DRAM sequencer sends the `dut_rsp` item back to the related sequence class (`dram_resp_seq`), which in turn adjusts the transaction to meet the settings received in the `dut_rsp` item (see section 8.10.1 for more details on the `dram_resp_seq`). This methodology is called the reactive stimulus driving technique mentioned in [28]. A snippet of the `drive()` task is shown below.

```
task dram_driver::drive(input ddr_sequence_item jedec_seq_item_1, output
ddr_sequence_item dut_rsp);
        if(!$cast(dut_rsp,jedec_seq_item_1.clone()))
        dut_rsp = jedec_seq_item_1;

        jedec_driver_vif.cb_J.DQS_AD_i <= jedec_seq_item_1.dqs;
        jedec_driver_vif.cb_J.DQ_AD_i <= jedec_seq_item_1.data;

endtask
```

### 8.10.1 DRAM Response Sequence

The `dram_resp_seq` is closely related to the DRAM driver. It receives the DUT response on the JEDEC interface and sends the proper DRAM response back to the driver. The response sequence implements seven tasks to perform the logic of DRAM response to any command, namely, `DRAM_resp`, `fill_data_q`, `fill_dqs_pre_q`, `fill_dqs_post_q`, `fill_dqs_inter_q`, `calc_rd_gap`, `drive_MRR`. The `DRAM_resp` task is the main task that calls the others. As shown below in the flow chart, two queues (DATA, DQS) are filled according to the DUT commands and settings, and then the transaction is created regularly by popping from the queues.

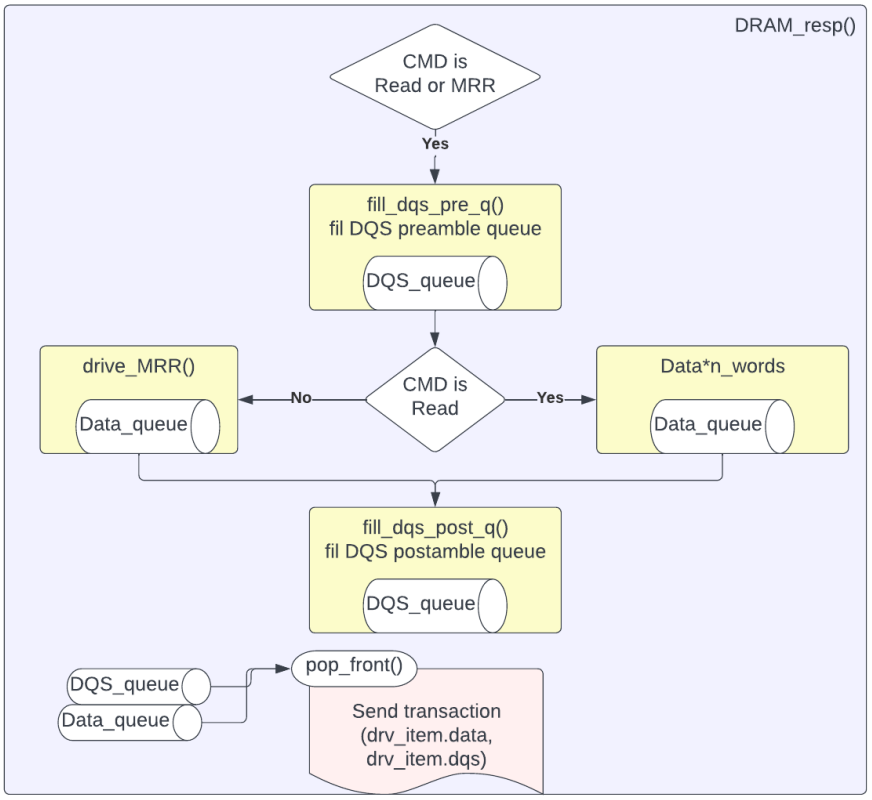


Figure 40. : Simplified Flow Chart showing the DRAM_resp task flow

## 8.11 DRAM Monitor

The goal of the monitor is the opposite of the driver. It observes the pin-level activities at the interface and converts them into the transaction level. The transactions constructed by the monitor are then sent to the scoreboard and subscriber for analysis.

The PHY block has two interfaces; One interface defined by the DFI protocol is between the PHY and memory controller. The other interface defined by the JEDEC is between the PHY and DRAM. The DRAM monitor class is concerned with the JEDEC interface. The JEDEC interface's signals are as follows:

| **Pin** | **Type (With respect to the PHY)** | **Function** |
|---|---|---|
| CA [13:0] | output | Sends the command and address to the DRAM. |
| CS | output | An active-low signal used for rank selection and is also part of the command code; It masks the command when its value is high. |
| DQS | input | It is a data valid flag |
| DQ | input | is the data bus for the RD and MRR data |

Also, the JEDEC interface uses the DFI PHY clock.

### 8.11.1 Using Two Threads In The Run Phase

The monitor retrieves the transactions for the pin-level activities. In the case of a RD or MRR, the essence of a single transaction is the command observed at the CA pins and the data observed at the DQ pins. Due to the simplicity of this task, the first thought about implementing the run phase of the DRAM monitor involved using only one thread. However. the concept of one thread proved to be problematic when considering the scenario of issuing a second RD or MRR command before the data of the first RD or MRR, respectively, is received completely. Figure 41 illustrates the timing diagram of the scenario. The reason behind the problem is that while the thread is in progress collecting the data of the first transaction, another command of a new transaction should be collected, too.

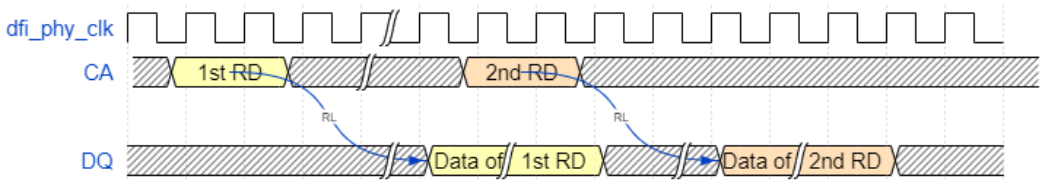

Figure 41. Two consecutive RDs.

Based on that, two threads were used, one for managing the commands and addresses from the CA pins and the other thread for collecting the data from the DQ pins, so for a single RD or MRR transaction, there are two sequence items sent to the scoreboard and subscriber. The first transaction contains the information observed at the CA pins from the command thread, and the second transaction contains the information observed at the DQ pins from the data thread.

### 8.11.2 Command Thread Implementation

The goal of the command thread is to observe the commands and addresses using the pins of CA and CS. It is implemented using a casex statement with CA [4:0] as the expression. These five bits of CA are used to determine the type of command according to Table 30 in section 4.1 of the JEDEC. Based on the command type, other information is being collected such as row, column, CID, auto-precharge flag (AP), burst length mode flag (BL_mod), and operands (OP).

### 8.11.3 Data Thread Implementation

The goal of the data thread is to collect the data of the RD and MRR commands from the pins of DQ with the help of the DQS. For compiling the valid data, a particular pattern must be detected before the valid data is received. This pattern is called a preamble. Also, it is required to detect another pattern called a postamble after receiving the valid data. The preamble and postamble patterns are programmable in DDR5 devices. Writing a code to detect these types of patterns is fairly simple. However, as specified in section 4.5 in the JEDEC, the memory controller shall not add any command gaps to satisfy the preamble and postamble settings, so in many cases, the difference between two consecutive RD or MRR commands is less than the sum of the postamble and preamble. In such cases, the postamble takes precedence over the preamble generating a pattern called the interamble. The difficulty in

detecting the interamble is due to the fact that the shape of this pattern depends on the preamble and postamble settings, burst length, and the time difference between the two consecutive commands. Figure 42 shows an example of the possible interamble patterns when the difference between two consecutive RD or MRR is BL/2 + 2.

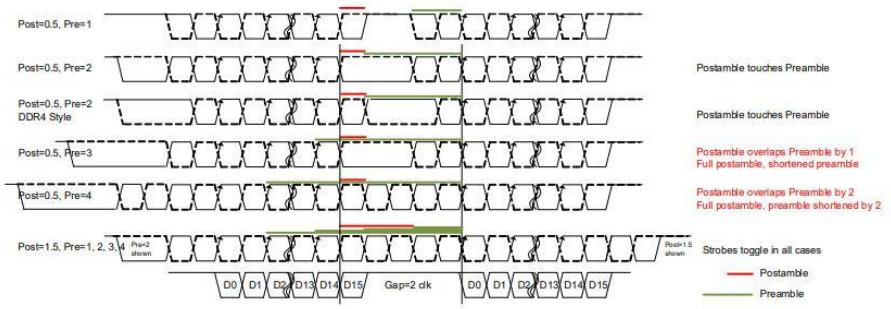


Figure 42. Interamble patterns with commands gap = BL/2 +2.

To avoid this difficulty, detecting the valid data is done by differentiating between two different cases based on the timing of the second RD or MRR command on the CA bus with respect to obtaining the data of the first RD or MRR completely on the DQ bus. To gain some perspective on why there is a need to differentiate between them,

The task used for collecting the data runs in a forever loop, so at the end of each loop, it determines how the data of the next cycle, i.e., command, should be collected by setting the values of some flags local to the DRAM monitor class.

As shown in figure 43, case 1 represents when the second RD or MRR command is issued after the data of the first RD or MRR, respectively, is obtained completely. In this case, after collecting the data of the first read (represented by the vertical, red line in the figure), the data task sets the flags for using the preamble pattern detector because the difference between the two commands is bigger than the sum of the preamble and postamble, so there will be no interamble.

Figure 44 shows the second case in which the second RD or MMR is issued while collecting the data of the first RD or MRR, respectively, is still in progress. The number of clock cycles difference between two RDs or MRRs that induces interamble depends on the preamble and postamble settings, but at worst, the interamble emerges if the clock cycles difference is less than or equal BL/2 + 4 as shown in section 4.5.1 in the JEDEC. Noting that BL/2 + 4 is a subset of the current case, there is no need for

detecting the interamble to know when the valid data begins. In this case, the data task sets the flags for using the number of clock cycles between the two RDs or MRRs.

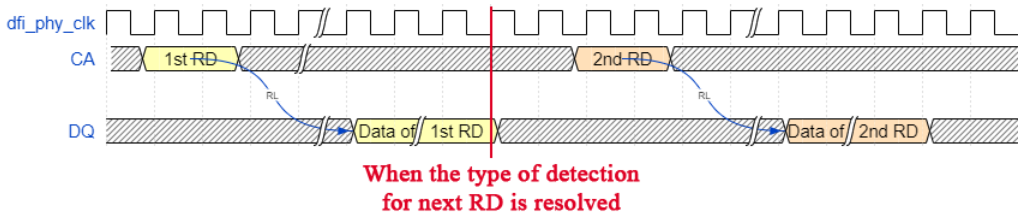


Figure 43. Case 1: 2nd RD is issued after the data of 1st RD is obtained completely.

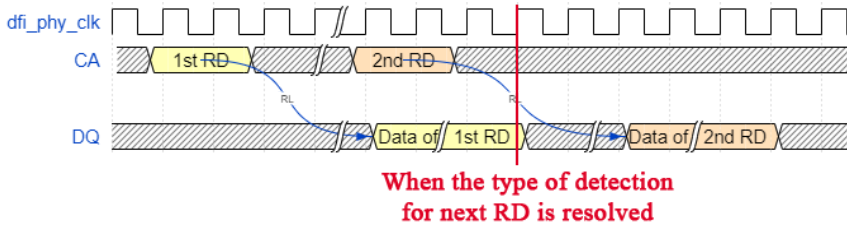


Figure 44. Case 2: 2nd RD is issued before the data of 1st RD is obtained completely.

Based on the previous discussion, using postamble for detecting the end of the valid data is problematic due to the many possible variations in the interamble pattern. Also, the use of clock cycles difference between consecutive RDs or MRRs eliminated the need for detecting the interamble. However, another aspect that needs handling to collect the data correctly is deciding when the valid data of the current RD or MRR transaction ends. It has been done using the burst length. The value of the burst length depends on many parameters. For the RD command, The burst length value could be according to the value stored in MR0 if CA[5] (BL*) is low. Otherwise, the burst length is BL16. For the MRR command, the burst length is always BL16 independent of the MR0.

### 8.11.4 Pattern Detector Implementation

As shown previously, pattern detection is an integral part of the monitoring at the JEDEC interface. To illustrate its implementation, the 000010 pattern will be used as an example. This pattern corresponds to the case when OP [1:0] in MR0 = 3’b011. Figure 45 shows the state transitions diagram of the pattern detector. As it does not correspond to a hardware block, a software implementation was exploited. The code snippet below shows its implementation in which if statements are used to check for the next state, continue statements are used to jump to the next iteration if the

expected bit is not received, and a break statement is used to end the loop and continue to the next part of the code. Furthermore, the DQS is driven to zero during its idle state, so in effect, the sub-pattern starting with the first one is the part that needs to be detected. This is handled using an overlap flag to skip the part of the pattern detected during the idle state of the DQS.

