



Faculty of Engineering
Cairo University



Cairo University

DDR5 SDRAM Memory Controller Design and Verification

Project Team

Belal Mohamed Ibrahim Fawzy
Saleh Ramadan Saleh Othman
Tarek Mohamed Sayed Hussein
Tarek Khaled Farouk Abdelshafi
Khaled Hassan Sayed Hassan
Abdelrahman Ahmed Saad Hassan

Supervised by

Dr. Hassan Mostafa

Sponsored by

Siemens EDA

A Graduation Project Thesis

In Partial Fulfillment of the Requirements for the

Degree of Bachelor of Science

Electronics and Electrical Communications Engineering

Faculty of Engineering, Cairo University

Giza, Egypt

July 2022

DDR5 SDRAM Memory Controller Design and Verification

Table of contents

Chapter 1: Introduction	12
1.1 Motivation	12
1.2 DRAM Background and DDR5	12
1.2.1 DRAM Basics	12
1.2.2 DDR5 Standard	15
1.3 HIGH COMPLEXITY AND THE NEED FOR VERIFICATION	16
1.4 Thesis scope	17
1.5 Thesis organization	17
Chapter 2: Design	18
2.1 Overview	18
2.1.1 Basic Features	18
2.1.2 Optional Features	19
2.1.3 Excluded Features	19
2.2 Scope of Our Design	19
2.3 Inputs and Outputs of Top Module	20
2.4 Operation of Controller	22
2.5 Design in Detail	23
2.5.1. Command_Decoder	23
2.5.2 Command_Address_FIFO	28
2.5.3 Write_Data_FIFO	29
2.5.4 Calculation of Depth of FIFOs	30
2.5.5 Counters	31
2.5.6 Command_FSM	34
2.5.7 Initialization_FSM	46
2.5.8 Self Refresh_FSM	51
2.6 Design Enhancement	56
2.6.1 Read_FSM	57
2.6.2 Write_FSM	60
2.6.3 Selection Decoder	64
Chapter 3: Introduction to Cocotb	67
3.1 Introduction	67

DDR5 SDRAM Memory Controller Design and Verification

3.2 Background	67
3.2.1 Functional Verification	68
3.2.2 Switching to Python.....	69
3.3 Design Verification using Cocotb	69
3.3.1 Architecture of Cocotb.....	70
3.3.2 Design Methodology	70
3.3.3 Cosimulation	71
3.4 Cocotb Coverage	71
3.4.1 Functional Coverage in SystemVerilog.....	71
3.4.2 Functional Coverage with Cocotb-coverage	72
3.4.3 Constrained Random Verification Features in SystemVerilog	73
3.4.4 Constrained Random Verification Features in cocotb-coverage	73
3.5 Code Coverage	74
3.5.1 Statement coverage/ Line coverage	75
3.5.2 Block/ Segment coverage	75
3.5.3 Conditional coverage.....	75
3.5.4 Branch coverage.....	75
3.5.5. Toggle coverage	75
3.5.6. Path coverage	75
Chapter 4: Block Level Verification	76
4.1 Goals and Overview	76
4.2 Block Level Testbench Architecture	76
4.3 Command Decoder Verification Plan	76
4.3.1 Functional Coverage Plan.....	76
4.3.2 Test Cases	77
4.3.3 Reported Bugs.....	80
4.3.4 Functional Coverage Results	83
4.3.5 Code Coverage Results	84
4.4 Command Finite State Machine	84
4.4.1 Functional Coverage Plan.....	84
4.4.2 Test Cases	85
4.4.3 Reported Bugs.....	94
4.4.4 Functional Coverage Results	98

DDR5 SDRAM Memory Controller Design and Verification

4.4.5 Code Coverage Results	98
4.5 Self_Refresh Finite State Machine	99
4.5.1 Functional Coverage Plan.....	99
4.5.2 Test Cases	99
4.5.3 Reported Bugs.....	100
4.5.4 Functional Coverage Results	100
4.5.6 Code Coverage Results	101
4.6 Initialization Finite State Machine	102
4.6.1 Functional Coverage Plan.....	102
4.6.2 Test Cases	102
4.6.3 Reported Bugs.....	103
4.6.4 Functional Coverage Results	103
4.6.5 Code Coverage Results	104
4.7 Counters	104
4.7.1 Functional Coverage Plan.....	104
4.7.2 Test Cases	104
4.7.3 Reported Bugs.....	105
4.7.4 Functional Coverage Results	105
4.7.5 Code Coverage Results	106
Chapter 5: UVM VS COCOTB	107
5.1 Introduction	107
5.1.1 UVM verification.....	107
5.1.2 UVM verification environment components	108
5.2 Verification Plan	108
5.2.1 Testing scenarios:	108
5.3 Results from UVM verification	109
5.4 Results from COCOTB verification	111
5.5 History of verification methods:	113
5.6 Tradeoffs between using UVM and COCOTB:	114
Chapter 6: Top Level Verification	117
6.1 Verification Environment	117
6.2 Functional Coverage Plan:	118
6.3 Reported Bugs:	119

DDR5 SDRAM Memory Controller Design and Verification

6.4 Functional coverage results:	121
6.5 Code coverage results:	122
6.6 Final Results	122
6.6.1 Initialization Sequence	122
6.6.2 Self-Refresh Sequence	123
6.6.3 Single Read after Write	123
6.6.4 Multiple Read after Multiple Write	123
6.7 QUESTA VERIFICATION IP	124
6.7.1 Overview	124
6.7.2 Memory Model Components	125
6.7.3 Configurations	126
Chapter 7: Conclusions and Future Work	129
7.1 Conclusion	129
7.2 Future Work	129
REFERENCES	130

DDR5 SDRAM Memory Controller Design and Verification

List of figures

Figure 1: DRAM Architecture.....	12
Figure 2 : Bandwidth Evolution of DRAM Standards.	14
Figure 3: Comparison of DDR4 and DDR5 Key Parameters.	16
Figure 4: Big Picture of Design.	18
Figure 5 : Inputs and Outputs of Top Module.....	20
Figure 6: Address Mapping.	21
Figure 7: Block Diagram of Top Module.	23
Figure 8: Block Diagram of Command Decoder.....	23
Figure 9: Block Diagram of Command_Address_FIFO.	28
Figure 10: Block Diagram of Write_Data_FIFO.....	29
Figure 11: Block Diagram of Counters.	31
Figure 12: Block Diagram of Command_FSM.....	34
Figure 13: Simplified State Diagram, JEDEC Reference.....	36
Figure 14: Timing Diagram for Read Burst Operation (BL16), JEDEC Reference.	37
Figure 15: Timing Diagram for Write Burst Operation (BL16), JEDEC Reference.	37
Figure 16: Command_FSM.....	39
Figure 17: Block Diagram of Initialization_FSM.....	46
Figure 18: Reset and Initialization Sequence at Power-on Ramping, JEDEC Reference.....	47
Figure 19: Initialization_FSM.....	48
Figure 20: Block Diagram of Self-Refresh_FSM.....	51
Figure 21: Self-Refresh Entry/Exit Timing with One-Cycle Exit Command, JEDEC Reference.	52
Figure 22: Self-Refresh_FSM.....	53
Figure 23: Clock Generator isn't synthesizable.....	56
Figure 24: Problem of ambiguous clock triggering.....	56
Figure 25: Block Diagram of Read_FSM.....	57
Figure 26: Read_FSM.....	58
Figure 27: Block Diagram of Write_FSM.....	60
Figure 28: Write_FSM.....	61
Figure 29: Multiple Driven Problem.....	63
Figure 30: Block Diagram of Selection_Decoder.....	64
Figure 31: Some Problems of Synthesis.....	66
Figure 32: Schematic from Synthesis Tool (Vivado).	66
Figure 33: Types of Functional Verification.....	68
Figure 34:Architecture of Cocotb.....	70
Figure 35: An example of the coverage tree structure.....	72
Figure 36: Block Level Verification Environment.	76
Figure 37: Command decoder coverage section written in python.	77
Figure 38: Initialization FSM Enable is asserted high for only one cycle.	81
Figure 39: Initialization FSM Enable after modification.....	82
Figure 40: Command Decoder Test Summary from Questasim.	82
Figure 41: Command decoder Functional Coverage XML Report.....	83
Figure 42: Command decoder Functional Coverage from Coverage Viewer.	83
Figure 43: Command decoder Code Coverage Summary from Questasim.	84

DDR5 SDRAM Memory Controller Design and Verification

Figure 44: Single write operation stuck at wait write done state.....	94
Figure 45: Single write operation after modifying CMD FSM.	95
Figure 46: DQS, DQT preambles aren't working properly.	96
Figure 47: DQS, DQT preambles are working properly after modification.....	96
Figure 48: CMD FSM gets stuck at wait_tRP state.....	97
Figure 49: CMD FSM works properly after modification.	97
Figure 50: Command FSM Functional Coverage from Coverage Viewer.....	98
Figure 51: Command FSM Code Coverage Summary from Questasim.	98
Figure 52: Self_Refresh sequence as specified in JESD79-5.	99
Figure 53: implemented Self_Refresh waveform.	100
Figure 54: Self-Refresh FSM Functional Coverage XML Report.....	100
Figure 55: Self-Refresh FSM Functional Coverage from Coverage Viewer.....	101
Figure 56: Self-Refresh FSM Code Coverage Summary from Questasim.	101
Figure 57: Initialization sequence as specified in JESD79-5.....	102
Figure 58: implemented initialization waveform.....	103
Figure 59: Initialization FSM Functional Coverage XML Report.....	103
Figure 60: Initialization FSM Functional Coverage from Coverage Viewer.	103
Figure 61: Initialization FSM Code Coverage Summary from Questasim.	104
Figure 62: Counter Flag is asserted High for only one Clock Cycle.	105
Figure 63: Counters Functional Coverage XML Report.....	105
Figure 64: Counters Functional Coverage from Coverage Viewer.....	106
Figure 65: Counters Code Coverage Summary from Questasim.	106
Figure 66: Counters Test Summary from Questasim.....	106
Figure 67: UVM environment for WR_Data_FIFO block.....	107
Figure 68: 1st example of error from simulation.....	109
Figure 69: 2nd example of error from simulation.	109
Figure 70: UVM Report Summary and Coverage.....	109
Figure 71: Assertions and UVM results.....	110
Figure 72: The assertions coverage.	110
Figure 73: Result of errors from COCOTB.	111
Figure 74: COCOTB verification results.....	112
Figure 75: COCOTB Functional coverage report.....	112
Figure 76: COCOTB Code coverage report.....	113
Figure 77: Constrained-random test progress over time vs. directed testing.....	114
Figure 78: Programming languages Complexity.	115
Figure 79: FPGA verification language adoption next twelve months	116
Figure 80: ASIC/IC verification language adoption next twelve months	116
Figure 81 : Verification Environment of Top-Level Verification.....	117
Figure 82: Block Diagram of Memory Module.....	118
Figure 83:Top Module is stuck at WAIT_ACT state.....	119
Figure 84: Read words with length less than burst length	120
Figure 85: Tri-state Buffer.....	121
Figure 86:Functional Coverage Report of Top Level.....	121
Figure 87:Code Coverage Report of Top Level	122

DDR5 SDRAM Memory Controller Design and Verification

Figure 88:Waveform of Initialization Sequence from Top Level	122
Figure 89:Waveform of Self-Refresh Sequence from Top Level.....	123
Figure 90: Waveform of Single Read after Write.....	123
Figure 91: Waveform of Multiple Read after Multiple Write	123
Figure 92: Questa Memory Model Flow	124
Figure 93:Memory Model Components	125
Figure 94: Test bench using QVIP Configurator	126
Figure 95 :Memory Model Configurations.....	127

DDR5 SDRAM Memory Controller Design and Verification

List of tables

Table 1: Inputs of Top Module.....	20
Table 2: Outputs of Top Module.....	21
Table 3: Inputs of Command Decoder	23
Table 4: Outputs of Command Decoder	24
Table 5: Description of Processor Commands	25
Table 6: Decoding of Processor Commands	25
Table 7: Description of Enable Signals of Finite State Machines	26
Table 8: Description of Enable Signals of FIFOs	27
Table 9: Inputs of Command_Address_FIFO.	28
Table 10: Outputs of Command_Address_FIFO	28
Table 11: Inputs of Write_Data_FIFO	29
Table 12: Outputs of Write_Data_FIFO	30
Table 13: Initialization Timing Parameters	31
Table 14: Activate Timing Parameters.....	32
Table 15: Precharge Timing Parameters.....	32
Table 16: Reading Timing Parameters	32
Table 17: Writing Timing Parameters	32
Table 18: Self-Refresh Timing Parameters.....	33
Table 19: Inputs of Command_FSM.....	34
Table 20: Outputs of Command_FSM.....	34
Table 21: Scenarios of Back-to-Back Operations	37
Table 22:Storage of Activate Rows	38
Table 23: Description of each state in Command_FSM.....	39
Table 24: Outputs of each state in Command_FSM	42
Table 25: Inputs of Initialization_FSM	46
Table 26: Outputs of Initialization_FSM	47
Table 27: Description of each state in Initialization_FSM.....	48
Table 28: Outputs of each state in Initialization_FSM.....	49
Table 29: Inputs of Self Refresh_FSM	51
Table 30: Outputs of Self Refresh_FSM	51
Table 31: Description of each state in Self Refresh_FSM	53
Table 32: Outputs of each state in Self Refresh_FSM.....	54
Table 33: Inputs of Read_FSM	57
Table 34: Outputs of Read_FSM	57
Table 35: Description of each state in Read_FSM	58
Table 36: Outputs of each state in Read_FSM.....	59
Table 37: Inputs of Write_FSM	60
Table 38: Outputs of Write_FSM	60
Table 39: Description of each state in Write_FSM	61
Table 40: Outputs of each state in Write_FSM.....	62
Table 41: Inputs of Selection Decoder	64
Table 42: Outputs of Selection Decoder	64
Table 43: Operation of Selection Decoder.....	64

DDR5 SDRAM Memory Controller Design and Verification

Table 44: Test cases of command decoder.....	77
Table 45: Test cases of command FSM.....	85
Table 46: Test cases of Self_Refresh FSM.....	99
Table 47: Test cases of Initialization FSM.....	102
Table 48: Test cases of counters.....	104
Table 49: Description of Memory Model Components	125

DDR5 SDRAM Memory Controller Design and Verification

List of Abbreviation

SDRAM: Synchronous Dynamic Random Access Memory

DDRx: Double Data rate xth generation

SoC: System on Chip

MPSoC: Multi-Processor System on Chip

DIMM: Dual Inline Memory Module

ACT: Activate Command

PRE: Precharge Command

RD: Read Command

RDA: Read Command with Auto-Precharge

WR: Write Command

WRA: Write Command with Auto-Precharge

NOP: No Operation Command

PREpb: Precharge bank Command

PREsb: Precharge same bank Command

PREab: Precharge all banks Command

REFsb: same-bank Refresh

CRC: Cyclic Redundancy Check

MPSM: Maximum Power Saving Mode

Cocotb: Co-routine Co-simulation test-bench

UVM: Universal Verification Methodology

FIFO: First-In First-Out

FSM: Finite State Machine

HDL: Hardware Description Language

RTL: Register Transfer Level

IC: Integrated Circuit

FPGA: Field Programmable Gate Array

EDA: Electronic Design Automation

DUT: Design Under Test

GUI: Graphical User Interface

VPI: Verilog Procedural Interface

VHDLPI: VHDL Procedural Interface

API: Application Programming Interface

IP: Intellectual Property

QVIP: Questa Verification IP

DDR5 SDRAM Memory Controller Design and Verification

Chapter 1: Introduction

1.1 MOTIVATION

Currently, we see a strong need to memory-dominated Applications such as Big Data, IoT, Cloud-Data Centers, Machine Learning etc...., Thus Dynamic Random-Access Memories (DRAMs) play a large role in compute platforms. Over the last years, the number of DRAM standards specified by the JEDEC Solid State Technology Association has been growing rapidly. The most recent DRAM standard is DDR5, which was released in mid-2020, because of the large number of new features, system designers are either challenged to adopt the new standard or they can move on with well-established standards like DDR4. If DDR5 is a potential candidate for a specific application, a further challenge is the configuration of the DDR5 subsystem, which is the configuration of the DDR5 subsystem, which features a lot of parameter choices. Fast and accurate simulation models are mandatory to explore the new features and compare different configurations.

1.2 DRAM BACKGROUND AND DDR5

In this section we introduce the basic terminology of DRAM devices and their controllers and give an overview on the new features of the DDR5 standard.

1.2.1 DRAM Basics

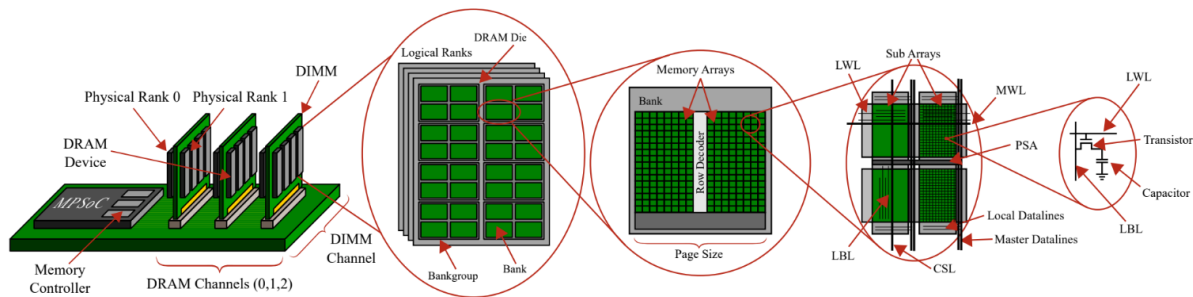


Figure 1: DRAM Architecture [1].

As shown in Figure 1, DRAM can be organized in a multi-hierarchical fashion of DIMMs, channels, physical ranks, devices, logical ranks, bank groups, banks, memory arrays, sub arrays, rows, and columns. Several DRAM channels can be connected to the Multi-Processor System on Chip (MPSoC). These channels are completely in-dependent of each other and have separate command/address and data buses. A channel can be composed of one or multiple physical ranks, which are sharing the data and command/address bus. A Dual Inline Memory Module (DIMM) is a small PCB that accommodates several DRAM devices, which work completely synchronously. One single device is called $\times 161$ if it has an I/O data width of 16 bit. A DIMM is assembled for

DDR5 SDRAM Memory Controller Design and Verification

instance out of four $\times 16$ devices in order to have a total I/O data width $n = 64$ bit (called $\times 64$). While the I/O data width is usually very limited, inside the DRAM a lot of data can be fetched or stored in parallel. However, the time between consecutive internal data accesses is very long due to the optimization for storage density, while the interface can be operated on much higher frequencies. To align this mismatch DRAM uses a so-called prefetching technique. For read large chunks of data are concurrently fetched to the interface and then transferred in one burst to the re1pronunciation: by-sixteen quester, for a write the process is reversed. In addition, data is transferred at the doubled interface frequency (double data rate, short DDR). Current devices such as DDR4 use an $8n$ prefetch architecture, where n is the I/O data width, 8 the Burst Length (BL) and $8n$ the number of bits for an internal data transfer. That means with each DRAM access the total amount of data received or delivered is $BL \cdot n = 8 \cdot 64 \text{ bit} = 512 \text{ bit} = 64 \text{ B}$, which is the usual cache line size in today's computing systems. In combination with interface frequencies up to 1600 MHz or pin transfer rates up to 3200 MT/s (mega transfers per second) DDR4 reaches a maximum bandwidth of 25.6 GB/s per channel. Each device itself can consist of several 3D-stacked logical ranks, which can form several bank groups that include several banks. The concept of bank groups was introduced with DDR5 and DDR4 in order to reduce the bank switching times to support a seamless burst behavior at high data rates and therefore a high bandwidth. All banks in a whole channel can be used concurrently (so called bank parallelism). However, there are some constraints due to the shared buses. Each bank usually consists of 212 to 218 rows, and each row can usually store 512B to 2KB of data in its columns. A memory controller is composed of a front end and a back end. The front end performs arbitration and scheduling of incoming read and write requests, whereas the task of the back end is to translate these incoming requests into a sequence of DRAM commands, which have to be orchestrated with respect to the current state of the device. To access data in a row of a certain bank, an activate (ACT) command must be issued by the controller before any column access, i.e., read (RD) or write (WR) commands, can be executed. The ACT command opens an entire row of the memory array, which is transferred into the bank's row buffer [2]. It acts like a small cache that stores the most recently accessed row of the bank. The latency of a memory access to a bank largely varies depending on the state of this row buffer. If a memory access targets the same row as the currently cached row in the buffer (called row hit), it results in a low latency and low energy memory access.

DDR5 SDRAM Memory Controller Design and Verification

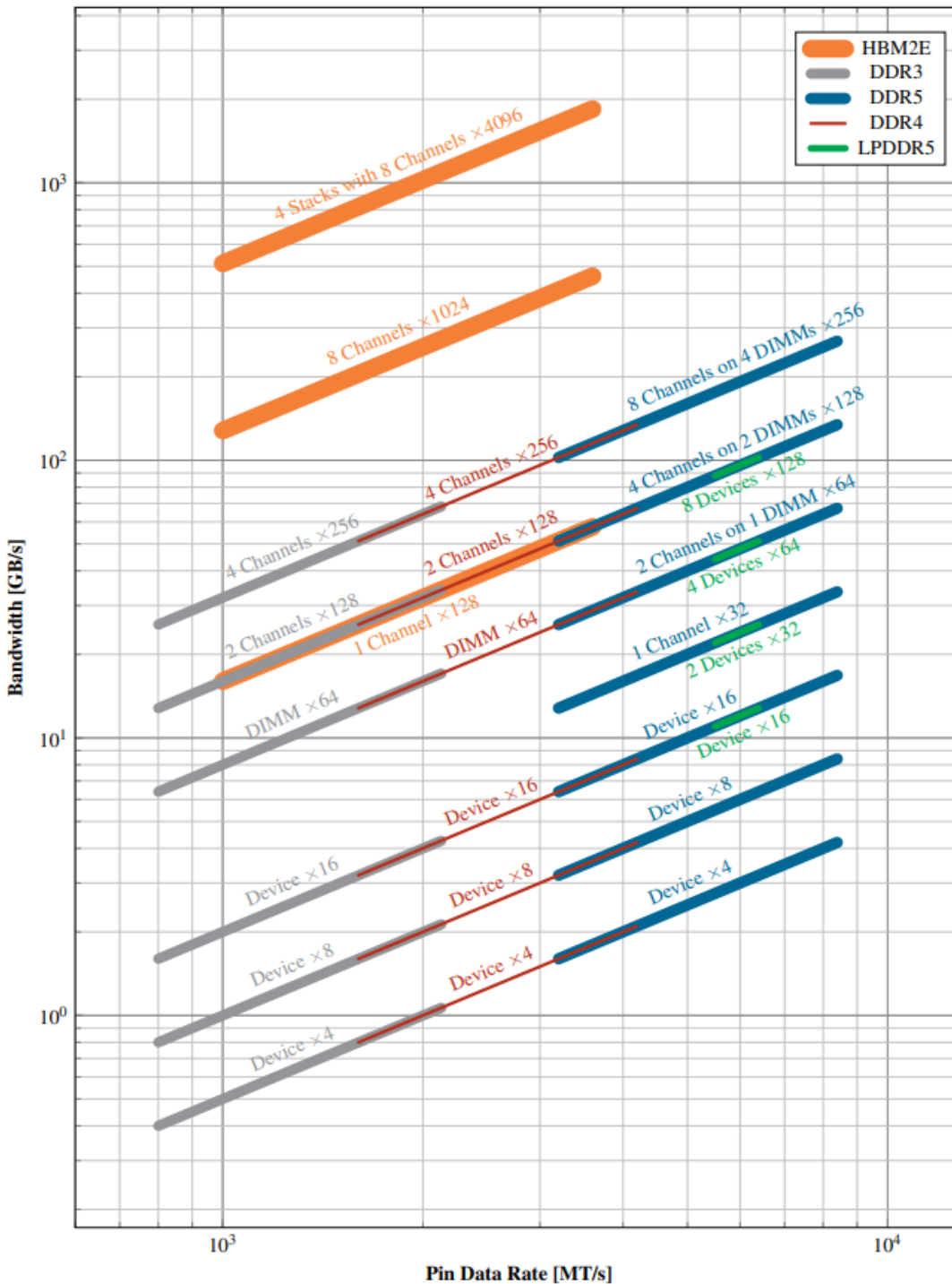


Figure 2 : Bandwidth Evolution of DRAM Standards [1].