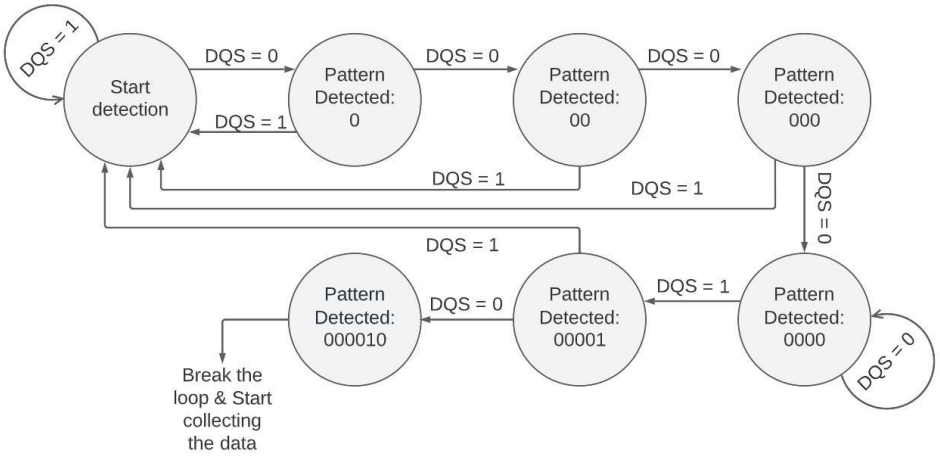


Figure 45. State Transitions Diagram of The Pattern Detector.

```
3'b011: begin
    if (overlap_flag_3tCK === 0) begin
        if (jedec_monitor_vif.DQS_AD_i !== 0) continue;
        @(posedge jedec_monitor_vif.dfi_phy_clk);
        if (jedec_monitor_vif.DQS_AD_i !== 0) continue;
        @(posedge jedec_monitor_vif.dfi_phy_clk);
        if (jedec_monitor_vif.DQS_AD_i !== 0) continue;
        @(posedge jedec_monitor_vif.dfi_phy_clk);
        if (jedec_monitor_vif.DQS_AD_i !== 0) continue;
        @(posedge jedec_monitor_vif.dfi_phy_clk);
    end
    if (jedec_monitor_vif.DQS_AD_i !== 1) begin
        overlap_flag_3tCK = 1;
        continue;
    end
    @(posedge jedec_monitor_vif.dfi_phy_clk);
    if (jedec_monitor_vif.DQS_AD_i !== 0) continue;
    break;
    end
```

### 8.11.5 Communication Between The Two Threads

The operation of the data thread depends heavily on information sent from the command thread. This is realized using mailboxes and static local variables in the DRAM monitor class. In the data thread implementation, it has been mentioned that

the data task uses the number of cycles between two RDs or MRRs; it is calculated in the command thread and then stored in a mailbox to maintain the order. Also, during MRW, the values of the burst length, preamble, and postamble are stored in static variables, so they can be used during data collection.

One final note regarding the DRAM monitor is that the DQS offset was not used. This is specific to the current design implementation because the DQS used is always the default which is zero clock cycles.

## 8.12 Scoreboard and Reference Model

As previously explained, the general goal of the scoreboard is to report whether the output of the DUT is correct as it is measured according to the expected output. Typically, this is done by sending the input, which is the same that was sent to the DUT, to a reference model that behaves in perfect accordance with the specification that is described in the standards. Afterward, the output of the reference model is sent to the scoreboard representing the “expected output”; then, the scoreboard compares the received output with the expected output.

In our case, there were two interfaces with different signals, frequencies, and protocols. This means that there are two different types of inputs and a corresponding two types of outputs. Furthermore, the input of one interface appears as output at the other interface and vice versa. Therefore, the scoreboard had to deal with four types of transactions: the DFI input and output, and the JEDEC input and output. For our DUT, the DFI input is a command and the DFI output is RD or MRR data whereas the JEDEC input is the RD or MRR data and the JEDEC output is the translated command.

That dictated sending the input transactions to the scoreboard to pass them to the reference model before obtaining the expected output which is then compared to the received output. Moreover, because each input was followed by its corresponding output, we chose to implement the in-order comparison; i.e., once an input arrives at the scoreboard, the expected output is instantly computed by the reference model, then the scoreboard waits for the corresponding DUT’s output.

Another structural element of the scoreboard is the use of queues. This was necessary because the rate of input transactions is different from the rate of output even though

there was a one-to-one mapping between the input and output queues elements and the comparison was done in order. There were two options regarding this point: use regular SystemVerilog queues or TLM FIFOs. The advantages of using TLM FIFOs are that the integration between the functionality of the TLM communication and queue operation with semaphores is done automatically by UVM. The comparison should wait until the transactions come from the monitors; this was also embedded in the TLM FIFO methods since its original implementation uses the concept of a mailbox. However, using TLM FIFOs dictated that each monitor have two analysis ports corresponding to 4 TLM FIFOs inside the scoreboard. On the other hand, using queues dictates being able to tell the difference between an input and output transaction inside the write function, and it required handling the synchronization of calling the comparison functions manually as we had to make sure that the queues are not empty. Another notable issue is the need to properly store the transaction in the queue by creating a new handle each time the write function is called and doing proper copying (deep copy) of the incoming transaction because it included queues itself; all of this manual labor is encapsulated in the use of TLM FIFOs. However, using queues needed only two analysis ports and implementations in addition to the flexibility of out-of-order comparison which was a possibility at the beginning of the project before the environment design became well-defined and stable. Therefore, we eventually modified the transaction to make the input and output distinct to be able to leverage the flexibility of queues to be ready for future changes even though the TLM FIFO had much functionality already built-in.

The sections of the scoreboard that are different from the normal component and thus need explanation are as follows:

- Two analysis implementations: one for each agent
- Four queues of DDR sequence items: one for the input and one for the output of each interface.
- The run phase: having two threads, one for each interface.
- Two write functions: one for each agent
- Two comparison functions: one for each interface
- Reference model functions: three functions to handle the inputs of both interfaces.

First, the two analysis implementations are connected to the analysis ports of the two agents. The corresponding write functions are called by the monitors when either an input or output transaction is sampled from one of the interfaces. When a transaction arrives in the write function within the scoreboard, it first copies it inside a local transaction. Then, it checks the type of the transaction (input or output) to push it into the corresponding queue. Then, the write function raises one of two flags to signal that either a new input or output, from one of the interfaces, is ready in its corresponding queue. When both the input (stimulus) and the output (response) flags are raised, the appropriate thread in the run phase task will call the appropriate comparison function. Consequently, the comparison function will first pop one input transaction from the appropriate (according to the interface) queue and pass it to reference model methods which will return the "expected output". Then, the comparison function will pop an output transaction from the appropriate queue which is the "received output" from the DUT. At last, the two transactions are compared; if they are not equivalent, an error will be reported. The stated sequence of events is illustrated in the following diagram. The diagram shows the command translation check, but the data translation is similar to it.

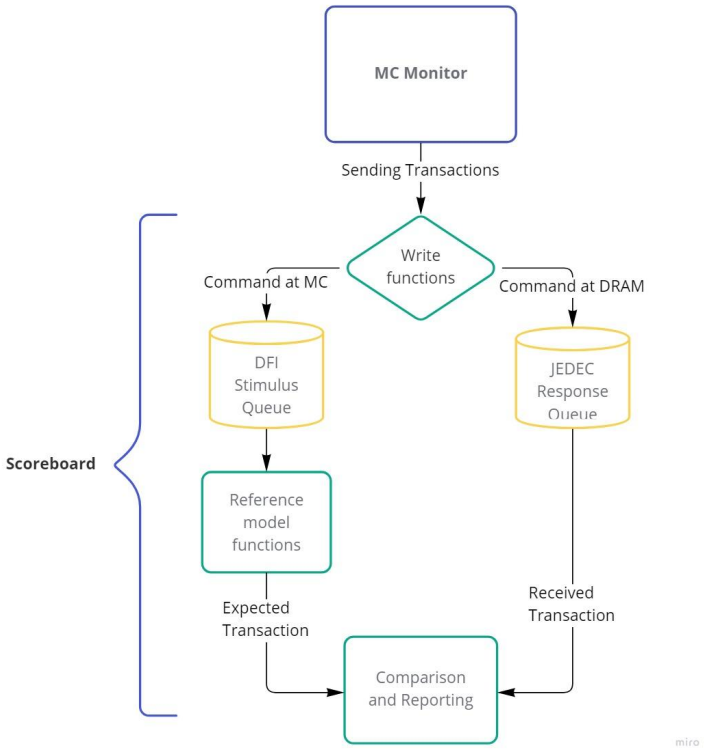


Figure 46. Flowchart Demonstrating Command Translation Check

The reference model is composed of two main functions that call other functions: the first takes a DFI input transaction and produces a JEDEC output transaction, and the second one takes a JEDEC input transaction and produces a DFI output transaction. Since the DUT behavior involves little computation (e.g., arithmetic operations), the reference model is simple and the verification of a lot of the functionality is done by assertions. There was a decision to determine the structure of the scoreboard in the beginning: make the reference model as a separate component or integrate it with the scoreboards. Separating the reference model is better for reusability and generally a better practice which keeps the different functionality of different environment components separated. However, we settled at the end on integrating the reference model in the scoreboards because of its simplicity. The two options are shown in the figure below.

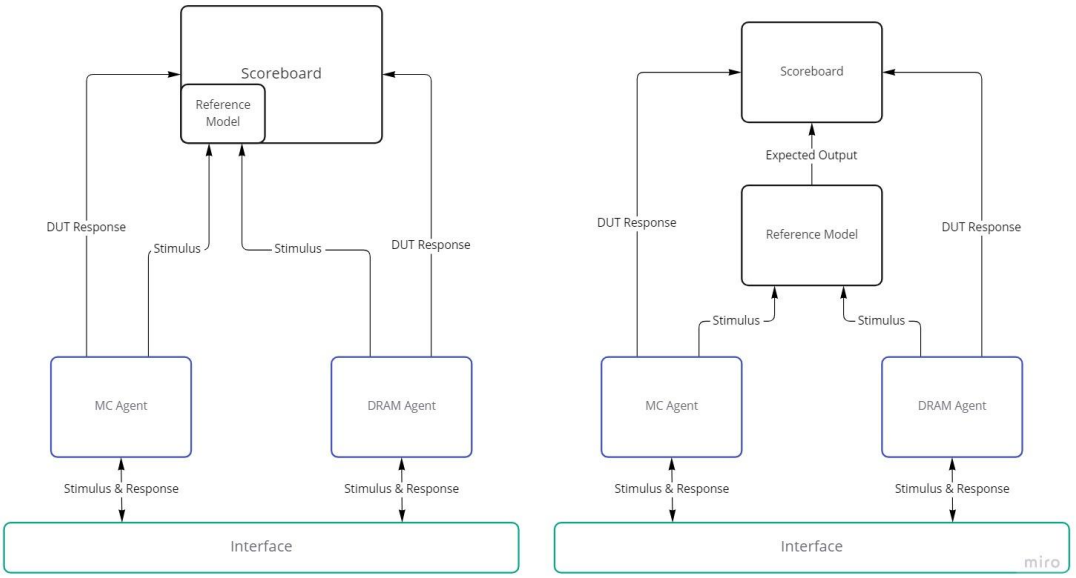


Figure 47. The Classes Architecture Alternatives

# 8.13 Subscriber

As the subscriber receives broadcasted transactions, it is mainly used to implement functional coverage. Functional coverage identifies the design requirements, so it measures the coverage of the design intent. Structurally, the implementation of the functional coverage is divided into two parts: JEDEC functional coverage and DFI functional coverage. The content of both parts is based on the design requirements as specified in the verification plan. The functional coverage features in SystemVerilog

are extremely powerful for translating the design requirements into code. However, if care is not taken, coverage can be collected for bins of no importance or meaning. Cross-coverage can easily generate a huge number of unwanted bins which is expensive in terms of simulation time and memory. The following part illustrates the implementation process to avoid this problem. Two cover groups were created, one for the JEDEC coverage and another for the DFI part.

### 8.13.1 Cover Points

Firstly, the relevant data members of the sequence item related to each interface were listed in their corresponding cover group. They were listed as cover points with their weights set to zero because they are only sampled to be used in the cross statements, and the total coverage is calculated based on the cover points and cross-coverage, so they should be excluded. The code snippet below shows the implementation of two cover points which are for the command and burst length data members. Also, the figure shows the different ways of defining the cover-point bins. They can be defined explicitly as in CMD_cp, generated automatically as in burst_length_cp, or excluded using ignore_bins as in OP_BL_cp.

```
CMD_cp: coverpoint jedec_sequence_item_coverage.CMD {
    type_option.weight = 0;
    bins MRR = {MRR};
    bins MRW = {MRW};
    bins ACT = {ACT};
    bins RD = {RD};}
burst_length_cp: coverpoint jedec_sequence_item_coverage.burst_length {
    type_option.weight = 0;}
OP_BL_cp: coverpoint jedec_sequence_item_coverage.OP[1:0] {
    type_option.weight = 0;

    ignore_bins exclude_BL32_OTF = {3};}
```

### 8.13.2 Cross coverage

“Cross” is the main feature used for translating the design intent into functional coverage; This is the part where code coverage completely fails to detect. “cross” performs cross product between two or more cover points within the same cover group or variables. It measures the coverage of all bin combinations associated with these cover points or variables, so the cross-coverage produces a large number of bins

because, unlike the cover point, it generates all possible cross bins automatically even if the required bins are defined explicitly. To disable generating cross bins automatically, ignore_bins is used as shown in the code snippet below. For further restrictions on the bin creation, the “with” clause is used for bin filtering. Moreover, illegal_bins can be used as checkers that the given scenarios must not happen. Otherwise, a run-time error will be generated. illegal_bins has a higher priority than ignore_bins.

For cross-coverage, Bins exclusion is implemented using “binsof” to specify the cover point and “intersect” to select the set of values so that a single ignore_bins statement sweeps out multiple bins. The excluded bins are the complement of the included bins, so after the included bins are extracted from the verification plan, the excluded bins are obtained using De Morgan's law.

```
JEDEC_DR_4_cross: cross CMD_cp, actual_burst_length_cp, burst_length_cp {
    type_option.comment = "Coverage model for features JEDEC_DR_4";
    illegal_bins ill = binsof(CMD_cp.MRR) with (actual_burst_length_cp !=
BL16);
    ignore_bins excluded_JEDEC_DR_4_bins = !binsof(CMD_cp.MRR);

    }
```

Another important SV construct is “iff” which defines a condition to disable the coverage for a cover point or bin. This construct was used to disable coverage for some bins when the corresponding commands are canceled so that meaningful coverage is collected.

### 8.13.3 Transition coverage

Defining the transition coverage is tricker than the other coverage types because in our case, the commands that represent real functions such as the ACT, MRR, and RD commands are separated by DES commands, so the required transition coverage is for non-consecutive sampling points. Moreover, the number of DES commands separating two commands is variable and indefinite. There is no syntax for transition bins of indefinite repetition. Another problem is that canceled commands must not be considered part of the sequence and must not break the active sequence evaluation. To overcome these problems, firstly, the DES commands are not sent to the subscriber as

they are of no importance to the coverage, and also, an if statement for the command cancel flag is implemented on the sampling event itself, not on the cover point or bin. Based on these two operations, only the commands representing real functions are sampled. The code snippet below shows part of the transition coverage for the JEDEC interface.

```
JEDEC_DR_7_and_10_cp: coverpoint jedec_sequence_item_coverage.CMD {
type_option.comment = "Coverage model for features JEDEC_DR_7_and_10";
    bins JEDEC_DR_7  =  (MRR => MRR => MRW),
                        (MRR => MRR => ACT);
    bins JEDEC_DR_10 =  (MRW => MRW => MRR),
                        (MRW => MRW => ACT);}

endgroup : JEDEC_transitions
```

### 8.13.4 Coverage Options

The coverage options allow controlling the behavior and calculation of cover points, crosses, and cover groups. Options can be defined for all the cover points inside a cover group by placing them in the cover group, or for finer control, they can be placed inside a single cover point. Examples of the options are weight, goal, and auto_bin_max. Also, the comment coverage option which adds comments to the coverage reports is extremely important for easier analysis of the coverage.

### 8.13.5 Write Functions Implementation

Subscriber should provide an implementation to the write function of the analysis port. However, the subscriber of our testbench is connected to two analysis ports, so for a class to support multiple inputs, `uvm_analysis_imp_decl is used, then two write functions are implemented, one for the JEDEC interface and another for the DFI. The goal of these functions is to receive the broadcasted transactions for collecting coverage, but the implementation of the monitors introduces some complexity in the subscriber. The subscriber receives two types of transactions based on their content. The first type is a command transaction which contains the address and command information and is collected by the command thread in the monitor. The second type is a data transaction which contains the data information of a command and is collected by the data thread in the monitor. The subscriber needs to construct a transaction containing the complete information because the cross coverage measures

values of cover points and variables happening at the same time. Based on that, the implementation of the write functions handle two cases. The first case is when the received command invokes data burst in the DQ bus such as MRR and RD. As shown in figure 48, the received command transaction is pushed into a queue until its data transaction is received. After that, the command transaction is popped up and merged with the data transaction to construct a complete transaction, then the coverage sampling event is triggered in the run phase. The second case is when the subscriber receives a command that does not invoke data in the DQ bus such as ACT and MRW commands. In this case, the transaction is already complete, so the coverage sampling event is triggered immediately.

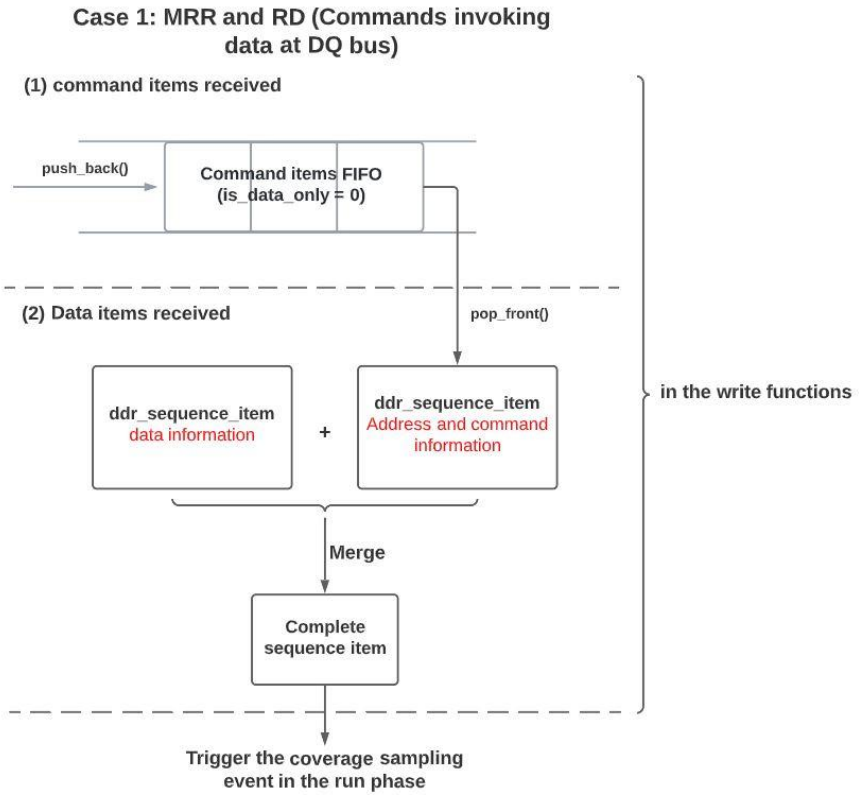


Figure 48. constructing a complete transaction before the coverage sampling.

# 8.14 DDR Assertions Module

The checkers that are derived from the JEDEC JESD79-5A and DFI standards, in the verification plan, and involve timing parameters or sequences of events were implemented using the SystemVerilog Assertions (SVA) language. The structure of the module includes sections for the properties and assertions as follows: JEDEC, DFI

1 to 1 ratio, DFI 1 to 2 ratio, and DFI 1 to 4 ratio. Each section is guarded by a conditional compilation macro (`ifdef) to enable the appropriate assertions for the chosen frequency ratio of the simulation and also for debugging purposes.

Some of the assertions were straightforward to implement using the delay (##n), range (##[n,m]), repetitions ([*n]), and implication (|-> or |=>) operators while others required special work to properly and accurately describe the assertion as written in the verification plan; those trickier assertions will be explained in more detail.

The first example is assertions that involve a DUT input and DUT output signals. The problem is that the stimulus is applied on the negative edge whereas the DUT drives its output on the positive edge. This produces a discrepancy of one cycle between the expected time that the assertion passes/fails and the actual time because the assertions sample signals in the preponed region. Therefore, the stimulus signal is sampled in the same cycle that it is driven in, whereas the DUT output signal is sampled in the subsequent cycle that it is driven in. So, if the stimulus signal is an antecedent and the output signal is the consequent, then the one cycle will be added to the timing between them. If it is the other way around, one cycle will be subtracted from the timing between them.

Another example is a design requirement stating that the number of clock cycles dfi_rddata_en is HIGH must be equal to dfi_rddata_valid. To do this, the assertion had to count the number of cycles within which the two signals were asserted and compare them. A first trial at the assertion attempted to OR two sequences that count the cycles during which a signal is HIGH. The first one deals with the dfi_rddata_en signal as it comes before the dfi_rddata_valid signal. This first sequence should count the number of cycles for dfi_rddata_en (N1), then pass that variable to the second ORed sequence which will count the number for dfi_rddata_valid (N2) then terminates by either passing or failing. Given that the dfi_rddata_valid signal will come at a later time, this dictates that the first sequence must “artificially” fail after calculating N1 (by inserting ##1 0 at the end of the sequence) so that the assertion waits for the second ORed sequence to calculate N2 and compare it to N1 before the final passing or failure. However, a more intuitive alternative is to use implication. But counting itself utilized implication to avoid the assertions failing on every cycle

the dfi_rddata_en and dfi_rddata_valid are not asserted. Therefore we resorted to defining two sequences, one for calculating N1 and another for N2, that are used like this: Sequence1 |-> Sequence2. Then, a comparison between N1 and N2 is done after a delay operator ##0. Lastly, the delay operator (##0) was used instead of the implication (|->) in the counting sequence because the implication operator is illegal inside SVA sequences; this was successfully implemented without the assertion failing every cycle because the property itself that used the counting sequence had an implication operator between the two sequences. A code snippet to illustrate the methodology is shown below.

```
sequence count(count, Signal);
          ($rose(Signal), count=0) ##0 first_match( (Signal[=1],
count=count+1)[*1:$] ##1 $fell(Signal) );
endsequence: count

property Equal_Count;
      int count_1; int count_2;
      @(posedge clk) disable iff (!reset)
      count(count_1,Signal_1) |-> ##[1:n] count(count_2, Signal_2) ##0
count_1==count_2;
endproperty: Equal_Count
```

The third challenging category of assertions is those using dynamically changing variables in delay operators. There are properties that depend on parameters that can be changed by MRW commands. These variables are stored in the UVM resource database by the agents and sent to the top testbench to be accessed by the assertions module. The problem is that dynamically changing delays are not permissible in SVA. Hence, a workaround was needed, and there are two ways to accomplish this. First, use a property that implements the dynamic delay as shown in the code excerpt below:

first, we store the delay in an int variable v then we subtract 1 from that variable till it is less than or equal to zero which is checked using first_match system task. Lastly, we use the dynamic_delay sequence inside the property to serve as a delay between two sequences.

```
sequence dynamic_delay(delay);
 int v;
```

```
 (1, v=delay) ##0 first_match((1, v=v-1'b1) [*0:$] ##1 v<=0);
endsequence: dynamic_delay
property Changing_Delay;
      @(posedge clk) disable iff (!reset)
      Antecedent |-> dynamic_delay(delay) ##0 Consequent;
endproperty: Changing_Delay
```

The second method is using an automatic task that is called from within the property. The general idea is to keep the antecedent as is and make the consequent `TRUE which will always pass. However, the rest of the assertion functionality is done inside the task which is called representing the sequence match action. Both the passing and failing conditions and messages are implemented inside the task which can legally operate on dynamically changing variables. The drawback of this way of implementing variable delays in assertions is that the assertion always passes on the waveform of an assertions debugger. Debugging such an assertion is done using the run log which will show the passing or failing messages. It is notable that if the task has an output (e.g., declared as "bit result") that is a local variable in the property, it could be used as the pass/fail criteria by using "##0 result" after the task call; however, it is illegal for a task called from an SVA property to have outputs. Therefore, this second method is easier to construct but more difficult to debug. A code snippet demonstrating the general use case is shown below.

```
task automatic Consequent_Task(input int delay);
    //Code
endtask
property Changing_Delay;
      @(posedge clk) disable iff (!reset)
      Antecedent |-> (`TRUE, Consequent_Task(delay));
endproperty: Changing_Delay
```

Assertions that check variable patterns of signals required merging both methods of dealing with dynamically changing variables. The time between the antecedent and the pattern (consequent) was, again, stored in the resource DB and can change; thus, the first method is preferably used to handle this delay parameter. But to be able to detect multiple patterns, ORed sequences are used where each pattern is represented by an ORed sequence so that if it fails, the assertion will wait for other patterns that might still succeed. An alternative to the ORed sequences is calling a task from the sequence match action which has the flexibility to detect multiple complex sequences.