Whereas, if a memory access targets a different row as the current row in the buffer (called row miss), it results in a higher latency and energy consumption. If a certain row in a bank is active it must first be precharged (PRE) before another row can be activated. In addition to the normal RD and WR commands, there exist read and write commands with an integrated auto-precharge (RDA,

DDR5 SDRAM Memory Controller Design and Verification

WRA). If auto-precharge is selected, the row being accessed will be precharged automatically at the end of the read or write access. Because a DRAM cell uses a capacitor with leakage effects for data storage, it usually has to be refreshed every 64ms to retain the data stored in it. Modern DRAMs are equipped with an all-bank refresh (REFab) command to perform this operation automatically on all banks of a rank in parallel. However, a prerequisite is that all banks are in a precharged state. This can be achieved by issuing a special all-bank precharge (PREab) command in advance. In addition to the commands, each DRAM standard defines a set of timing dependencies, which are temporal constraints that must be satisfied between issued commands. For example, between two ACT commands to the same bank the timing dependency tRC (row cycle time) must be satisfied. Timing dependencies can also exist on other hierarchies of the DRAM, e.g., between commands to the same bank group, to the same logical/physical rank or to different logical/physical ranks. The selection of a DRAM subsystem usually has three main dimensions: bandwidth, latency, and capacity. Bandwidth is the amount of data that can be transferred between DRAM and a computational unit within a given time. As shown in Figure 2, the maximum theoretical DRAM bandwidth is limited to the number of data pins times the interface pin data rate (number of accesses per time per pin). Latency is the time that it takes to complete an access. In fact, latency helps bandwidth, but not vice versa [4]. For instance, lower DRAM latency results in more accesses per time, and therefore higher bandwidth, whereas increasing the number of data pins increases the bandwidth without decreasing latency. In realistic scenarios, the full theoretical bandwidth is never reached due to many timing dependencies, interference between different requests, and refresh. The actual achieved bandwidth for a specific application is called sustainable bandwidth.

1.2.2 DDR5 Standard

With the development of a new DRAM standard generation there are always several key parameters that should be enhanced, e.g., bandwidth, power consumption, and device capacity. Figure 3 shows a comparison between key parameters of the new DDR5 standard and its predecessor DDR4. In the following we will also describe the most important differences in more detail. For a higher bandwidth DDR5 raises the maximum pin data rate to 8400 MT/s compared to 3200 MT/s for DDR4. Because the frequency of internal data accesses stays more or less the same as a result of the capacity- and cost optimized architecture, the prefetch was incremented from 8n to 16n. When using the same 64-bit-wide data bus for one channel as all previous DDR generations, this would result in 128 B of transferred data per access. However, since the usual cache line size of modern processors is only 64 B, the data bus of each DDR5 DIMM is split up into two independent channels of 32-bit width. That way only 64 B of data are transferred per access. Theoretical transfer rates then reach a maximum of 33.6 GB/s per channel and 67.2 GB/s per DIMM compared to 25.6 GB/s per channel/DIMM for DDR4, as shown in Figure 2. At the same time supply voltages are reduced from 1.2 V to 1.1 V for an improved power consumption. The maximum number of banks per device increases from 16 to 32 distributed over 8 instead of 4 bank groups, the total capacity of a single device from 16 Gb to 64 Gb. In addition, up to 16 instead of 8 devices can now be stacked in a three-dimensional fashion (logical ranks)³. This enables stack capacities of up to 512 Gb (max. 16×32 Gb or 8×64 Gb because of limited address bits). One

DDR5 SDRAM Memory Controller Design and Verification

problem that always arises with higher device capacities is the increased refresh overhead, because each cell still has to be refreshed approximately every 64 ms, as a consequence, either the controller has to issue refresh commands more frequently or the individual refresh cycles take a longer time. Since all banks of a rank cannot be accessed during an all-bank refresh, it can lead to significant performance drops. To overcome this problem, DDR5 introduces same-bank refresh (REFsb) and associated same-bank precharge (PREsb) commands as an alternative to all-bank refresh (REFab) and all-bank precharge (PREab) commands. When issuing them, only one bank in each bank group of the target rank is refreshed and inaccessible, while all other banks can still process incoming read and write requests. Most modern DRAM controllers use advanced reordering techniques for an improved performance so they can try to hide the same-bank refresh by sending requests to other banks in the meantime. Finally, DDR5 devices implement an on-die error correction to improve the data integrity[2].

ITEMS	DDR4	DDR5
Frequency	1600~3200Mbps	3200~8400Mbps
Density	2Gb, 4Gb, 8Gb, 16Gb	8Gb, 16Gb, 24Gb, 32Gb, 64Gb
On die ECC	No	Yes
Bank	16banks	32banks
VDD/VDDQ	1.2V	1.1V
VPP	2.5V	1.8V
BL	8	16
DFE	No	Yes
Same bank refresh	No	Yes

Figure 3: Comparison of DDR4 and DDR5 Key Parameters [3].

1.3 HIGH COMPLEXITY AND THE NEED FOR VERIFICATION

With advanced and complex features, there is a need for meticulous verification. Memories have a vast set of configurations that allow them to operate at various data rates with different densities. Further, these can be combined with a vast set of features such as Self-Refresh, Auto Refresh, Cyclic Redundancy Check (CRC), Post Package Repair, Maximum Power Saving Mode (MPSM), training across different settings of latencies and speeds. The permutation and combinations of these variables can grow exponentially across different memory vendors as each of these memory

DDR5 SDRAM Memory Controller Design and Verification

vendors offer 100s of part numbers. As you can imagine, it can easily become a daunting experience for verification engineers to verify a memory subsystem.

DDR5 DIMMs adds more challenges. For example, in order to achieve higher power efficiency, voltage is reduced from 1.2V to 1.1V which brings additional complexity for DIMM vendors around noise immunity. Higher speeds raise data integrity concerns which need precise training results. This creates the need for careful verification, as modeling of real-world scenarios and visualization of those scenarios going into wire level toggling would consume a lot of time.

Measuring verification progress through functional coverage is equally important as the verification of design features. Also, performance analysis is extremely important for memories. These add a few extra cycles in the verification flow.

1.4 THESIS SCOPE

1.Design of DDR5 SDRAM Memory Controller based on JEDEC79-5 Standard.

2.Verification of DDR5 SDRAM Memory Controller using Python Language especially CoCotb library (new trend method in verification).

1.5 THESIS ORGANIZATION

The structure of the thesis is as follows:

- Chapter 2: This Chapter discusses the design of DDR5 SDRAM Memory Controller.
- Chapter 3: This Chapter presents an introduction to Cocotb and discusses The Verification Methodology.
- Chapter 4: This chapter outlines the results of Block level verification phase.
- Chapter 5: A Comparison between UVM Verification and A Cocotb Verification is discussed in this chapter.
- Chapter 6: This chapter outlines the results of Top-level verification phase.
- Chapter 7: The chapter consists of the conclusions and details possible future work

DDR5 SDRAM Memory Controller Design and Verification

Chapter 2: Design

2.1 OVERVIEW

In this chapter, we will discuss design of **DDR5 SDRAM Controller**. Our design will interface with CPU and DDR5 SDRAM Memory as shown in Figure 4:

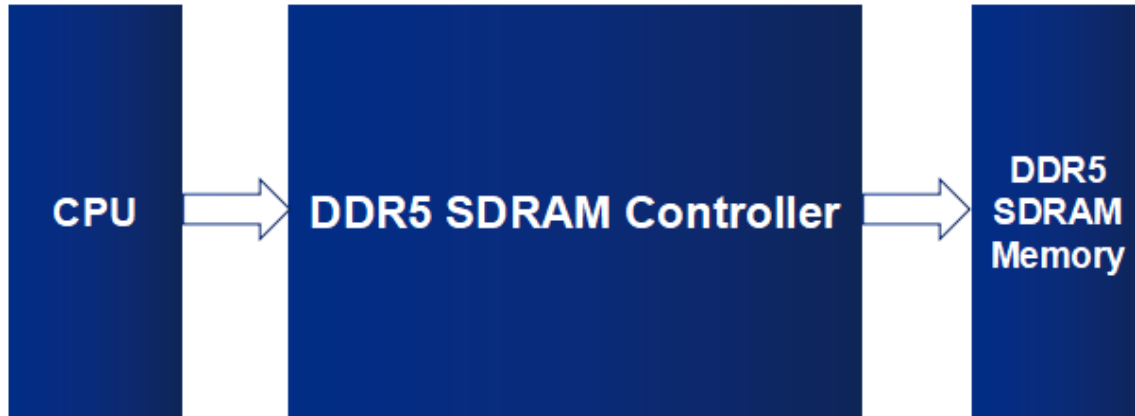


Figure 4: Big Picture of Design.

to interface with CPU, we followed **Native Interface** and to interface with DDR5 SDRAM Memory we followed **JEDEC79-5 Standard**. For DDR5 SDRAM Memory, we chose **x16** Configuration with **3.2GHZ** input clock frequency. This choice will affect the values of timing parameters as we will illustrate. Not all the features of DDR5 SDRAM Memory in JEDEC79-5 Standard are scoped in our project. To choose features that will be implemented in our design, we divided features into three sections:

2.1.1 Basic Features

- Initialization
- Activation
- Precharge
- Reading
- Writing
- Auto Precharge
- Burst
- Self-Refreshing

DDR5 SDRAM Memory Controller Design and Verification

- Mode Register Operations

2.1.2 Optional Features

- On-Die ECC (Check Memory Error)
- DDR5 ECC Transparency and Error Scrub
- CRC (Check Bus Error)
- Write Pattern Command
- Refreshing
- Power Down Mode

2.1.3 Excluded Features

All modes related to testing, training modes such as ZQ Calibration, Read Training, Change Clock Frequency, etc.