Another advantage of using a task is that upon failure, customized error messages can be generated at the exact instant the pattern failed whereas the ORed sequences approach may fail to detect a pattern A, but will keep waiting to see if another pattern B will succeed. Therefore, the task approach was chosen. The code snippet below demonstrates both approaches.

```
//First Approach
property Mixed_Property;
      @(posedge clk) disable iff (!reset)
      Antecedent |-> dynamic_delay(delay) ##0 (Consequent_1 or
Consequent_2 or ...);
endproperty: Mixed_Property

//Second Approach
property Mixed_Property;
      @(posedge clk) disable iff (!reset)
      Antecedent |-> dynamic_delay(delay) ##0 (1, Consequent_Task());
endproperty: Mixed_Property
```

Lastly, one of the problems that came out during simulation is excessive memory usage. This has two reasons: assertions that have an antecedent that is level triggered making the assertion start a new thread every cycle after the rising edge of the antecedent condition, or assertions that an open-ended range (e.g., [*1:$]). The first situation is easily fixed by using the $rose() system function to start only one thread when the antecedent first goes HIGH; this of course assumes that the intention of the assertion is not affected by this modification. The second situation is more difficult to circumvent because in a realistic random test, successive antecedent firings can occur before any consequent is asserted which can open so many threads that will eat up the memory quite rapidly. The first, and best, way is to reconstruct the assertion to check the same feature in a different way. It is even better to even think about how the same feature can be checked at a higher level of abstraction in the scoreboard, for example, which would alleviate the burden of checking because it will be converted from pin-level (cycle-by-cycle) checking to transaction-level checking. As an example from our implementation, we tried to make sure that a precharge command must not follow an activate command directly. So, we tried to make a property describing this scenario and then assert this property. The problem that occurred was excessive memory usage. Accordingly, a more diligent way is introduced to solve this problem which is to implement the assertion as a checker in the scoreboard. We stored the

previous command and compared it with the subsequent one and check whether a precharge command follows an activate command or not.

If there is no way to reconstruct the assertion or move it to the scoreboard, a way of suppressing the assertion when the number of open threads reaches a certain threshold may be employed. This can be done by calling a function from the action block of the property's antecedent; this function would increment a counter that keeps track of the number of open threads of this property, and it will decrement the counter when a thread closes. When the counter reaches the appropriate threshold, the assertion is either forced to pass by ORing the consequent with a sequence that passes when the counter is above the limit. Another similar way is to put this sequence after the antecedent so that it won't be detected unless the counter is below the threshold. The code snippet below demonstrates the latter approach.

```
int k=0;
function count_m(bit add);
    if(add)
        k = k+1;
    else
        k = k-1;
endfunction
property Memory_Intensive ;
    @(posedge clk)  disable iff (!reset)
    (Antecedent, count_m(1)) ##0 (k<=N) |-> ##[0:$] (Consequent, count_m(0));
endproperty: Memory_Intensive
```

# 8.15 Testcase Library

There are four test classes extending the base test class, each one of them incorporates specific sequences that fulfills certain purposes. The tests are `ddr_sanity_test.sv`, `rand_test.sv`, `rand_test_corners.sv`, `b2b_test.sv`. The run phases of each test class run the run phase of the base class (which includes base, reset sequence) through the super method, after that the tests raise objection and they don't drop it until all sequences called in the run phase are consumed. See the code snippet below for illustration.

```
class #_test extends base_test;
...
        task run_phase(uvm_phase phase);
                super.run_phase(phase);
                        phase.raise_objection(this);
```

```
                        ...
                        phase.drop_objection(this);
        endtask

endclass
```

Notice that, each sequence can have access to one driver at a time. A single driver cannot run multiple sequences at the same time. However, if it happens that multiple sequences want to access the same driver, the sequencer comes to action and does an arbitration process to grant the access to whatever sequence relative to an arbitration mode, but this is not the case in this work.

### 8.15.1 Sanity Test

The `ddr_sanity_test` performs a sanity check to the DUT, it starts the `ddr_sanity_seq` on the `mc_sequencer`. Also, it runs the `dram_resp_seq` on the `dram_sequencer` in a parallel thread. The sanity test is a direct test that performs multiple command transaction with specific settings.

#### 8.15.1.1 Sanity Sequence

The `ddr_sanity_seq` implements six tasks; five tasks each of which performs a command transaction and the sixth is for test termination. While the sequence of transactions is determined by the sequence of tasks calling in the body task as shown below in the snippet.

```
class ddr_sanity_seq extends base_seq;
...
       task body();
          ACT_cmd(.iscanceled(0), .delay(8), .CMD_prev(CMD), .CMD(CMD));
         read_cmd(.BL_mod(0), .AP(1), .C10(1), .iscanceled(0), .delay(8), ...
         ...
         terminate(.delay(8));
       endtask

       extern task MRW_cmd(byte MRA, bit [7:0] OP, int delay, bit iscanceled, ...
       extern task read_cmd(bit BL_mod, bit AP, bit C10, bit iscanceled ...
       extern task PREab_cmd(bit AP, int delay, command_t CMD_prev, output command_t CMD);
       extern task ACT_cmd(bit iscanceled, int delay, command_t CMD_prev, output command_t CMD);
       extern task MRR_cmd(byte MRA, int delay, bit iscanceled, ...
       extern task terminate(int delay);
endclass : ddr_sanity_seq
```

### 8.15.1.2 Simulation of the Sanity test

The figure below shows a portion of the sanity test waveform. It depicts an ACT then read commands and a precharge. The data is driven with proper dqs then captured correctly by the DUT. This test is run at matched frequency ratio.

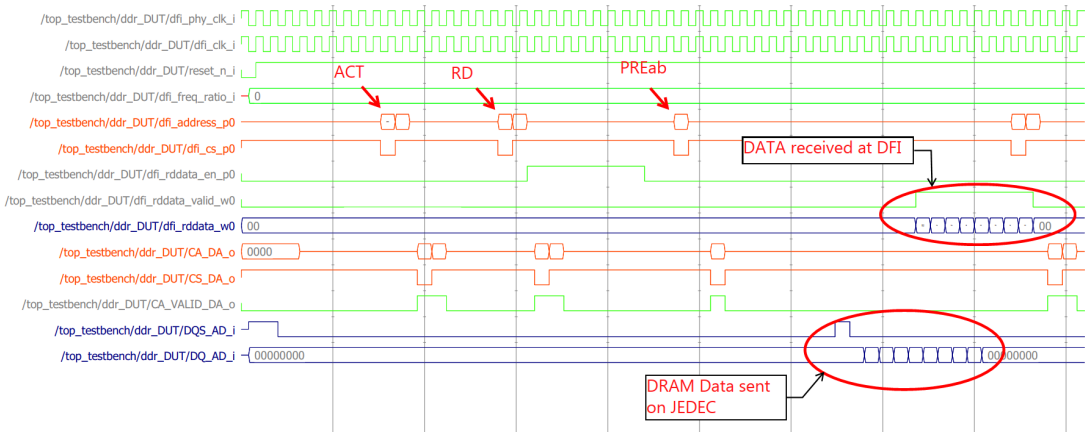


Figure 49. A portion from the sanity test waveform (Matched Freq. ratio)

The figure below shows the same portion of the test but in 1:2 frequency ratio.

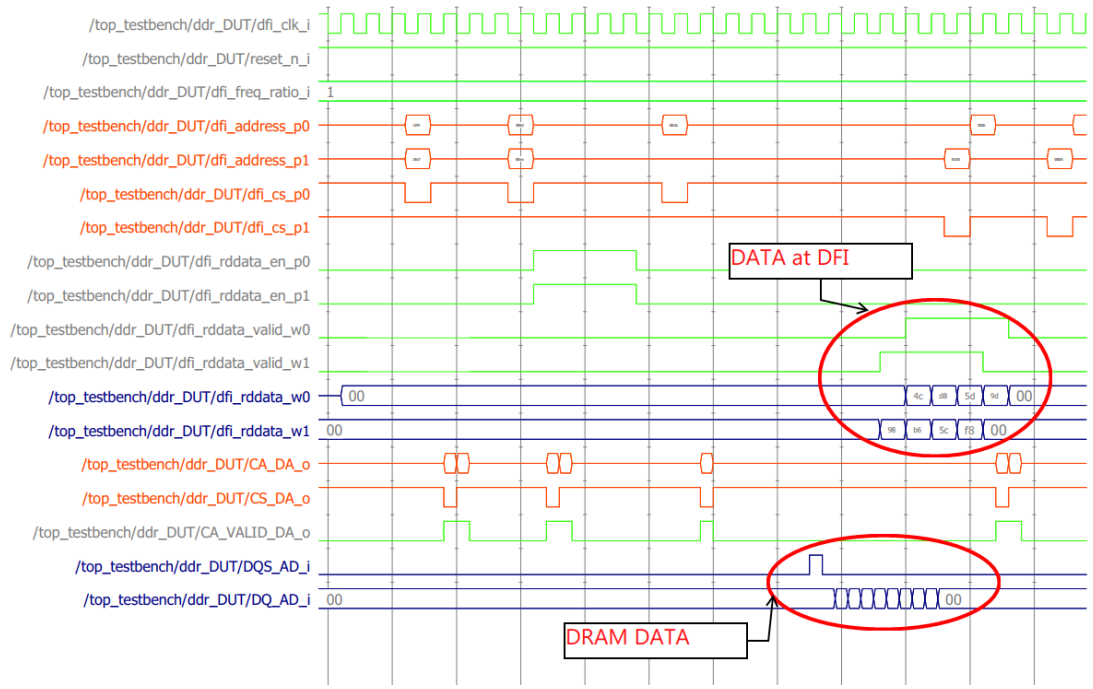


Figure 50. A portion from the sanity test waveform (1:2 Freq. ratio)

### 8.15.2 Random Test

The `rand_test` uses the randomization utilities offered by UVM, its purpose is to increase the coverage collection and hit as many scenarios as possible. It utilized a random sequence class called `rand_seq` sent to MC driver through MC sequencer. A `no_of_transfers` variable is used to define the number of transactions in the test run as shown in the code snippet below.

```
class rand_test extends base_test;
...
    task run_phase(uvm_phase phase);
        super.run_phase(phase);
                phase.raise_objection(this);
                no_of_transfers = 300;
...

                fork
                        begin
                        repeat (no_of_transfers) begin

rand_seq_inst.start(env1.mc_agent1.mc_sequencer1);
                         end
                         DES_seq_inst.start(env1.mc_agent1.mc_sequencer1);
                        end
                        resp_seq.start(env1.dram_agent1.dram_sequencer1);
                join
                phase.drop_objection(this);
    endtask

endclass : rand_test;
```

#### 8.15.2.1 Random Sequence

The `rand_seq` arbitrates randomly between different commands sequences (`MRW_seq`, `MRR_seq`, `ACT_seq`, `RD_seq`, `PREab_seq`) applying the constraints defined in the `ddr_sequence_item` class. According to the JEDEC JESD79-5A standard, there are some constraints on the latency between each command and any other command, also, there are constraints defining the eligibility of issuing commands. For example, it is not allowed to issue MRW or MRR commands if a page is open (i.e. after ACT cmd) [3].

### 8.15.2 Random Corners Test

The `rand_test_corners` is used to perform more specific random test scenarios to achieve high coverage. It is similar to the `rand_test` except for it utilized sequences

(`MRW_seq_corners`, `MRR_seq_corners`, `ACT_seq_corners`, `RD_seq_corners`, `PREab_seq_corners`)

### 8.15.3 Back-to-back Test

The `b2b_test` is a direct test implemented to hit all interamble scenarios. It utilizes `b2b_seq` sequence class. The `b2b_seq` sequence class is similar to the sanity sequence, the difference is that it implements three layers of for loops. Looping on different preamble, postamble settings, and different delays between b2b reads.

## 8.16 Using EDA Tools

The basic tools that are needed for the verification task are a text editor for writing code, an RTL simulator that supports UVM, coverage, SVA, waveform debugging, and a file version control system. The text editor that was used is Visual Studio Code which includes a terminal and a native interface to the version control tool which is Git; this allowed the cross-platform development of code as we needed to work on Windows and Linux. The main simulator that was used is Synopsys VCS, which has a built-in UVM 1.2 library, along with its graphical user interface that is called DVE for debugging.

To automate and configure the process of compilation, running, coverage collection & merging, a shell script was used. When the script is invoked from the command line, different arguments that are related to VCS, DVE, UVM, and our code can be passed. The script is constructed into sections as the following:

- Tool-related variables: used for choosing log verbosity, simulation timeout, …
- Paths-related variables: used for passing the relevant folder and file paths to the simulator
- Environment definitions variables: used for `ifdef .. `endif guards
- Switches sections: used to receive the options and arguments that are used when invoking the script
- Handling the passed options: used for deciding how to compile, run, and report depending on the passed options and arguments
- Help section: used to explain the script capabilities for a user

The main three commands that were invoked are:

- vcs: for compilation
- ./simv: for running
- urg: for coverage reports creation and merging

The following two tables explain the usage of script switches and every compile, run, and report option that was used in the script.