2.2 SCOPE OF OUR DESIGN

As a first step in design, we started our design by **basic features** mentioned above in section 2.1.1 and after we verify these operations, we will extend our project based on available time with other optional features of DDR5. There are also some modes we mentioned in section 2.1.3 we excluded them as there are related if we completed our design as a product and we need to test it in hardware so we excluded them from our design as we will not reach in our project to fabrication. To implement basic features these are the memory commands that will be used:

- Activate (ACT)
- Precharge (PREpb)
- Read (RD)
- Read w/Auto Precharge (RDA)
- Write (WR)
- Write w/Auto Precharge (WRA)
- Mode Register Write (MRW)
- Self_Refresh Entry (SRE)
- NOP

DDR5 SDRAM Memory Controller Design and Verification

➤ Deselect (DES)

2.3 INPUTS AND OUTPUTS OF TOP MODULE

After we chose features that will be implemented, we could determine inputs and outputs of our design from Native interface and JEDEC79-5 standard as shown in Figure 5:

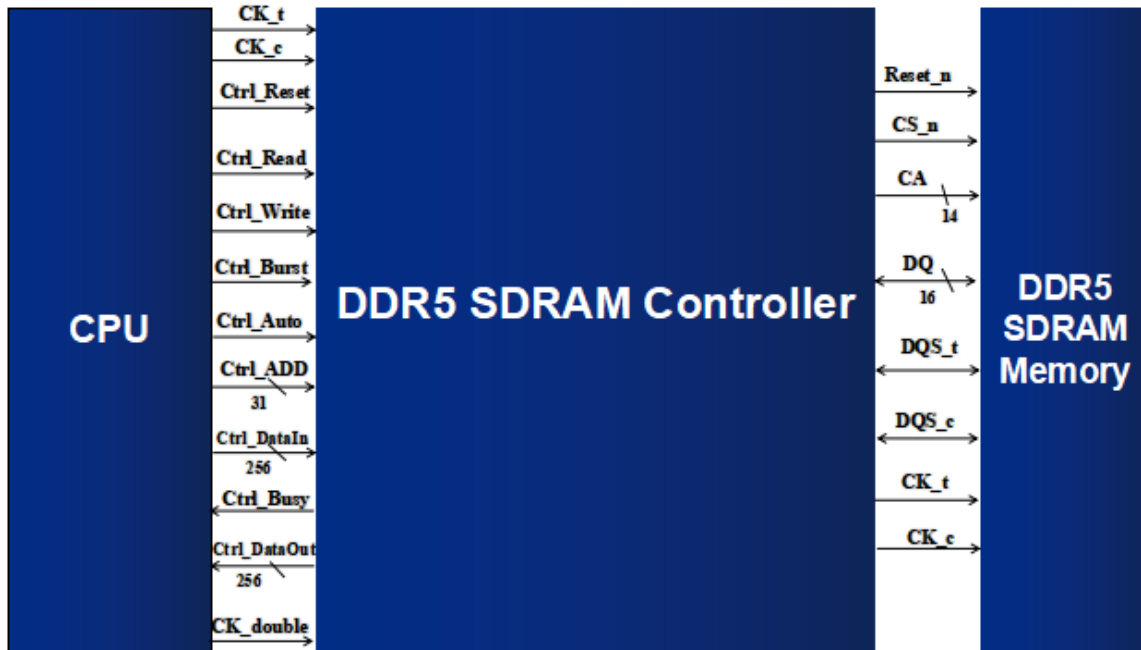


Figure 5 : Inputs and Outputs of Top Module.

The description of each input is shown in Table 1, and also description of each output is shown in Table 2:

Table 1: Inputs of Top Module [4]

Signal	Description
CK_t, CK_c	Differential clock of DDR5 SDRAM, all address and control input signals are sampled on the crossing of the positive edge of CK_t and negative edge of CK_c.
Ctrl_Reset	Control signal from CPU to controller to issue the initialization sequence to memory. It's active high reset
Ctrl_Read	Read request (active high).
Ctrl_Write	Write request (active high).
Ctrl_Burst	When 1 defines alternate read/write burst mode (BC8) and when 0 defines default BL16 mode.

DDR5 SDRAM Memory Controller Design and Verification

Ctrl_Auto	When asserted high along with Ctrl_Read or Ctrl_Write, causes the command to be issued as read with auto-precharge and write with auto-precharge, respectively.
Ctrl_ADD[30:0]	Address that data will be read from memory or written in memory.
Ctrl_DataIn[255:0]	Data that will be written in memory during writing operation.
CK_double	Clock that has double frequency of SDRAM clock, DQS signals will be generated with positive edge of this clock, also data from memory will be sampled also on the positive edge of this clock

Table 2: Outputs of Top Module [5]

Signal	Description
Reset_n	Active low asynchronous reset: reset is active when reset_n is low, and inactive when reset_n is high. reset_n must be high during normal operation and takes some values during initialization sequence.
CS_n	Chip Select: all commands are masked when CS_n is registered high. For one cycle commands (CS_n=0), for two cycle commands (CS_n=0 for first cycle, CS_n=1 for second cycle).
CA[13:0]	Command/Address Inputs: CA signals provide the command and address inputs according to the command truth table in JEDEC79-5 standard section 4.1 Table 241.
DQ[15:0]	Data Input/Output: Bi-directional data bus
DQS_t,DQS_c	Data Strobe: output with read data, input with write data. Edge-aligned with read data, centered in write data. DDR5 SDRAM supports differential data strobe only and does not support single-ended.
Ctrl_DataOut[255:0]	Data will be delivered from memory to CPU.
Ctrl_Busy	Control signal is sent to CPU to tell it to stop sending reading or writing requests as FIFO of commands became full or when there is initialization or self refresh for memory.

❖ Assumptions & Notes

1. We assume that Physical address that comes from CPU (Ctrl_ADD) will be 31 bit and address mapping will be one to one mapping as shown in Figure 6:



Figure 6: Address Mapping [5].

DDR5 SDRAM Memory Controller Design and Verification

2. According to Native Interface data bus (Ctrl_DataIn, Ctrl_DataOut) is eight times the width of the SDRAM device data bus (DQ) but for simplicity we made data bus (Ctrl_DataIn, Ctrl_DataOut) is 16 times the width of the SDRAM device data bus (DQ) as default BL.

2.4 OPERATION OF CONTROLLER

To know blocks and detailed block diagram of our design, we could get this by defining operation of our design. We can summarize operation of memory controller in five points as follow [7]:

1. Memory controller is responsible for **queuing** requests from CPU like read and write requests and also responsible for scheduling these requests to choose which request will be executed.
2. It's responsible for **decoding** these requests into **memory commands** and issue these commands to memory keeping **timing constraints** between these commands according to period of clock of SDRAM.
3. In case of read request, it's responsible for getting data from memory to CPU and vice versa in write request.
4. As we talk about dynamic memory so it will need refresh every certain time, memory controller is responsible for issuing **self-refresh sequence** to memory.
5. Memory controller is also responsible for issuing **initialization sequence** to memory.

From red highlighted keywords we can know which blocks that we need in our design to achieve operation of DDR5 SDRAM Controller:

Queuing: Command_Address_FIFO, Write_Data_FIFO

Decoding: Command_Decoder

Memory Commands: Command_FSM

Timing Constraints: Counters

Self-Refresh Sequence: Self Refresh_FSM

Initialization Sequence: Initialization_FSM

So, block diagram of our design will be as shown in Figure 7 [8]:

DDR5 SDRAM Memory Controller Design and Verification

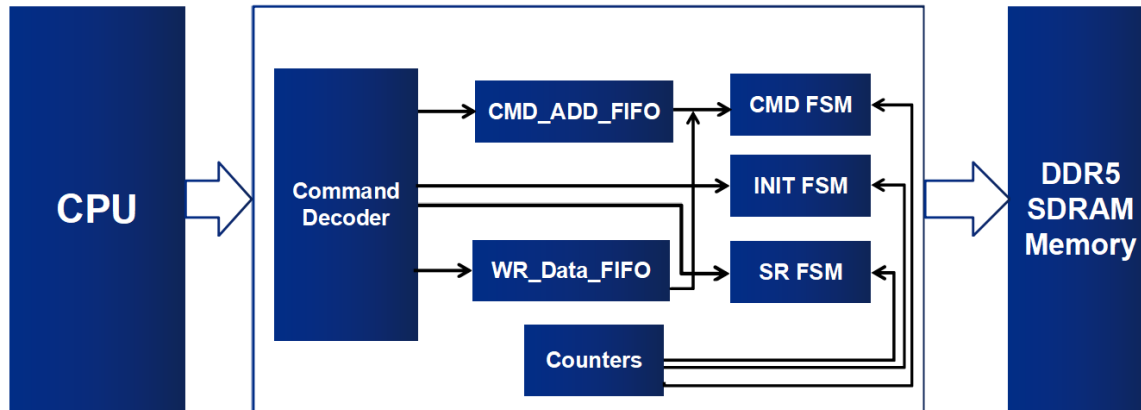


Figure 7: Block Diagram of Top Module.

2.5 DESIGN IN DETAIL

2.5.1. Command_Decoder

2.5.1.1. Block Diagram

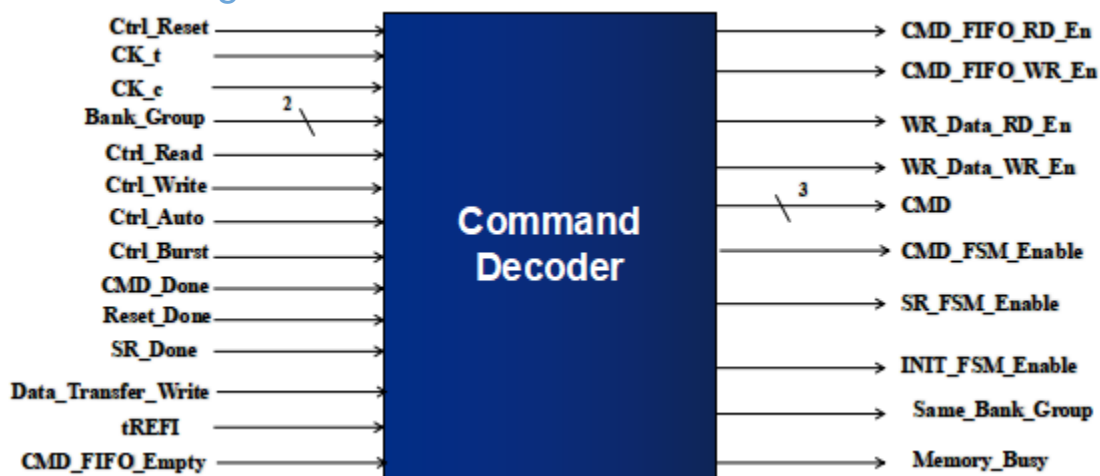


Figure 8: Block Diagram of Command Decoder.

2.5.1.2. List of Inputs and Outputs

Table 3: Inputs of Command Decoder

Signal	Description
Ctrl_Reset	Control signal from CPU to controller to issue the initialization sequence to the SDRAM. It's active high reset
CK_t, CK_c	Differential clock of DDR5 SDRAM, all address and control input signals are sampled on the crossing of the positive edge of CK_t and negative edge of CK_c
Bank_Group[1:0]	Bank group address, it is Ctrl_ADD[1:0]
Ctrl_Read	Read request (active high)
Ctrl_Write	Write request (active high)

DDR5 SDRAM Memory Controller Design and Verification

Ctrl_Burst	When 1 defines alternate read/write burst mode (BC8) and when 0 defines default BL16 mode
Ctrl_Auto	When asserted high along with Ctrl_Read or Ctrl_Write, causes the command to be issued as read with auto-precharge and write with auto-precharge, respectively
CMD_Done	When 1 defines that command has been executed
Reset_Done	When 1 defines that initialization sequence has been executed
SR_Done	When 1 defines that self-refresh sequence has been executed
Data_Transfer_Write	Input signal from command FSM, when 1 defines that we need to set WR_Data_RD_En high to write data from WR_Data_FIFO to memory
tREFI	Flag signal of self-refresh counter to issue the self-refresh sequence to memory
CMD_FIFO_Empty	Empty flag of Command_Address FIFO

Table 4: Outputs of Command Decoder

Signal	Description
CMD_FIFO_RD_En	Active high read enable of Command_Address_FIFO
CMD_FIFO_WR_En	Active high write enable of Command_Address_FIFO
WR_Data_RD_En	Active high read enable of Write_Data_FIFO
WR_Data_WR_En	Active high write enable of Write_Data_FIFO
CMD[2:0]	Defines type of command (read/write) that will be executed by Command_FSM.
CMD_FSM_Enable	Active high enable of Command_FSM
INIT_FSM_Enable	Active high enable of Initialization_FSM
SR_FSM_Enable	Active high enable of Self Refresh_FSM
Same_Bank_Group	When 1 defines that bank group address of current command is the same as address of previous command
Memory_Busy	When 1 indicates that memory in initialization or self-refresh sequence. When 0 indicates that memory in normal operation (reading or writing)