Table 1. Script Switches

| Switch | Usage |
|---|---|
| -help | Print help/usage |
| -verbos_debug | Set UVM_VERBOSITY to UVM_DEBUG |
| -verbos_hi | Set UVM_VERBOSITY to UVM_HIGH |
| -gui | Open DVE for running and debugging |
| -cln_bld | Clean the work area before building |
| -cov_en | Enable code coverage generation |
| -comp_only | Compile only, Do not run the simulation |
| -timeout | time in ns for UVM timeout |
| -ratio | Decide the frequency ratio used throughout the simulation |
| -assert_en | Define flag to bind and use ddr assertions |
| -extra | Extra simulator arguments |

Table 2. Compile, Run, and Report Option

| Option (type) | Usage |
|---|---|
| -sverilog (compile) | switch to the SystemVerilog mode |
| -timescale (compile) | specification of the timescale for the source files |
| -ntb_opts uvm-1.2 (compile) | specify UVM library |
| -debug_acc+all (compile) | addition debug capability |
| +define (compile) | Defines a text macro |
| +libext+.sv+ (compile) | specify the file name extension of the files VCS will search for |
| +incdir (compile) | Specifies directories containing files specified with ‘include |
| top.sv (compile) | Specify top module |
| -q (compile) | Quiet mode; suppresses messages |

| | |
|---|---|
| -l (compile & run) | Specifying a Log File |
| -cm (compile) | Specifies elaborating for the specified types of coverage |
| -cm_hier (compile) | Specify the modules and files to be excluded from coverage |
| -gui (run) | Open graphical user interface |
| +UVM_TESTNAME= (run) | Specify UVM test to be run |
| +UVM_TIMEOUT= (run) | Specify UVM simulation time-out |
| +ntb_random_seed= (run) | Specify randomization seed for the simulator |
| -cm_dir (run) | Specify directory to store coverage database after running |
| -dir (report) | Specify directory to fetch/merge coverage database for reporting |

To collect coverage for a certain test and frequency ratio, the script will save the coverage database in a folder that is distinguished by the test name, frequency ratio, and the seed that was used. Therefore, the script can be used to run a certain test with multiple seeds and the coverage databases will be merged to produce a complete report with non-overlapping coverage data. Since the meaningful code coverage is that of the RTL only, the cm_hier file was used to exclude the UVM package, interfaces and assertions modules, and environment classes from the coverage report.

For debugging purposes, DVE waveform viewer and the assertions pane were utilized. The assertions window was particularly useful because it shows unattempted and incomplete assertions which helps to infer how to modify/create tests to cover all assertions.

Normally, different simulators behave differently in matters such as randomization. However, a notable point about using the clocking block in the monitor is that the implementation differed between VCS and the Mentor Graphics Questasim simulator. The difference arose when a signal (data) was to be sampled after detecting the enable signal. Since both signals were outputs of the DUT, the sampling of them utilized the “1step” keyword to make sure the monitor samples output signals when the DUT is not changing them anymore. There was a 1 cycle discrepancy between VCS and Questa which made one of them miss the first data word and the other miss the last one. Ultimately, we modified the condition that initiated the monitor sampling of data to have the same behavior in both simulators and avoid simulator-specific errors. The

behavior is depicted in the simulation below which was done on EDA Playground; the code is similar in essence to the situation in the MC monitor. The design is a simple up/down counter that outputs a “Valid” signal when its count output (Q) is enabled. As seen below, the data is pushed in a queue using the Valid signal as a condition. It is shown that VCS captures only up till the 8th count, whereas Questa captures the 9th count. Code snippet from a demonstrative testbench and the output of both simulators is shown below.

```
initial
begin
  fork
    begin
      CB1.Enable     <= 1;
      CB1.UpDn       <= 1;
      delay();
      Running        <= 0;
    end
    begin
      forever
      begin
        if(CB1.Valid) begin
          Data_Q.push_back(CB1.Q);
        end
        @(CB1);
      end
    end
  join_any
  $display("The data queue is: %p", Data_Q);
end
```

---

**VCS Output:**
The data queue is: '{1, 2, 3, 4, 5, 6, 7, 8}
**Questasim Output:**
The data queue is: '{1, 2, 3, 4, 5, 6, 7, 8, 9}

---

Link to playground: https://www.edaplayground.com/x/s7Ra

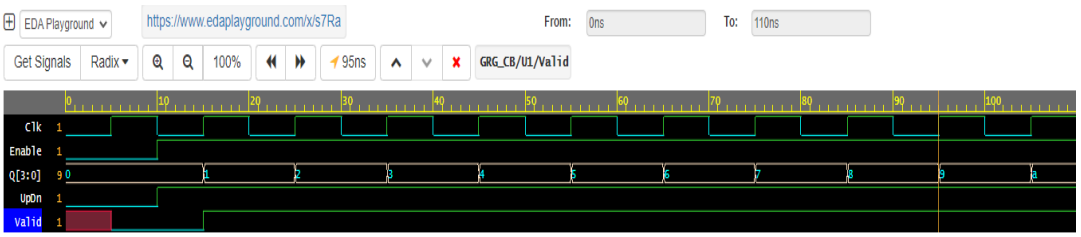


Figure 51. Waveform of the demonstrative testbench

## 8.17 Standards Usage in Project Execution

As previously stated, there are four standards that were part of this project. The SV and UVM standards were used during code development (syntax, classes, macros, …). Furthermore, the DFI and JEDEC JESD79-5A standards were used for feature extraction and verification plan development as they represent the design intent. Those standards served as the starting point from where the design and verification teams started their work and they are the returning destination whenever a “bug” is found to judge the validity of the bug.

# Chapter 9

# Results

## 9.1 Initial Coverage Report and Steps to improve it.

The coverage report is an output of simulation that shows how much of the Design was covered by the test that ran. Coverage reports of all the tests that we ran can be combined into one report that represents the coverage of all the tests. There are mainly two types of coverage in the report: code coverage and group coverage.

### 9.1.1 code coverage

Code coverage is automatically generated by the simulator. The simulator tries to figure out how much of the code is covered by the test. Code coverage includes Line coverage which shows whether each line of code was executed, condition coverage which measures the proportion of conditions within each decision expression that have been evaluated to both true and false, toggle coverage which shows whether each bit of a variable has gone through all transitions, FSM coverage which covers all FSM states and all possible transitions from one state to another, branch coverage which evaluates the branches that have been set to both true and false in testing, and assertion coverage which shows the percentage of the covered assertions during testing.

### 9.1.2 group coverage

Group coverage is a report of cover points that the verification team has written as part of their plan to cover certain features of the design.

### 9.1.3 initial coverage reports

After running some random tests and sanity tests at different ratios a coverage report was generated to analyze the coverage state and figure out how to improve the coverage report results.

| NAME | SCORE | LINE | COND | TOGGLE | FSM | BRANCH | ASSERT |
|---|---|---|---|---|---|---|---|
| uvm_pkg | 33.33 | | | | | | 33.33 |
| CountCalc | 48.79 | 50.00 | | 60.00 | | 36.36 | |
| FSM_setting | 56.25 | 43.75 | | 85.00 | | 40.00 | |
| ControlUnit | 60.42 | 50.00 | | 89.58 | | 41.67 | |
| tb_pkg | 61.54 | | | | | | 61.54 |
| EdgeDetectorFSM | 70.71 | 92.86 | | 50.00 | 50.00 | 90.00 | |
| pattern_detector | 79.73 | | | 79.73 | | | |
| FIFO | 81.96 | 86.96 | 73.68 | 87.18 | | 80.00 | |
| GapCounter | 85.30 | 96.30 | | 93.33 | 57.14 | 94.44 | |
| generic_FSM | 87.76 | 97.06 | | 80.56 | 77.78 | 95.65 | |
| CA_Manager | 92.33 | 97.73 | | 93.24 | 83.33 | 95.00 | |
| DataManager | 92.57 | 100.00 | | 85.14 | | | |
| DDR5_PHY | 92.99 | | | 92.99 | | | |
| ddr_assertions | 95.23 | | | 95.59 | | | 94.87 |
| ValidCounter | 95.56 | 100.00 | | 86.67 | | 100.00 | |
| top | 96.33 | | | 96.33 | | | |
| Serializer_V1 | 96.71 | 95.83 | 100.00 | 95.56 | | 95.45 | |
| Deserializer_V1 | 97.22 | 100.00 | | 91.67 | | 100.00 | |
| top_testbench | | | | | | | |

Figure 52. initial code coverage report Results

| NAME | SCORE | WEIGHT | GOAL | AT LEAST | PER INSTANCE | AUTO BIN MAX | PRINT MISSING | COMMENT |
|---|---|---|---|---|---|---|---|---|
| tb_pkg::subscriber::JEDEC_transitions | 0.00 | 1 | 100 | 1 | 0 | 64 | 64 | |
| tb_pkg::subscriber::DFI_transitions | 0.00 | 1 | 100 | 1 | 0 | 64 | 64 | |
| tb_pkg::subscriber::DFI_coverage | 88.89 | 1 | 100 | 1 | 0 | 64 | 64 | "Coverage model for the DFI features" |
| tb_pkg::subscriber::JEDEC_coverage | 98.68 | 1 | 100 | 1 | 0 | 64 | 64 | "Coverage model for the JEDEC features" |

Figure 53. initial group coverage report Results

### 9.1.4 suggested tests and improvements

The coverage report was analyzed and each cover hole was identified and a suggestion of how to deal with it was set.

Table 3. Initial coverage report analysis

| **Cover Hole** | **Description** | **Location** | **Suggested Improvement** |
|---|---|---|---|
| CRC enable | The CRC enable feature was not planned to be covered in the Verification plan as it was not | CountCalc, FIFO, CA_Manager, DataManager, | to be included in future work |

| | | | |
|---|---|---|---|
| | planned to be implemented by the design team. | DDR5_PHY, Deserializer_V1 Top modules | |
| Preamble patterns | some preamble patterns are still not covered | FSM_setting, ControlUnit, pattern_detector modules | more random tests |
| Interample patterns | some interamble patterns are still not covered | FSM_setting, ControlUnit, pattern_detector, DataManager, modules | running more random tests or directed back to back read tests |
| Reset and enable bits | Reset and enable signals are only triggered at the begging of the tests which causes some states not to be covered and some toggling holes for the reset and enable bits in multiple modules | EdgeDetectorFSM, pattern_detector, GapCounter, generic_FSM, CA_Manager, DataManager, Serializer_V1, Deserializer_V1 Top modules | Directed test for reset and enable in the middle of the simulation in different states. |
| FIFO data when empty and when full | a FIFO is used to store read commands and read them when data arrives. the case when the FIFO is full (more reads without data) and when the FIFO is empty (data without read) are not covered | FIFO module | future work |
| Uncovered states in generic_FSM | when reading a preamble pattern, only valid preamble patterns are covered (could be an unreachable state). | generic_FSM module | future work |
| CA toggling | not all CA bits are toggling | CA_Manager module | out of scope (should be excluded) |
| rd_data_en gap counter | wide enough distance between the rd_data_en pulses to trigger overflow bit in the counter | DataManager module | future work |

| | | | |
|---|---|---|---|
| frequency ratio bits toggling and default value | The bits for frequency ratio are covered but are not toggling because frequency ratio is set before the simulation starts. frequency ratio change protocol is not supported in DUT. | DDR5_PHY, Top, Serializer_V1, Deserializer_V1 modules | out of scope (should be excluded) |
| DDR_assertions | bit toggling coverage should be excluded from assertions module | ddr_assertions module | should be excluded |
| BL toggling | BL has specific values | ValidCounter, Serializer_V1 modules | should be excluded |
| command transitions | code for collecting coverage need to be rewritten | JEDEC_transitions, DFI_transitions groups | need to be reviewed |
| Deselect command | deselect command is not covered as it is not sent to the subscriber | DFI_coverage, JEDEC_coverage groups | should be excluded |
| MR40 | DQS offset mode register change is out of design scope | DFI_coverage, JEDEC_coverage groups | out of scope (should be excluded) |
| BL32_OTF | BL32_OTF burst length option is out of design scope | DFI_coverage group | out of scope (should be excluded) |
| Row corners | upper corner of Rows is not covered | DFI_coverage group | directed test constraining the random test to be at row and column corners |
| Column corners | lower corner of columns is not | DFI_coverage | directed test |

| | | | |
|---|---|---|---|
| | covered | group | constraining the random test to be at row and column corners |
| interamble patterns | some interamble patterns are not covered | JEDEC_coverage group | running more random tests or directed back to back read tests |

# 9.2 Final coverage report

## 9.2.1 Applied tests and improvements

The initial coverage report was analyzed and some steps were taken to improve it. First, another test was added to cover the rows and columns' corners. Then more random tests were run. Finally, we were able to exclude irrelevant coverage points from covergroups. However, we were not able to use exclusion files to exclude coverpoints in code coverage from the coverage report because we neither had a license for Verdi nor a license for using an exclusion file. So, we were limited to preventing coverage collection from a whole file or a certain type of group coverage in a file which we used to exclude all code coverage types except assertions from the ddr_assertions module and excluding the CRC_valid file for example.

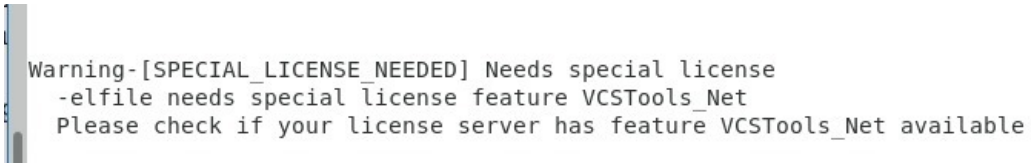


Figure 54. License Error that shows up when exclusion file is used

### 9.2.2 The final coverage report

The final coverage report was produced after running the sanity test 50 times for each frequency ratio, the random test 200 times for each ratio, and the random corners test 400 times for the 1:1 ratio and 50 times for the two other ratios.