2.5.1.3 Operation

Command Decoder is the brain of our design, we can say it's the controller of our controller so it will be responsible for four points as follow:

A. Decode Processor Commands:

Command Decoder decodes processor commands based on control signals from CPU (Ctrl_Read, Ctrl_Write, Ctrl_Burst, Ctrl_Auto) and sets signal CMD with code equivalent to the decoded command, the description of each processor command and it's equivalent code are shown in Table 5:

DDR5 SDRAM Memory Controller Design and Verification

Table 5: Description of Processor Commands

Command	Description	CMD
Read	Read operation with default BL16 mode	000
Write	Write operation with default BL16 mode	001
Read_With_AutoPrecharge	Read operation with autoprecharge with default BL16 mode	010
Write_With_AutoPrecharge	Write operation with autoprecharge with default BL16 mode	011
Read_Burst	Read operation with BC8 mode	100
Write_Burst	Write operation with BC8 mode	101
Read_Burst_AutoPrecharge	Read operation with autoprecharge with BC8 mode	110
Write_Burst_AutoPrecharge	Write operation with autoprecharge with BC8 mode	111

The decoded command based on control signals from CPU is shown in Table 6:

Table 6: Decoding of Processor Commands

Decoded Command	Ctrl_Read	Ctrl_Write	Ctrl_Burst	Ctrl_Auto
Read	1	0	0	0
Write	0	1	0	0
Read_With_AutoPrecharge	1	0	0	1
Write_With_AutoPrecharge	0	1	0	1
Read_Burst	1	0	1	0
Write_Burst	0	1	1	0
Read_Burst_AutoPrecharge	1	0	1	1
Write_Burst_AutoPrecharge	0	1	1	1

❖ Assumption & Notes:

As we saw that we have four control signals so we have 16 combinations of values, we showed only in Table 6 eight of them but the last eight combination will have Ctrl_Read=1 and Ctrl_Write=1 and this is an error from CPU and unintentional but our design deals with this error by decoding this case as read command as a default command but don't store this command in Command_Address_FIFO.

B. Detect Same Bank Group:

Bank_Group signal represents bank group address; Command Decoder compares this address with address of previous command and set Same_Bank_Group signal high if they are equal. We need Same_Bank_Group signal in our design as the value of timing parameters between memory commands depends on if the two commands have the same bank group address or not.

DDR5 SDRAM Memory Controller Design and Verification

C. Enable FSM:

As we made three separate FSMs (Command, Initialization, Self-Refresh), so we made three enable signals (CMD_FSM_Enable, INIT_FSM_Enable, SR_FSM_Enable) to choose only one FSM to work and disable the other two FSMs during working of the selected FSM. There is also signal called Memory_Busy must be set high during working of INIT_FSM or SR_FSM. We can say that we have three operations Initialization, Self-Refresh and Normal Operation (reading and writing), so we should set priority for executing these operations. Priority of enabling them will be in this order: Initialization - Self Refresh - Normal. The description of each enable signal is shown in Table 7:

Table 7: Description of Enable Signals of Finite State Machines

Signal	Description
INIT_FSM_Enable	Command Decoder should set this signal high when comes from CPU Ctrl_Reset signal then set it low to disable INIT_FSM after Reset_Done becomes high noting that Reset_Done will be input from INIT_FSM that is raised high when initialization sequence finishes. When INIT_FSM_Enable becomes high, Memory_Busy signal is raised high
SR_FSM_Enable	Command Decoder should set this signal high when comes from self-refresh counter tREFI signal, then set it low to disable SR_FSM after SR_Done becomes high noting that SR_Done will be input from SR_FSM that is raised high when self-refresh sequence finishes. When SR_FSM_Enable becomes high,Memory_Busy signal is raised high
CMD_FSM_Enable	Command Decoder should set this signal high when reading enable of Command_Address_FIFO becomes high then set it low to disable CMD_FSM after Memory_Busy comes high or CMD_Done becomes high noting that CMD_Done will be input from CMD_FSM that is raised high when command finishes. When CMD_FSM_Enable becomes high,Memory_Busy signal is set low

D. Read &Write Enable Signals for FIFOs:

Command decoder here will be responsible for driving values of enable signals of FIFOs. The description of each enable signal is shown in Table 8:

DDR5 SDRAM Memory Controller Design and Verification

Table 8: Description of Enable Signals of FIFOs

Signal	Description
CMD_FIFO_RD_En	Command Decoder will set this enable signal high when Command_Address_FIFO is not empty and there isn't initialization or self-refresh process running (i.e., Memory_Busy=0) and pervious command has been executed. Command Decoder will set this signal low in the next cycle of setting it high.
CMD_FIFO_WR_En	Command Decoder will set this signal high when comes from CPU read or write request, so Command Decoder will decode this request to command and set this enable signal high to store this command in Command_Address_FIFO and to handle error that we said before comes read and write request in the same time, Command Decoder will ignore this by not storing this command by disable this write enable signal in this case. Command Decoder will also set this signal low in the next cycle of setting it high.
WR_Data_RD_En	We defined signal called Data_Write_Transfer comes from Command_FSM as output of WRITING_DATA state to indicate that we need activate WR_Data_RD_En to store this data in memory so Command Decoder will set read enable of Write_Data_FIFO high when this signal is raised high and set it low when this signal is lowered low.
WR_Data_WR_En	Command Decoder will act also as CMD_FIFO_WR_En signal but this time, it will set this signal high when the request is write to store data that comes with write request in Write_Data_FIFO and also will set it low in the next cycle of setting it high.

❖ Assumption & Notes

Command Decoder set these enable signals low in the next cycle of setting it high to avoid unwanted read or write operation from FIFO, this thing isn't applied on WR_Data_RD_En signal as we don't want to disable reading from Write_Data_FIFO in next cycle ,we need read from it more than one cycle as memory takes data from controller with width of DQ bus(16 bit) and can't take 256 bit in one cycle. So, Write_Data_Transfer signal will be responsible for setting this enable signal low when transfer data from controller to memory finishes.

DDR5 SDRAM Memory Controller Design and Verification

2.5.2 Command_Address_FIFO

2.5.2.1 Block Diagram

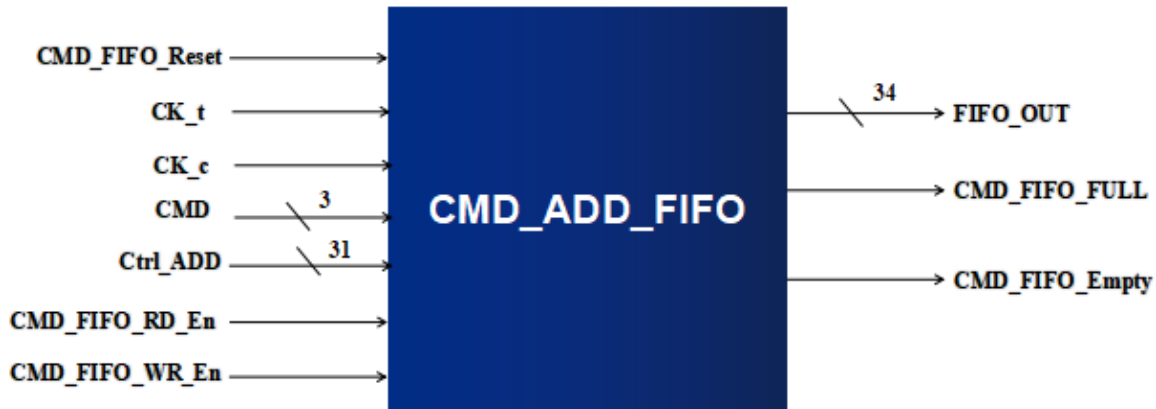


Figure 9: Block Diagram of Command_Address_FIFO.

2.5.2.2 List of Inputs & Outputs

Table 9: Inputs of Command_Address_FIFO.

Signal	Description
CMD_FIFO_Reset	Active high reset signal for FIFO, it's raised high when initialization sequence finishes.
CK_t,Ck_c	Differential clock of DDR5 SDRAM, all address and control input signals are sampled on the crossing of the positive edge of CK_t and negative edge of CK_c.
CMD[2:0]	Decoded Command comes from Command Decoder.
Ctrl_ADD[30:0]	Address that data will be read from memory or written in memory.
CMD_FIFO_RD_En	Read enable signal of FIFO comes from Command Decoder.
CMD_FIFO_WR_En	write enable signal of FIFO comes from Command Decoder.

Table 10: Outputs of Command_Address_FIFO

Signal	Description
FIFO OUT[33:0]	Output of FIFO that contains information stored in FIFO (CMD and Ctrl_ADD concatenated)
CMD_FIFO_FULL	Flag when 1 defines that FIFO is full.
CMD_FIFO_Empty	Flag when 1 defines that FIFO is empty.

2.5.2.3. Operation

Command decoder receives from CPU command every clock cycle but to execute the command by memory it needs more than one clock cycle as speed of CPU differs from speed of memory. Memory also does internal operations and features (i.e., self-refresh, Initialization, etc.), so memory can't always handle processor commands only. So, to avoid ignoring commands or drop of commands from CPU, we will store these commands and their related addresses from CPU in

DDR5 SDRAM Memory Controller Design and Verification

storage then after we finish command, we will take the next from storage to execute it and so on. We followed a simple policy First Come First Serve (FIFO) in scheduling and selecting which command will be executed. There are other polices that can achieve high performance than this policy but they will be more complex in implementation.

❖ Assumption & Notes

FIFO_OUT [33:31] represents CMD and FIFO_OUT [30:0] represents Ctrl_ADD.

2.5.3 Write_Data_FIFO

2.5.3.1 Block Diagram



Figure 10: Block Diagram of Write_Data_FIFO

2.5.3.2 List of Inputs & Outputs

Table 11: Inputs of Write_Data_FIFO

Signal	Description
CK_t,CK_c	Differential clock of DDR5 SDRAM,all address and control input signals are sampled on the crossing of the positive edge of CK_t and negative edge of CK_c
Ctrl_DataIn[255:0]	Data comes from CPU that will be written in memory during writing operation
WR_Data_RD_En	Read enable signal of FIFO comes from Command Decoder
WR_Data_WR_En	Write enable signal of FIFO comes from Command Decoder

DDR5 SDRAM Memory Controller Design and Verification

Table 12: Outputs of Write_Data_FIFO

Signal	Description
FIFO_WR_Data[255:0]	Output of FIFO that contains data stored in FIFO to store it in memory

2.5.3.3 Operation

As we discussed before why we need FIFO and answered this question that we need it to store information that comes from CPU to use it later, in Command_Address_FIFO we store command and address. It's still data need to be stored when write request comes then we made Write_Data_FIFO and here comes question why we used two FIFOs instead of only one and we can answer this question that we need to read data from controller to store it in memory in time differs from time of reading command and address so we will need two read enables so we made better a separate FIFO for data

2.5.4 Calculation of Depth of FIFOs

FIFO depth calculation is a critical phase in the design which needs to consider the worst case in all aspects and to do some assumptions [6]:

❖ Assumptions

1. Burst length (No. of data items to be transferred) = 120
2. Write frequency (Processor frequency) = 3.2 GHz
3. Reading frequency (Memory frequency) = 3.2 GHz
4. Command execution in 30 cycles (worst case)

❖ Calculations:

- Time required to write one data = $\frac{1}{3.2GHz} = 0.3125ns$
- Time required to read one data (execute command) = $30 * \frac{1}{3.2GHz} = 9.375ns$
- Time required to write all the data in the burst = $120 * 0.3125 = 37.5ns$
- Number of data items can be read in a duration of 37.5ns = $\frac{37.5ns}{9.375ns} = 4$
- The remaining no. of information to be stored in the FIFO = $120 - 4 = 116$

Then, the **minimum depth of the FIFO in our case should be equal 116**. In our code for simplicity we used depth=16 but in practical we should use depth as we mentioned.

DDR5 SDRAM Memory Controller Design and Verification

2.5.5 Counters

2.5.5.1 Block Diagram

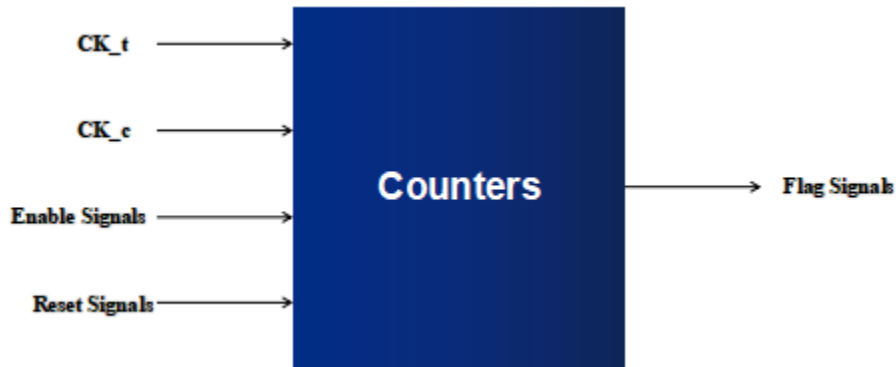


Figure 11: Block Diagram of Counters.

2.5.5.2 Operation

There are timing parameters in our design, then we used this block that contains some counters to count these parameters. Each Counter has Enable to enable counting when we enter certain state and want to count some parameters, after we reached value of timing parameter that we count it, counter outputs a flag indicates that counting has finished and after finishing counting, reset input to counter is raised high to reset counter.

2.5.5.3 List of Timing Parameters

Table 13: Initialization Timing Parameters [5]

Timing Parameter	Symbol	Value
Minimum reset_n low time after completion of voltage Ramp	tINIT1	200 μ s
Minimum cs_n low time before reset_n high	tINIT2	10 ns
Minimum cs_n low time after reset_n high	tINIT3	4 ms
Minimum time for dram to register exit on cs_n with cmos	tINIT4	2 μ s
Minimum cycles required after cs_n high	tINIT5	3 nCK
Minimum time from exit reset to first valid configuration command	tXPR	2 μ s
Minimum delay from MRR or MRW command to any other valid command	tMRD	max (14ns, 16nCK)
ZQ Calibration Time	tZQCAL	1 μ s
ZQ Calibration Latch Time	tZQLAT	max(30ns,8nCK)

DDR5 SDRAM Memory Controller Design and Verification

Table 14: Activate Timing Parameters [5].

Timing Parameter	Symbol	Value
ACT to Precharge command delay	tRAS	32 ns
ACT to ACT Command delay to same bank group	tRRD_L	Max(8nCK, 5ns)
ACT to ACT Command delay to different bank group	tRRD_S	8nCK

Table 15: Precharge Timing Parameters [5].

Timing Parameter	Symbol	Value
Precharge delay	tRP	13.750 ns

Table 16: Reading Timing Parameters [5].

Timing Parameter	Symbol	Value
Minimum Read to Read command delay for same bank group	tCCD_L_slr	Max(8nCK, 5ns)
Minimum Read to Read command delay for different bank group	tCCD_S_slr	8 nCK
Minimum Read to Write command delay for same bank group	tCCD_L_RTW_slr	20 ns
Minimum Read to Write command delay for different bank group	tCCD_S_RTW_slr	15 ns
Internal Read command to Precharge command delay	tRTP	Max(12nCK, 7.5ns)

Table 17: Writing Timing Parameters [5].

Timing Parameter	Symbol	Value
Minimum Write to Write command delay for same bank group	tCCD_L_WR_slr	Max(32nCK, 20ns)
Minimum Write to Write command delay for different bank group	tCCD_S_WR_slr	8 nCK

DDR5 SDRAM Memory Controller Design and Verification

Minimum Write to Read command delay for same bank group	tCCD_L_WTR_slr	20 ns
Minimum Write to Read command delay for different bank group	tCCD_S_WTR_slr	15 ns

Table 18: Self-Refresh Timing Parameters [5].

Timing Parameter	Symbol	Value
Command pass disable delay	tCPDED	5 ns
Self-Refresh CS_n low Pulse width	tCSL	10 ns
Self-Refresh exit CS_n high	tCASRX	0
Self-Refresh exit CS_n High Pulse width	tCSH_SRExit	13 ns
Self-Refresh exit CS_n Low Pulse width	tCSL_SRExit	3 nCK
Exit Self-Refresh to next valid command NOT requiring a DLL	tXS	2 μ s
Time interval between two selfrefresh operations	tREFI	3.9 μ s

❖ Assumption & Notes

1. These values of timing parameters are listed in JEDEC79-5 standard in section 3.3.1 Table 11, Table 329, Table 20, Table 467, Table 520, Table 481 and Table 525.
2. Our counters count in unit of clock cycles, so we needed to convert these values of timing parameters to form of multiple of clock cycles by dividing value of timing parameter /clock period
3. Enable and reset signals comes from FSMs blocks.
4. Outputs of these counters will be mainly inputs to FSMs blocks.
5. As we discussed before that each counter for timing parameter has reset, enable and flag that indicated counting has finished. Naming that we use here for reset signal of timing parameter counter is named by name of timing parameter and “Reset” word attached to it(for

DDR5 SDRAM Memory Controller Design and Verification

ex: tINIT1_Reset) ,the same thing for enable signal but “En” word is attached to it(for ex:tINIT1_En) and for flag has name as the name of timing parameter(for ex:tINIT1).

2.5.6 Command_FSM

2.5.6.1 Block_Diagram

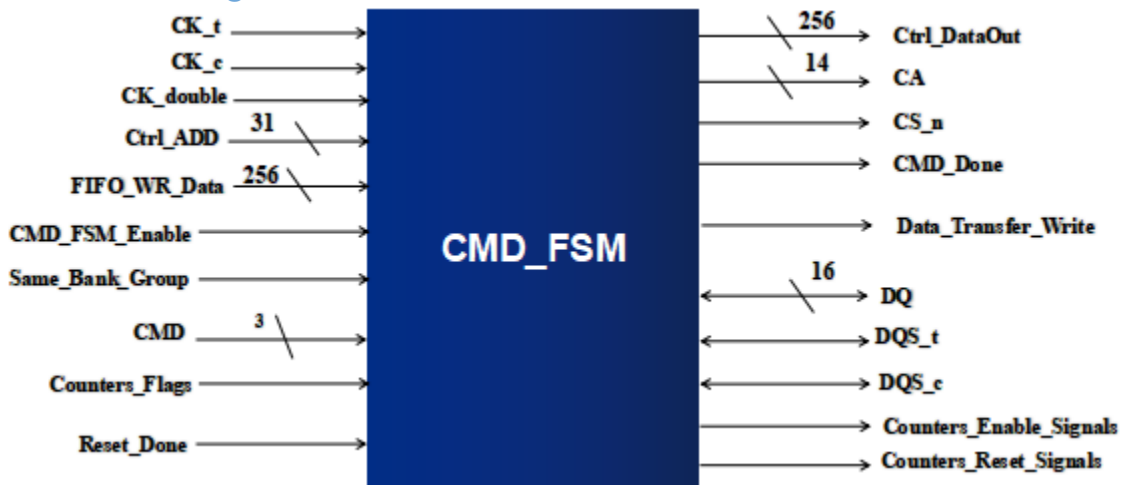


Figure 12: Block Diagram of Command_FSM

2.5.6.2 List of Inputs and Outputs

Table 19: Inputs of Command_FSM

Signal	Description
CK_t, CK_c	Differential clock of DDR5 SDRAM, all address and control input signals are sampled on the crossing of the positive edge of CK_t and negative edge of CK_c.
CK_double	Clock that has double frequency of SDRAM clock, DQS signals will be generated with positive edge of this clock, also data from memory will be sampled also on the positive edge of this clock.
Ctrl_ADD[31:0]	Address that data will be read from memory or written in memory.
FIFO_WR_Data[15:0]	Output of write_data_FIFO.
CMD_FSM_Enable	Enable signal of Command_FSM comes from Command Decoder.
Same_Bank_Group	Flag signal comes from Command Decoder.
CMD[2:0]	Decoded commend comes from Command Decoder.
Counter_Flags	Activate parameters, Percharge parameters, Read Parameters-Write parameters.

Table 20: Outputs of Command_FSM

Signal	Description
Ctrl_DataOut[255:0]	Data will be delivered from memory to CPU
CA[13:0]	Command/Address Inputs: CA signals provide the command and address inputs according to the Command Truth Table in standard section 4.1 Table 241

DDR5 SDRAM Memory Controller Design and Verification

CS_n	Chip Select: all commands are masked when cs_n is registered high. for one cycle commands (cs_n=0), for two cycle commands (cs_n=0 for first cycle,cs_n=1 for second cycle).
CMD_Done	When 1 defines that command has been executed
Data_Transfer_Write	When 1 defines that we need to set WR_Data_RD_En high to write data from WR_Data_FIFO to memory.
DQ[15:0]	Data Input/Output: Bi-directional data bus.
DQS_t,DQS_c	Data Strobe: output with read data, input with write data. Edge-aligned with read data, centered in write data. DDR5 SDRAM supports differential data strobe only and does not support single-ended.
Counter_Enable_Signals, Counter_Reset_Signals	Activate parameters -Percharge Parameters-Read Parameters-Write parameters.

2.5.6.3 Operation

This block will be responsible for implementing finite state machine that controls normal operation of memory (reading and writing), the simplified state diagram that describes reading and writing operation is shown in JEDEC79-5 standard in section 3.1 and Figure 13 shows this simplified state diagram.

DDR5 SDRAM Memory Controller Design and Verification

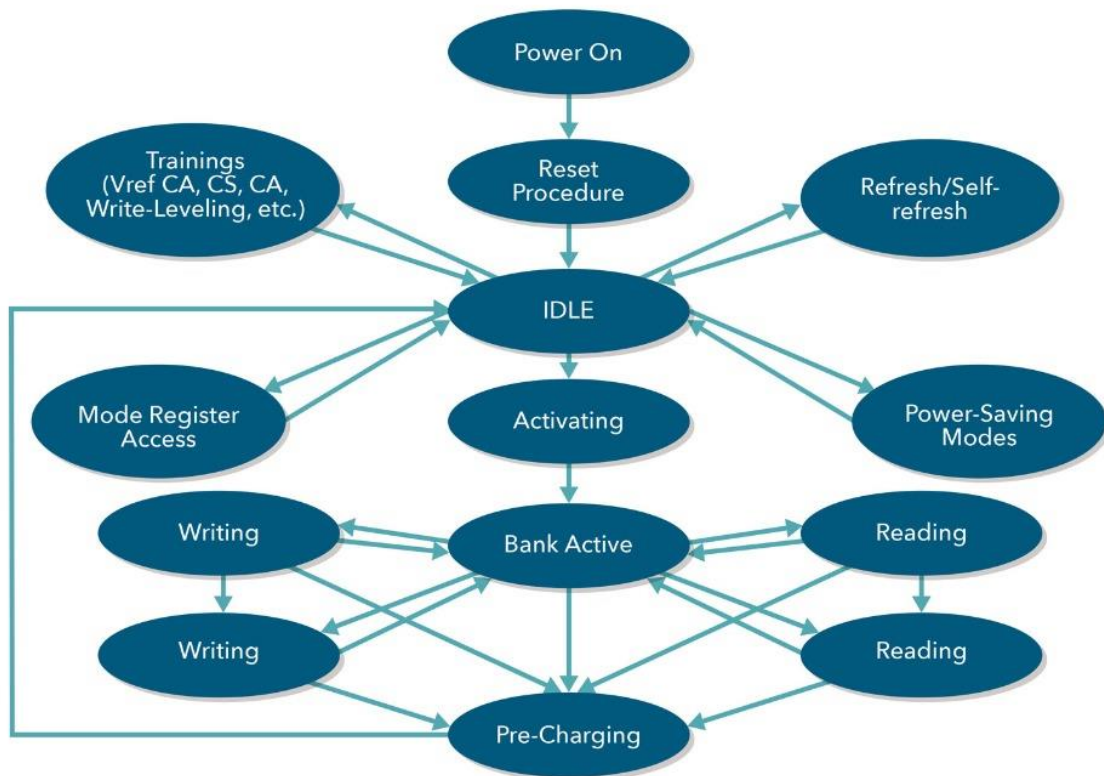


Figure 13: Simplified State Diagram, JEDEC Reference [5].