**Total Module Definition Coverage Summary**

| SCORE | LINE | COND | TOGGLE | FSM | BRANCH | ASSERT |
|---|---|---|---|---|---|---|
| 91.38 | 95.58 | 100.00 | 92.10 | 77.14 | 94.15 | 89.29 |

Figure 55. final code coverage report Results summary

| NAME | SCORE | LINE | COND | TOGGLE | FSM | BRANCH | ASSERT |
|---|---|---|---|---|---|---|---|
| uvm_pkg | 33.33 | | | | | | 33.33 |
| CountCalc | 60.87 | 57.14 | | 80.00 | | 45.45 | |
| EdgeDetectorFSM | 74.04 | 100.00 | | 46.15 | 50.00 | 100.00 | |
| GapCounter | 84.47 | 96.30 | | 90.00 | 57.14 | 94.44 | |
| ddr_assertions | 88.89 | | | | | | 88.89 |
| Deserializer_V1 | 90.17 | 94.64 | | 86.98 | | 88.89 | |
| pattern_detector | 90.54 | | | 90.54 | | | |
| DDR5_PHY | 92.68 | | | 92.68 | | | |
| generic_FSM | 92.97 | 97.06 | | 90.28 | 88.89 | 95.65 | |
| FSM_setting | 94.03 | 93.75 | | 95.00 | | 93.33 | |
| CA_Manager | 95.16 | 100.00 | | 97.30 | 83.33 | 100.00 | |
| DataManager | 96.62 | 100.00 | | 89.86 | | 100.00 | |
| ValidCounter | 96.67 | 100.00 | | 90.00 | | 100.00 | |
| top | 96.79 | | | 96.79 | | | |
| ControlUnit | 97.92 | 100.00 | | 93.75 | | 100.00 | |
| FIFO | 98.40 | 100.00 | 100.00 | 93.59 | | 100.00 | |
| Serializer_V1 | 98.94 | 100.00 | 100.00 | 95.75 | | 100.00 | |
| tb_pkg | 100.00 | | | | | | 100.00 |
| top_testbench | | | | | | | |

Figure 56. final code coverage report Results

| NAME | SCORE | WEIGHT | GOAL | AT LEAST | PER INSTANCE | AUTO BIN MAX | PRINT MISSING | COMMENT |
|---|---|---|---|---|---|---|---|---|
| tb_pkg::subscriber::DFI_coverage | 100.00 | 1 | 100 | 1 | 0 | 64 | 64 | "Coverage model for the DFI features" |
| tb_pkg::subscriber::JEDEC_transitions | 100.00 | 1 | 100 | 1 | 0 | 64 | 64 | |
| tb_pkg::subscriber::JEDEC_coverage | 100.00 | 1 | 100 | 1 | 0 | 64 | 64 | "Coverage model for the JEDEC features" |
| tb_pkg::subscriber::DFI_transitions | 100.00 | 1 | 100 | 1 | 0 | 64 | 64 | |

Figure 57. final group coverage report Results

### 9.2.3 Improved cover points

Table 4. Coverage points Current state

| **Cover Hole** | **Current state** |
|---|---|
| CRC enable | Future work |
| Preamble patterns | covered with more tests |
| Interample patterns | covered with more tests |
| Reset and enable bits | Directed test for reset and enable in the middle of the simulation in different states is included in future work. |
| FIFO data when empty and when full | covered with more tests |
| Uncovered states in generic_FSM | future work |
| CA toggling | out of scope (should be excluded) |
| rd_data_en gap counter | future work |
| frequency ratio bits toggling and default | out of scope (should be excluded) |

| | |
|---|---|
| value | |
| DDR_assertions | excluded |
| BL toggling | should be excluded |
| command transitions | The transition coverage is collected by evaluating an expression as a cover point, and this expression was firstly implementing the complement of the required logic. Therefore, the coverage was zero although all cases were hit. so the transitions are actually covered. |
| Deselect command | excluded |
| MR40 | excluded |
| BL32_OTF | excluded |
| Row corners | covered by the corners test |
| Column corners | covered by the corners test |
| interamble patterns | covered by more tests |

### 9.2.4 Future coverage work

For the coverage report to be completely closed the license error needs to be fixed by installing the required license to be able to exclude the coverpoints that need to be excluded. Furthermore, a direct test is required to cover the enable and reset coverpoints. However, the need for a direct back-to-back read test is removed since the coverage holes that required them were covered by running more random tests. More testing for invalid inputs to observe DUT behavior and close coverage holes could be done in the future.

## 9.3 Bugs list

Some bugs were found during the environment development. Others were discovered while debugging the Environment with the sanity test. But most bugs were discovered while running the random test. We present below a sheet of these bugs, their description, a visual photo if available, steps to reproduce it, expected outcome vs actual outcome, and finally the current state of the bug. The design team had access to this sheet and had bugs to every new bug added. We received updated RTL from the design team with the bugs fixed as long as we were still debugging the environment. Later, we stopped receiving new RTL to collect the coverage and finalize the project so some bugs remained open. Table 5 shows the discovered bugs, their description, visual proof if available, and their current state.

Table 5. List of discovered bugs

| Bug ID | Description | Visual | Steps to reproduce | Expected vs actual results | State |
|---|---|---|---|---|---|
| 1 | CA bus bit ordering (CA[0:13] not CA[13:0]) | fig. 58 | Send any Command | CA [13:0] | Closed |
| 2 | rddata_valid signal is not synced with data when negedge driving | fig. 59 | | | Closed |
| 3 | reading at preamble setting 00001010 (PRE 4) doesn't return proper data | fig. 60 | reading at preamble setting 00001010 | | Closed |
| 4 | No proper data rotation | fig. 61 | read with 1 to 2 or 1 to 4 | | Open |
| 5 | DUT don't consider the BL_mode bit in the read command | - | read with default BL (BL mode == 1) at any Bl setting should produce BL16 | should be able to change BL on the fly | Open |
| 6 | Faulty dut if MR0 is not initialized, All MR | - | read without | | Invalid |

| | | | | | |
|---|---|---|---|---|---|
| | should be in an initial state on reset | | setting MR0 | | |
| 7 | DQS is on full cycle, should be toggling each half cycle | - | read CMD | should be toggling each half cycle thus it doesn't introduce interamble delays | Open |
| 8 | tctrl_delay is not consistent across frequency ratios | fig. 62 | Send any command | Command should arrive at the DRAM interface after tctrl_delay DFI clock cycles | Open |
| 9 | MRR is following BL settings in MR0 which is a bug as MRR has default BL16 | fig. 63 | MRR with non default BL | MRR should always have BL 16 | Closed |
| 10 | CS is LOW while reset, it should be high by default at reset | fig. 64 | reset | CS should not toggle with reset but remains HIgh | Closed |
| 11 | The CS goes from high to LOW (one cycle) after reset without a command on the DFI interface | fig. 65 | Assert Reset -> Deassert Reset -> Wait for 2 cycles | CS_n should not be driven LOW unless the dfi_cs was driven | Closed |
| 12 | Bl 32 reads alternate in actual returned BL between 16 and 32 | fig. 66 | MRW BL32 -> RD -> RD -> RD -> RD | All reads in BL32 should return BL 32 | Open |
| 13 | No data returned (at DFI, i.e. rddata_valid is not asserted) if the preamble settings is changed from 2 or 4 to any other setting | fig. 67 | MRW PRE2 or 4 >> MRW PRE 1, 3 >> read / MRR | Expected read data valid asserted on all cases of preambles | Open |
| 14 | Postamble settings are taken from OP[7] instead of OP[6] in MR8 | - | MRW (MR8) Postamble = 1 (OP[6] = 1) | Postamble settings should be taken from OP[6] in MR8 | Closed |

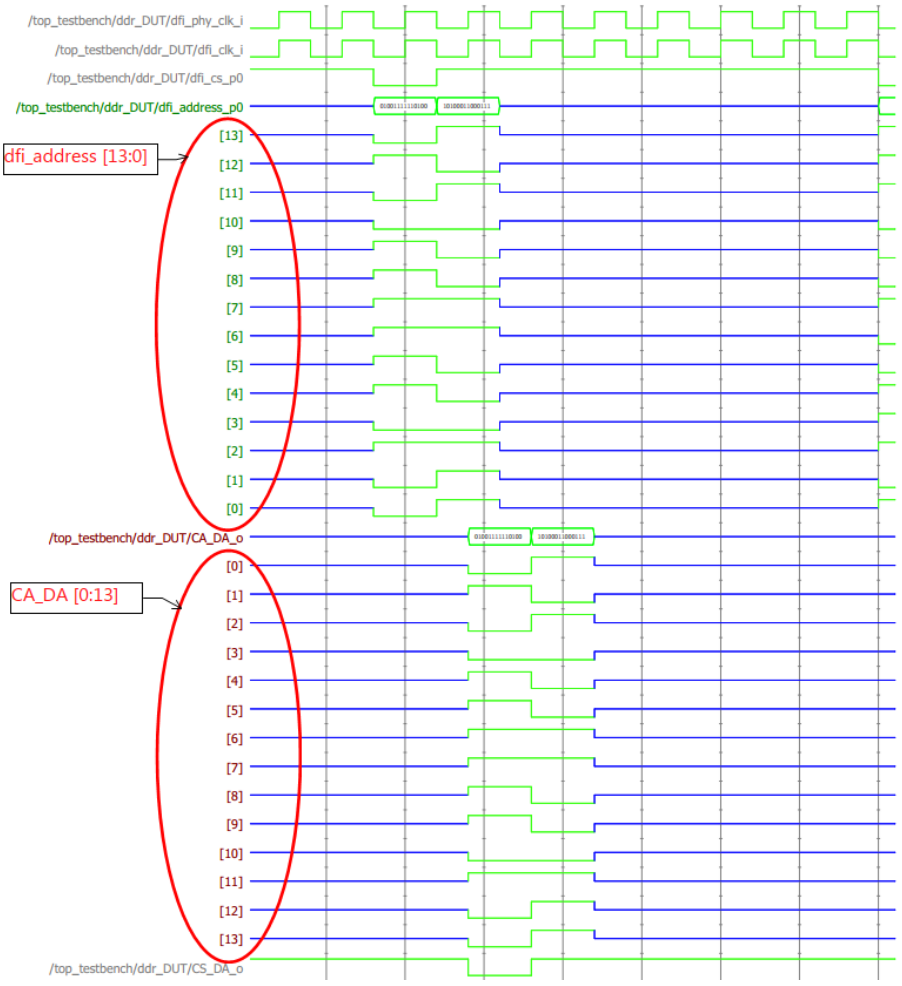


Figure 58. Bug 1

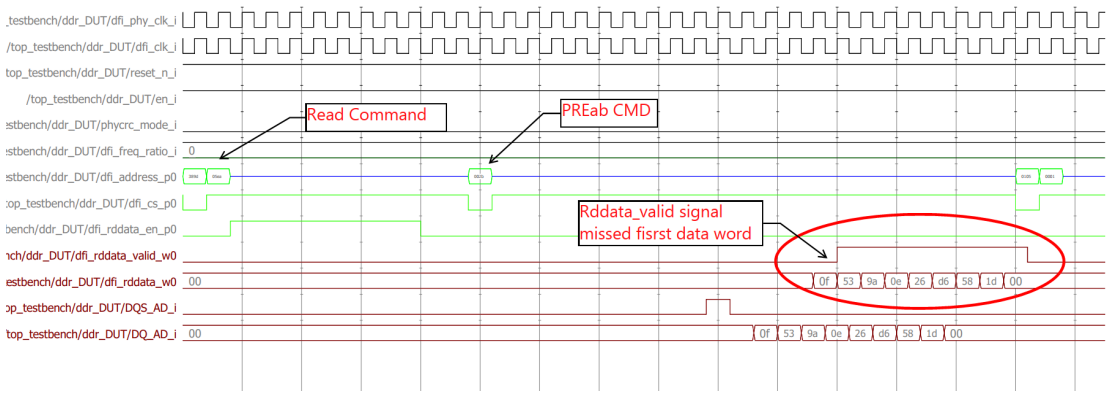


Figure 59. Bug 2

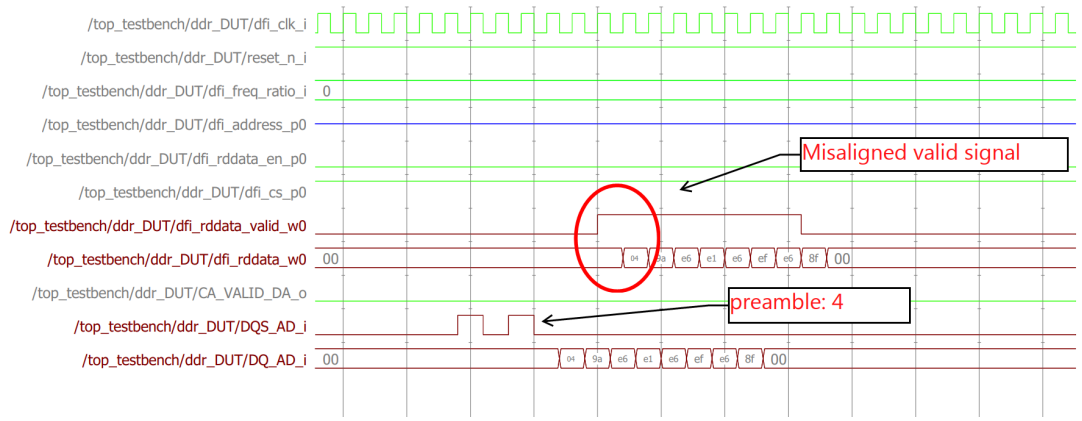


Figure 60. Bug 3

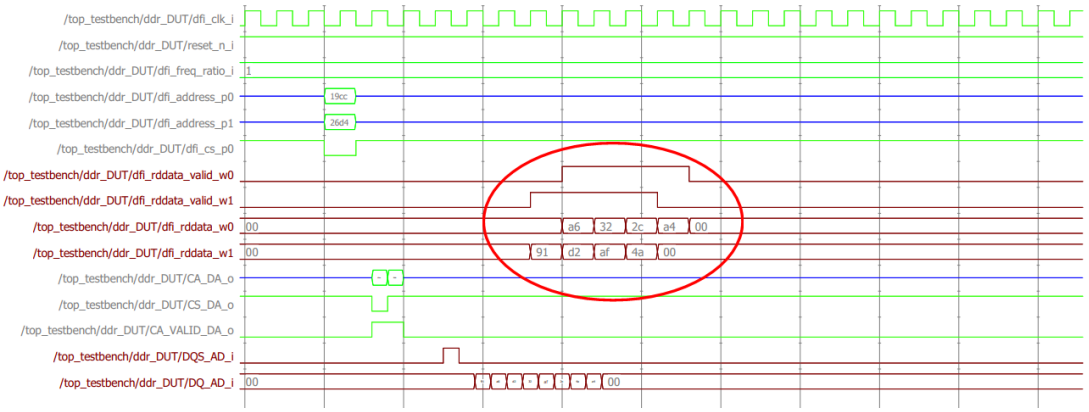


Figure 61a. Bug 4 (Previous Read Data)