With help of this simplified state diagram and timing diagrams in JEDEC79-5 standard section 4.7 and section 4.8 that describes reading and writing operation and the pattern of preamble and postamble of these operations and these timing diagrams are shown in Figure 14 and Figure 15. So, we could figure out the detailed state diagram for the finite state machine that controls normal operation as will be illustrated later.

DDR5 SDRAM Memory Controller Design and Verification

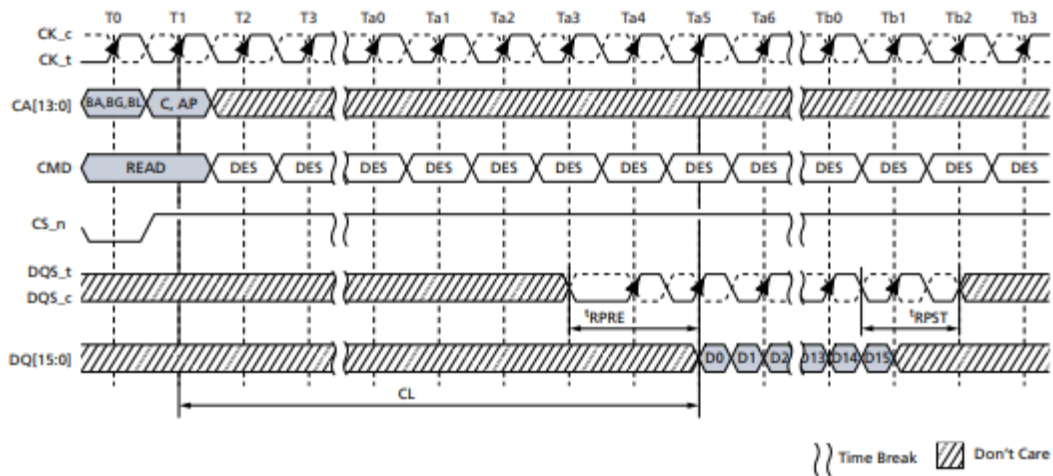


Figure 14: Timing Diagram for Read Burst Operation (BL16), JEDEC Reference [5].

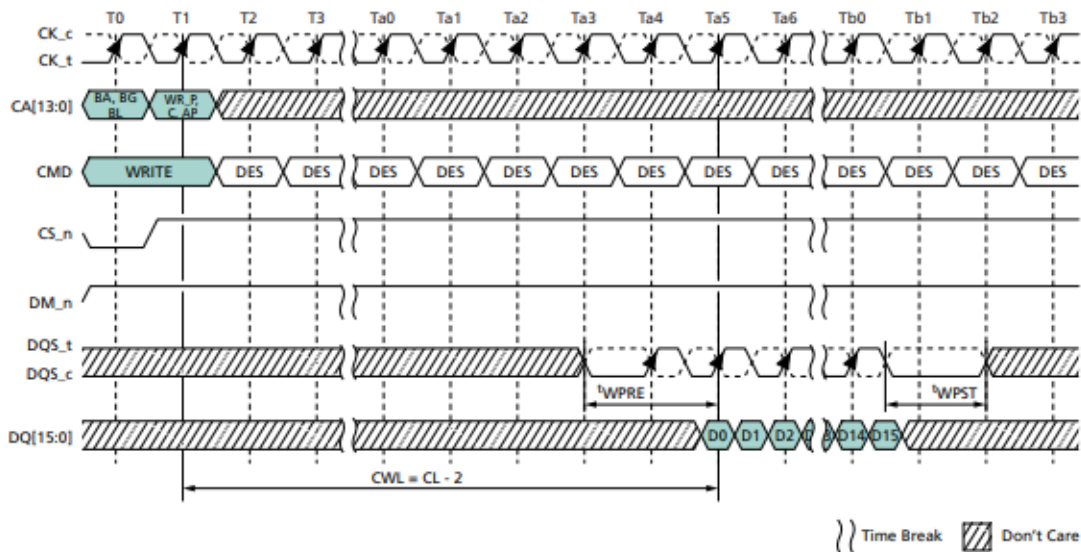


Figure 15: Timing Diagram for Write Burst Operation (BL16), JEDEC Reference [5].

Before we talk about this state diagram, there is an important thing we should handle it in this FSM and it is back-to-back operations (i.e., read after read, read after write, write after write, write after read). In back-to-back operations we have three scenarios, the description of each scenario is shown in Table 21:

Table 21: Scenarios of Back-to-Back Operations [5].

Scenario	Description
Bank is not activated	It means that operation is done on row in bank has not been activated before so in this scenario we will need to activate this row in this bank

DDR5 SDRAM Memory Controller Design and Verification

Bank is activated and same row	It means that operation is done on the same row in bank as the previous operation so we will proceed to execute operation and no need for precharge operation
Bank is activated and different row	It means that operation is done on the same bank as the previous operation but on different row so here we will need to execute precharge operation for row of previous operation then activate row of the current operation then proceed to execute the operation on the activated row

To handle back to back operations ,we used array as storage to store activated rows in banks, the length of the array is 16 as we have 4 bank groups each has 4 banks so we have 16 banks ,the entry for this array will bank group address and group address concatenated(4 bits) and the content of each location in array will be row address and status bit for this row to indicate row is activated or not (if 1 active and if 0 not active) so width of each location will be 19 bit (18 bits for row address and 1 bit for status bit).the shape of array will be as shown in Table 22:

Table 22:Storage of Activate Rows

Entry(Bank Group,Bank)(4 bits)	Row Address(18 bits)	Status bit
0000	XXXXXXXXXXXXXXXXXXXX	1
0001	XXXXXXXXXXXXXXXXXXXX	0
0010	XXXXXXXXXXXXXXXXXXXX	1
0011	XXXXXXXXXXXXXXXXXXXX	0
0100	XXXXXXXXXXXXXXXXXXXX	1
0101	XXXXXXXXXXXXXXXXXXXX	0
0110	XXXXXXXXXXXXXXXXXXXX	1
0111	XXXXXXXXXXXXXXXXXXXX	0
1000	XXXXXXXXXXXXXXXXXXXX	1
.	XXXXXXXXXXXXXXXXXXXX	1
.		
.		

When row is activated, it is stored in array and its status bit is set 1 and when the row is precharged its status bit is set 0.

Here we can show detailed state diagram for normal operations as shown in Figure 12:

DDR5 SDRAM Memory Controller Design and Verification

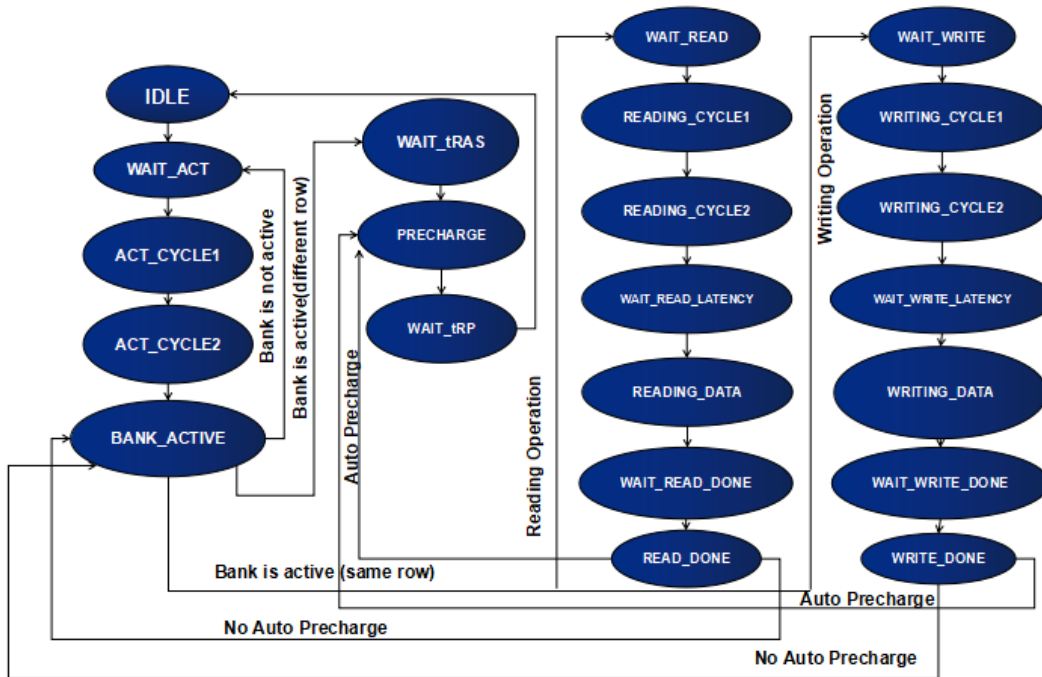


Figure 16: Command_FSM

The description of each state is illustrated in Table 23:

Table 23: Description of each state in Command_FSM

State	Description
IDLE	In this state, CMD_FSM_Enable signal is checked if it's asserted high or not, if it's asserted high normal operation is started by going to WAIT_ACT state
WAIT_ACT	In this state, timing constraints are checked between consecutive active commands and the timing constraints that are checked will differ according the two consecutive operations are in same bank group or not and we can detect this through same_bank_group signal. If they are in same bank group tRRD_L is checked and if they are not tRRD_S is checked. If timing constraints are satisfied, normal operation will be continued by going to ACT_CYCLE1 state.
ACT_CYCLE1	As active command from commands that are executed in two cycles, each cycle different information is sent to memory so we send information of active command in two states ACT_CYCLE1 and ACT_CYCLE2 so this state is followed with ACT_CYCLE2 state without checking any conditions.
ACT_CYCLE2	In this state, information of active command in the second cycle is sent to memory, then it's followed by BANK_ACTIVE state without checking any conditions

DDR5 SDRAM Memory Controller Design and Verification

<p>BANK_ACTIVE</p>	<p>In this state, three different scenarios of back to back operations are checked, checking if bank is not active so go to WAIT_ACT state to activate it or if bank is active then checking if the row of current operation is the same as previous or not. If they are the same, checking if the current operation is read or write if read go to WAIT_READ state and if write go to WAIT_WRITE state. If they are different, go to WAIT_tRAS state to precharge the previous row then activates the current row.</p>
<p>WAIT_tRAS</p>	<p>Before applying precharge operation, timing between active command and precharge command should be checked so in this state tRAS is checked and if it's satisfied, go to PRECHARGE state.</p>
<p>PRECHARGE</p>	<p>In this state, information of precharge command is sent to memory and it's one cycle command so it needs only one state then go to WAIT_tRP state without checking any conditions.</p>
<p>WAIT_tRP</p>	<p>Precharge operation needs time to be executed by memory so in this state this time(tRP) is checked before executing other operations and if this time is satisfied, go to IDLE state.</p>
<p>WAIT_READ</p>	<p>In this state, timing constraints of reading command are checked. There are two timing constraints depending on if the previous command was read or write. So, in this state first checking if previous command was read or write. If it's read timing related to read-to-read delay is checked and also here timing parameter that is checked will differ if they are in same bank group or not, if they are same tCCD_L_slr is checked and if they are not tCCD_S_slr is checked. If it's write timing related to write to read delay is checked and also here timing parameter that is checked will differ if they are in same bank group or not, if they are same tCCD_L_WTR_slr is checked and if they are not tCCD_S_WTR_slr is checked. If these timing constraints are satisfied, go to READING_CYCLE1 state.</p>
<p>READING_CYCLE1</p>	<p>As read command from commands that are executed in two cycles, each cycle different information is sent to memory so information of read command are sent in two states READING_CYCLE1 and READING_CYCLE2 so this state is followed with READING_CYCLE2 state without checking any conditions.</p>
<p>READING_CYCLE2</p>	<p>In this state, information of read command in the second cycle is sent to memory, then go to WAIT_READ_LATENCY state without checking any conditions.</p>
<p>WAIT_READ_LATENCY</p>	<p>In this state, Read Latency (RL-2 clock cycles for preamble) is checked as shown in Figure 10 and if it's satisfied, go to READING_DATA state.</p>