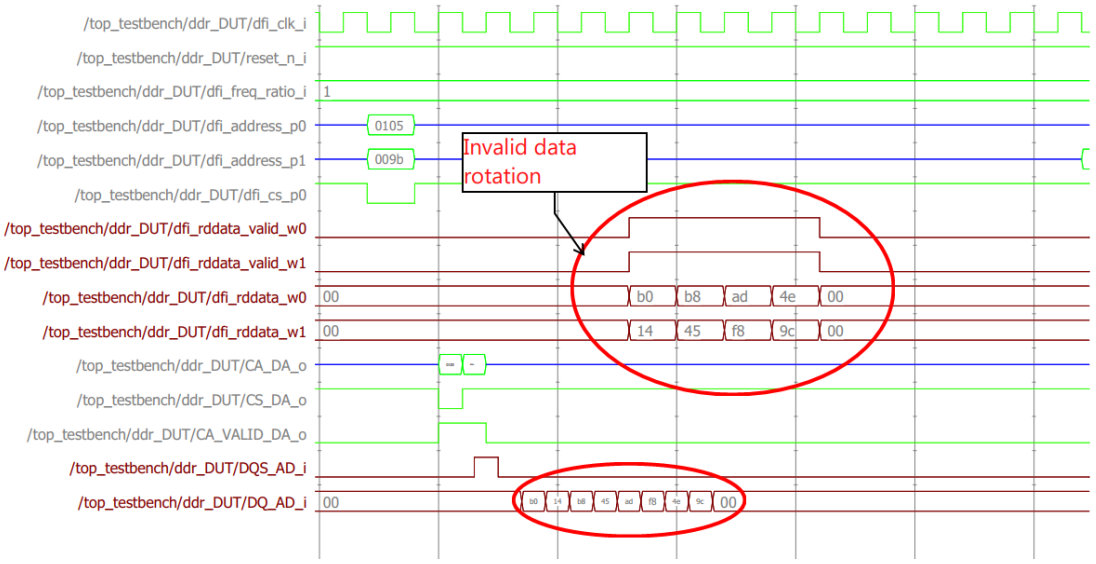


Figure 61b. Bug 4

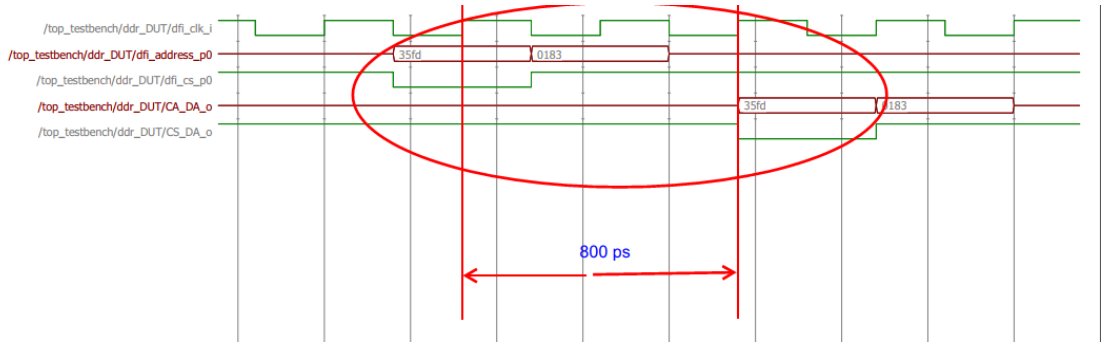


Figure 62a. Bug 8

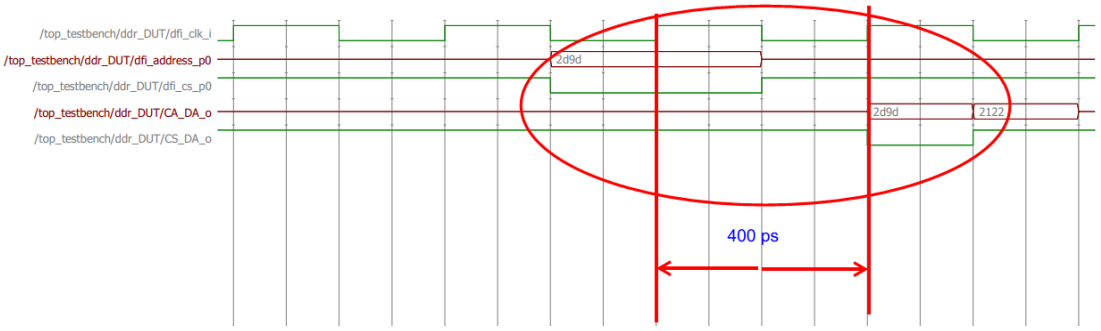


Figure 62b. Bug 8

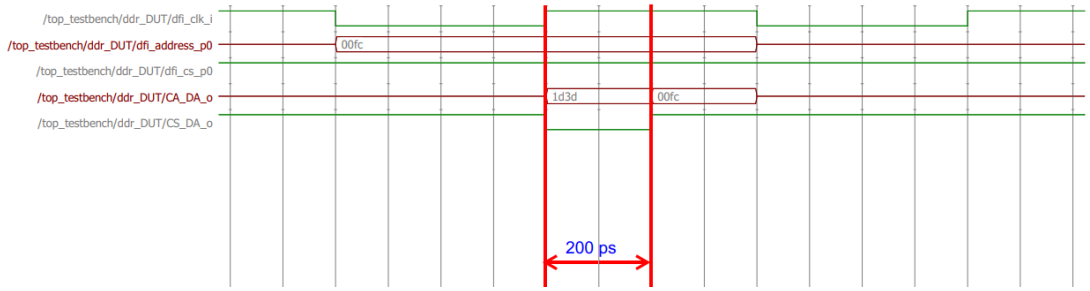


Figure 62c. Bug 8

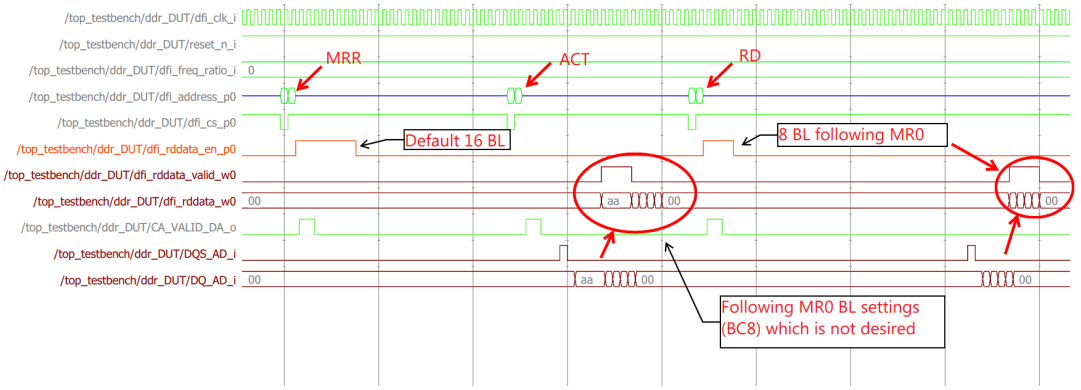


Figure 63. Bug 9

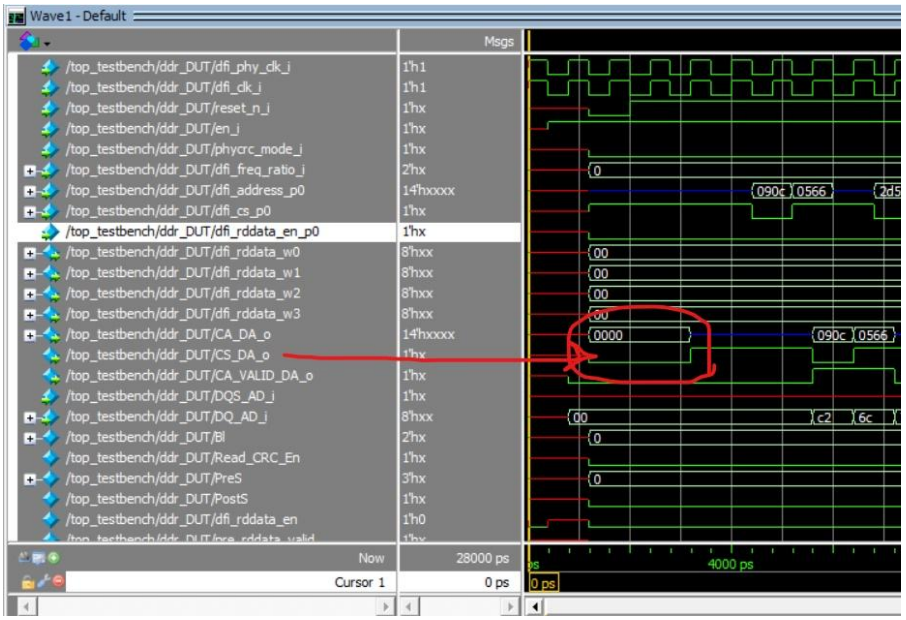


Figure 64. Bug 10

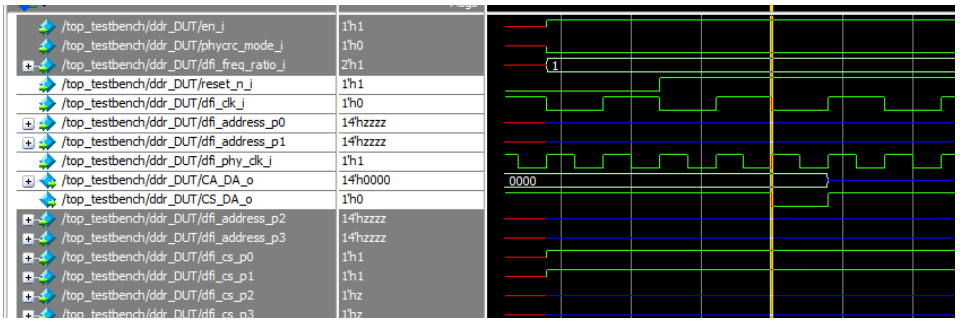


Figure 65. Bug 11

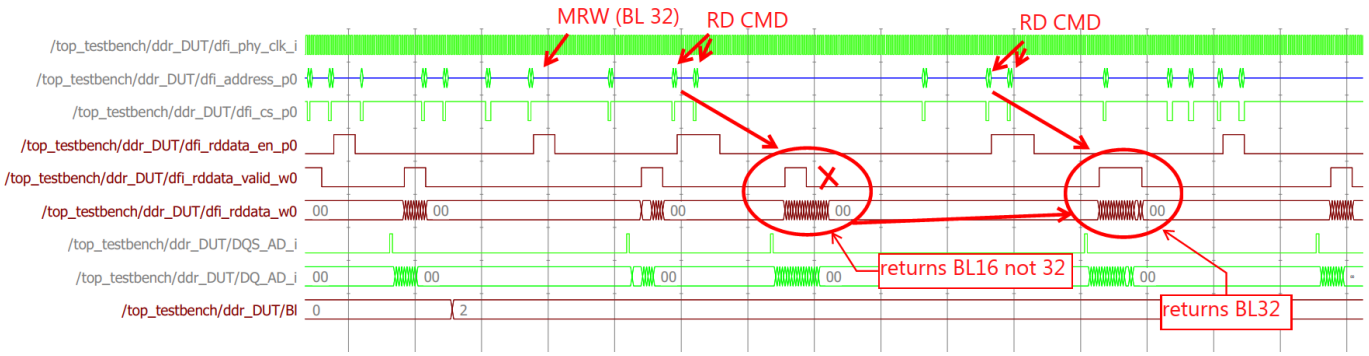


Figure 66. Bug 12

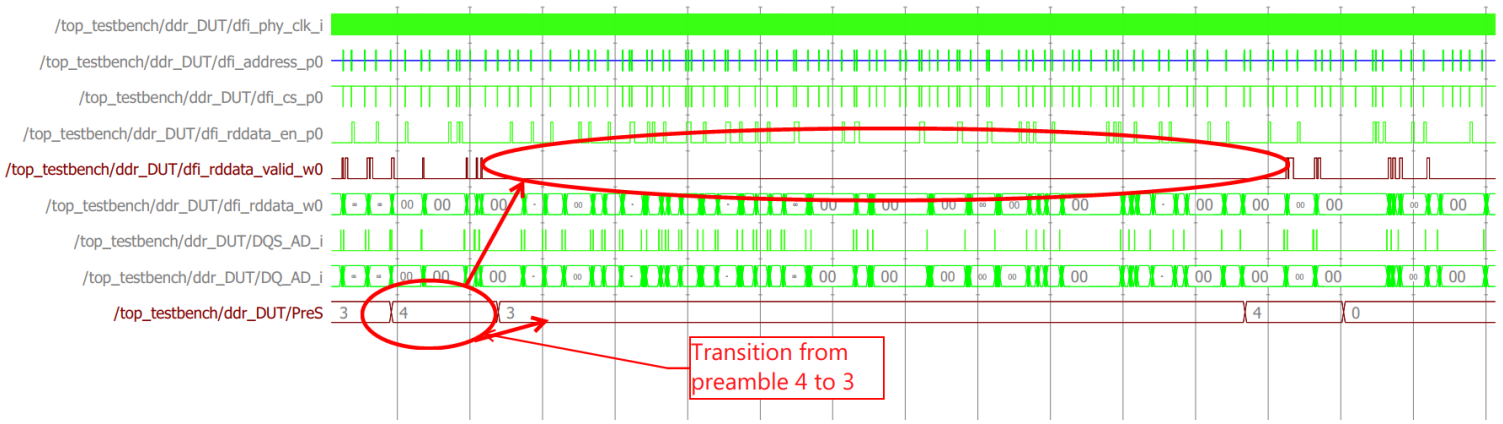


Figure 67a. Bug 13

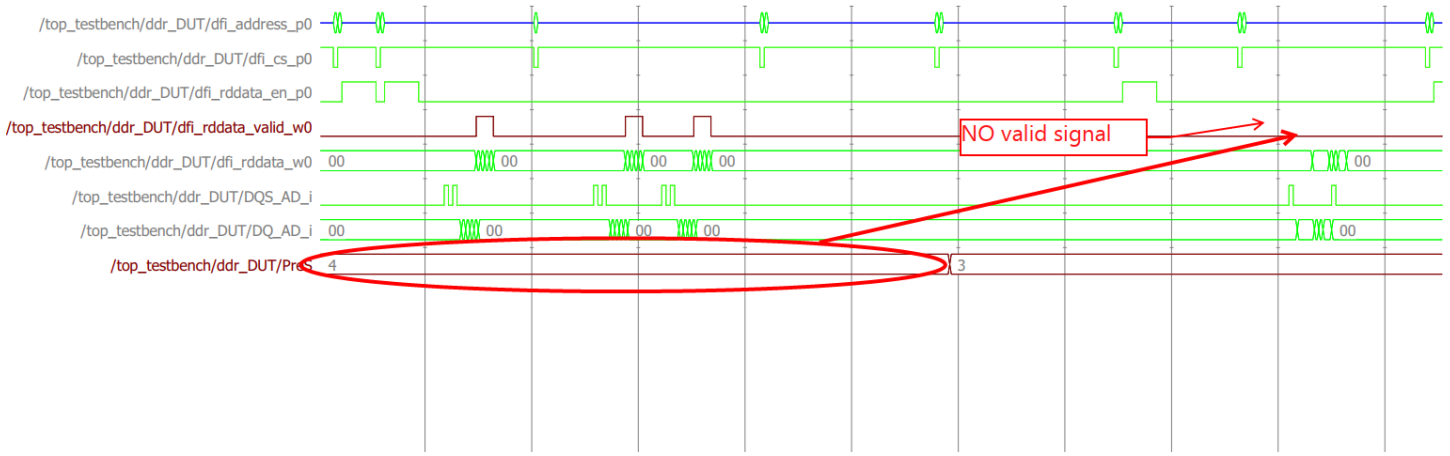


Figure 67b. Bug 13

# Chapter 10

# Cost Analysis

The work done in this project can be monetized as a verification IP that is used by designers working on DDR5 PHYs either as the sole product or as part of a system or SoC. Because the VIP consists of only code, the cost of development is mainly two things: the engineering effort and the cost of the EDA tools that were used during development. There are no parts of the products of this project that are manufacturable; thus, there are no concerns about the negative environmental impact or health and safety.

Because the cost of the EDA tools is fairly a constant, what makes the price of a VIP change is the following:

- How old is the code? Does it support the latest version of an engineering standard?
- How complex is the testbench? (depends on the standards/DUT)
- Does it utilize the latest verification methodologies and languages?
- What is the extent of reuse that the IP can provide?

All the factors mentioned above will affect the financial utility of the VIP. We believe that the work done in this project addresses all these points well:

- The testbench supports the latest JEDEC JESD79-5A and DFI standards for DDR5 PHYs.
- The testbench supports the complex operation described by the two standards.
- The testbench is written in SV and uses UVM in addition to SVA which is the de-facto methodology.
- The testbench is highly reusable as it adheres to the concepts of UVM and OOP.

# Chapter 11

# Conclusion and Future work

In this work, verification of the digital datapath of a DDR5 PHY was done. The flow started with extracting the features from the JEDEC JESD79-5A and DFI standards, which describe the two interfaces and functionality of the PHY. Consequently, a verification plan and test plan were developed to guide the verification effort. Then, a SystemVerilog UVM environment was developed with a supporting SVA module as a reusable testbench for the DDR5 PHY. Since the coverage-driven verification methodology was chosen, the testing continued until a satisfactory code and functional coverage results were obtained. Further work includes adding testing capabilities for more PHY functionalities as described in the JEDEC JESD79-5A and DFI standards in addition to developing a formal verification component of the testbench to make our testing more efficient and comprehensive.

# References

[1]“DRAM market anticipated to surpass USD 221.67 billion by 2030, with a CAGR of 9.2% - report by Market Research Future (MRFR),” GlobeNewswire News Room, 09-May-2022. [Online]. Available: https://www.globenewswire.com/en/news-release/2022/05/09/2438836/0/en/DRAM-Market-Anticipated-to-Surpass-USD-221-67-Billion-by-2030-with-a-CAGR-of-9-2-Report-by-Market-Research-Future-MRFR.html#:~:text=According%20to%20a%20comprehensive%20research,rate%20of%209.2%25%20by%202030. [Accessed: 17-Jun-2022].

[2] *DFI DDR PHY Interface,* 5.1, Cadence Design Systems, Inc., MAY 21, 2021

[3] *DDR5 SDRAM,* JESD79-5A, JEDEC SOLID STATE TECHNOLOGY ASSOCIATION, U.S.A., October 2021

[4] "IEEE Standard for Universal Verification Methodology Language Reference Manual." Available: 10.1109/ieeestd.2020.9195920 [Accessed 18 June 2022].

[5] W. Chen, S. Ray, J. Bhadra, M. Abadir and L. Wang, "Challenges and Trends in Modern SoC Design Verification", *IEEE Design & Test*, vol. 34, no. 5, pp. 7-22, 2017. Available: 10.1109/mdat.2017.2735383.

[6] Wagner, I.; Bertacco, V.; Austin, T. StressTest: An automatic approach to test generation via activity monitors. In Proceedings of the 42nd Design Automation Conference, Anaheim, CA, USA, 13–17 June 2005; pp. 783–788.

[7] Wang, F.; Zhu, H.; Popli, P.; Xiao, Y.; Bodgan, P.; Nazarian, S. Accelerating Coverage Directed Test Generation for Functional Verification: A Neural Network-Based Framework. In Proceedings of the Great Lakes Symposium on VLSI, ACM, New York, NY, USA, 23–25 May 2018; pp. 207–212

[8] Hughes, W.; Srinivasan, S.; Suvarna, R.; Kulkarni, M. Optimizing Design Verification using Machine Learning: Doing better than Random. In Proceedings of

the Design and Verification Conference (DVCON-Europe), Virtual Conference, 26–27 October 2021.

[9] R. Punnoose, R. Armstrong, M. Wong and M. Jackson, "Survey of Existing Tools for Formal Verification.", 2014. Available: 10.2172/1166644 [Accessed 22 February 2022].

[10] A. Souri and M. Norouzi, "A State-of-the-Art Survey on Formal Verification of the Internet of Things Applications", *Journal of Service Science Research*, vol. 11, no. 1, pp. 47-67, 2019. Available: 10.1007/s12927-019-0003-8.

[11] M. Loghi, T. Margaria, G. Pravadelli and B. Steffen, "Dynamic and Formal Verification of Embedded Systems: A Comparative Survey", *International Journal of Parallel Programming*, vol. 33, no. 6, pp. 585-611, 2005. Available: 10.1007/s10766-005-8911-2.

[12] S. Qamar, W. Butt, M. Anwar, F. Azam and M. Khan, "A Comprehensive Investigation of Universal Verification Methodology (UVM) Standard for Design Verification", *Proceedings of the 2020 9th International Conference on Software and Computer Applications*, 2020. Available: 10.1145/3384544.3384547 [Accessed 22 February 2022].

[13] “DDR5,” Cadence. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/verification-ip/simulation-vip/memory-models/dram/ddr5.html. [Accessed: 18-Jun-2022].

[14] “VC verification IP for DDR5,” *Synopsys*. [Online]. Available: https://www.synopsys.com/verification/verification-ip/memory/ddr5.html. [Accessed: 18-Jun-2022].

[15] “VC verification IP for DFI,” Synopsys. [Online]. Available: https://www.synopsys.com/verification/verification-ip/memory/dfi-verification-ip.html. [Accessed: 18-Jun-2022].

[16] B. Wile, J. Goss and W. Roesner, *Comprehensive functional verification the complete industry cycle*, 1st ed. Amsterdam: Elsevier/Morgan Kaufmann, 2005, pp. 8-446.

[17] A. Mehta, *ASIC/SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*, 1st ed. Springer Cham, 2018, pp. 1-148.

[18] J. Bergeron, *Writing testbenches using SystemVerilog*, 3rd ed. New York: Springer Science+Business Media, 2006, pp. 1-21.

[19] C. Spear, *System Verilog for Verification*. Springer US, 2008.

[20] *Universal Verification Methodology (UVM) 1.2 User's Guide*. Accellera Systems Initiative, 2022.

[21] M. Chen, "Brief introduction to verification methodology," DEV Community, 13-Feb-2022. [Online]. Available: https://dev.to/angelia/brief-introduction-to-verification-methodology-4lik. [Accessed: 18-Jun-2022].

[22] n.d. Universal Verification Methodology UVM Cookbook. [ebook] Siemens Digital Industries Software. Available at: <https://verificationacademy.com/cookbook/uvm> [Accessed 17 June 2022].

[23]"UVM TLM - Verification Guide", *Verification Guide*, 2022. [Online]. Available: https://verificationguide.com/uvm/uvm-t lm/. [Accessed: 18- Jun- 2022].

[24] B. Jacob, S. Ng and D. Wang, Memory systems. Burlington, MA: Morgan Kaufmann Publishers, 2010.

[25]"The history and future of DRAM architectures in different application domains – an analysis | imec", *Imec-int.com*, 2022. [Online]. Available: https://www.imec-int.com/en/imec-magazine/imec-magazine-june-2020/the-history-and-future-of-dram-architectures-in-different-application-domains-an-analysis. [Accessed: 18- Jun- 2022].

[26] Saxena, K., 2022. Next Generation Memory Interfaces. [online] p.17. Available at: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-115.html> [Accessed 17 June 2022].

[27] C. Cummings, “Applying Stimulus & Sampling Outputs ‐ UVM Verification Testing Techniques,” 2016. [Online]. Available: http://www.sunburst-design.com/papers/CummingsSNUG2016AUS_VerificationTimingTesting.pdf. [Accessed: 17-Jun-2022].

[28] C. Cummings, H. Chambers and S. D'Onofrio, *UVM Reactive Stimulus Techniques*. San Jose, CA, 2020.

# Appendices

GitHub Repo Link -

https://github.com/Intel-Insiders/ddr_gp.git

Verification Plan Link -

https://docs.google.com/spreadsheets/u/1/d/1K_4bOjD0W13mHrpJjebvUnYoPyZlF7W5J3NFmwm0BEU/edit#gid=930111970