DDR5 SDRAM Memory Controller Design and Verification

<p>READING_DATA</p>	<p>In this state, reading operation are actually executed and this as shown in Figure 10 ,this done by receiving preamble from memory then read data with burst length that CPU has chosen it ,then after receiving data postamble is sent by memory to controller, after receiving postamble we can say that reading operation is done .This state in the beginning is implemented in un synthesizable manner but as we will discuss in detail in this chapter section 6 that we implemented it in synthesizable manner by calling Read FSM that will perform reading operation through states and after finishing it sends signal called Read_Done, this signal is checked in WAIT_READ_DONE state so this state is followed by WAIT_READ_Done state.</p>
<p>WAIT_READ_DONE</p>	<p>In this state, Read_Done signal is checked. If it's asserted high, go to READ_DONE state.</p>
<p>READ_DONE</p>	<p>In this state, it's checked if the command was with autoprecharge or not. If it's with auto precharge ,go to WAIT_tRAS state to apply precharge operation and if it's not go to BANK_ACTIVE state waiting another operation.</p>
<p>WAIT_WRITE</p>	<p>In this state, timing constraints of writing command are checked. There are two timing constraints depending on if the previous command was read or write. So, in this state first checking if pervious command was read or write. If it's write timing related to write to write delay is checked and also here timing parameter that is checked will differ if they are in same bank group or not, if they are same tCCD_L_WR_slr is checked and if they are not tCCD_S_WR_slr is checked. If it's read timing related to read to write delay is checked and also here timing parameter that is checked will differ if they are in same bank group or not, if they are same tCCD_L_RTW_slr is checked and if they are not tCCD_S_RTW_slr is checked.If these timing constraints are satisfied, go to WRITING_CYCLE1 state.</p>
<p>WRITING_CYCLE1</p>	<p>As write command from commands that are executed in two cycles, each cycle different information is sent to memory so information of write command is sent in two states WRITING_CYCLE1 and WRITING_CYCLE2 this state is followed by WRITING_CYCLE2 state without checking any conditions.</p>
<p>WRITING_CYCLE2</p>	<p>In this state, information of write command in the second cycle is sent to memory, then go to WAIT_WRITE_LATENCY state without checking any conditions.</p>
<p>WAIT_WRITE_LATENCY</p>	<p>In this state, Write Latency (WL-2 clock cycles for preamble) is checked as shown in Figure 11 and if it's satisfied, go to WRITING_DATA state</p>

DDR5 SDRAM Memory Controller Design and Verification

WRITING_DATA	In this state, writing operation are actually executed and this as shown in Figure 11 ,this done by sending preamble to memory then write data with burst length that CPU has chosen it ,then after writing data postamble is sent by controller to memory, after sending postamble we can say that writing operation is done .This state in the beginning is implemented in un synthesizable manner but as we will discuss in detail in this chapter 2 section 6 that we implemented it in synthesizable manner by calling Write FSM that will perform writing operation through states and after finishing it sends signal called Write_Done, this signal is checked in WAIT_WRITE_DONE state this state is followed by WAIT_WRITE_Done state.
WAIT_WRITE_DONE	In this state, write_Done signal is checked. If it's asserted high, go to Write_DONE state.
WRITE_DONE	In this state, it's checked if the command was with autoprecharge or not. If it's with auto precharge, go to WAIT_tRAS state to apply precharge operation and if it's not go to BANK_ACTIVE state waiting another operation.

The outputs of each state are illustrated in Table 24:

Table 24: Outputs of each state in Command_FSM

State	Outputs
IDLE	<ul style="list-style-type: none"> ➤ tRP_Reset = 1 ➤ tRP_En = 1
WAIT_ACT	No Outputs
ACT_CYCLE1	<ul style="list-style-type: none"> ➤ CS_n = 0 ➤ CA = ACT_Cycle1 <p>If they are same bank group</p> <ul style="list-style-type: none"> ➤ tRRD_L_Reset = 1 ➤ tRRD_L_En = 0 <p>If they are not same bank group</p> <ul style="list-style-type: none"> ➤ tRRD_S_Reset = 1 ➤ tRRD_S_En = 0

DDR5 SDRAM Memory Controller Design and Verification

<p style="text-align: center;">ACT_CYCLE2</p>	<ul style="list-style-type: none"> ➤ Store this row in array of activated rows and set its status bit high. ➤ CS_n = 1 ➤ CA = ACT_Cycle2 ➤ tRAS_En = 1 <p>If they are same bank group</p> <ul style="list-style-type: none"> ➤ tRRD_L_En = 1 <p>If they are not same bank group</p> <ul style="list-style-type: none"> ➤ tRRD_S_En = 1
<p style="text-align: center;">BANK_ACTIVE</p>	<p>No Outputs</p>
<p style="text-align: center;">WAIT_tRAS</p>	<p>No Outputs</p>
<p style="text-align: center;">PRECHARGE</p>	<ul style="list-style-type: none"> ➤ Remove this row from array of activated rows by setting its status bit low. ➤ CS_n=0 ➤ CA=PREpb ➤ tRAS_Reset = 1 ➤ tRAS_En = 0 ➤ tRP_En = 1
<p style="text-align: center;">WAIT_tRP</p>	<p>No Outputs</p>
<p style="text-align: center;">WAIT_READ</p>	<p>No Outputs</p>
<p style="text-align: center;">READING_CYCLE1</p>	<ul style="list-style-type: none"> ➤ CS_n=0 ➤ CA=RD_Cycle1 ➤ tCCD_L_WTR_slr_Reset= 1 ➤ tCCD_S_WTR_slr_Reset = 1 ➤ tCCD_L_slr_Reset = 1 ➤ tCCD_S_slr_Reset = 1 ➤ tCCD_L_WTR_slr_En = 0

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ tCCD_S_WTR_slr_En = 0 ➤ tCCD_L_slr_En = 0 ➤ tCCD_S_slr_En = 0
READING_CYCLE2	<ul style="list-style-type: none"> ➤ CS_n=1 ➤ CA=RD_Cycle2 ➤ RL_En = 1 <p>If they are same bank group</p> <ul style="list-style-type: none"> ➤ tCCD_L_slr_En = 1 ➤ tCCD_L_RTW_slr_En = 1 <p>If they are not same bank group</p> <ul style="list-style-type: none"> ➤ tCCD_S_slr_En = 1 ➤ tCCD_S_RTW_slr_En = 1
WAIT_READ_LATENCY	No Outputs
READING_DATA	<ul style="list-style-type: none"> ➤ Set Pervious command as read ➤ RL_Reset = 1 ➤ RL_En = 0 ➤ Enable Read FSM
WAIT_READ_DONE	No Outputs
READ_DONE	<ul style="list-style-type: none"> ➤ CMD_Done=1 ➤ Disable Read FSM
WAIT_WRITE	No Outputs
WRITING_CYCLE1	<ul style="list-style-type: none"> ➤ CS_n=0 ➤ CA=WR_Cycle1 ➤ tCCD_L_RTW_slr_Reset = 1 ➤ tCCD_S_RTW_slr_Reset = 1 ➤ tCCD_L_WR_slr_Reset = 1

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ tCCD_S_WR_slr_Reset = 1 ➤ tCCD_L_RTW_slr_En = 0 ➤ tCCD_S_RTW_slr_En = 0 ➤ tCCD_L_WR_slr_En = 0 ➤ tCCD_S_WR_slr_En = 0
WRITING_CYCLE2	<ul style="list-style-type: none"> ➤ CS_n=1 ➤ CA=WR_Cycle2 ➤ WL_En = 1 <p>If they are same bank group</p> <ul style="list-style-type: none"> ➤ tCCD_L_WR_slr_En = 1 ➤ tCCD_L_WTR_slr_En = 1 <p>If they are not same bank group</p> <ul style="list-style-type: none"> ➤ tCCD_S_WR_slr_En = 1 ➤ tCCD_S_WTR_slr_En = 1
WAIT_WRITE_LATENCY	No Outputs
WRITING_DATA	<ul style="list-style-type: none"> ➤ Set Pervious command as write ➤ WL_Reset = 1 ➤ WL_En = 0 ➤ Enable Write FSM ➤ Data_Transfer_Write = 1
WAIT_WRITE_DONE	<ul style="list-style-type: none"> ➤ Data_Transfer_Write = 0
WRITE_DONE	<ul style="list-style-type: none"> ➤ CMD_Done=1 ➤ Disable Write FSM

DDR5 SDRAM Memory Controller Design and Verification

❖ Assumption & Notes

1. We reset counters in the next state of finishing of counting.
2. In Table 24 of outputs, CA pins are written in form of basic commands that are in Command Truth Table in JEDEC79-5 Standard section 4.1 Table 241.
3. As we know that there are commands with auto precharge, the difference between it and without auto precharge in pattern of CA pins in second cycle of command and there is a bit called AP, we set it low in case of autoprecharge and high in case of without autoprecharge.
4. In any state, if timing constraint isn't satisfied, we will wait in this state until it's satisfied.
5. In Figure 10, $RL=CL$ and in Figure 11 $WL=CWL$.

2.5.7 Initialization_FSM

2.5.7.1 Block Diagram

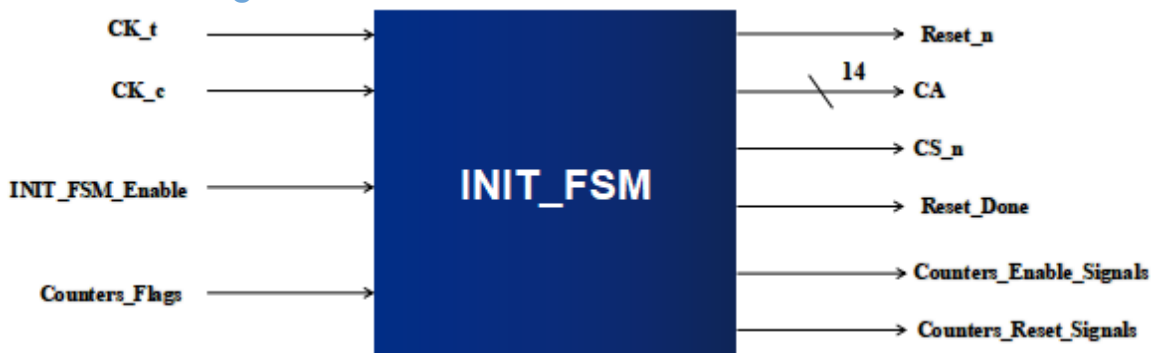


Figure 17: Block Diagram of Initialization_FSM

2.5.7.2 List of Inputs and Outputs

Table 25: Inputs of Initialization_FSM

Signal	Description
CK_t, CK_c	Differential clock of DDR5 SDRAM, all address and control input signals are sampled on the crossing of the positive edge of CK_t and negative edge of CK_c
INIT_FSM_Enable	Enable signal of Initialization_FSM comes from Command Decoder
Counter_Flags	Initialization parameters

DDR5 SDRAM Memory Controller Design and Verification

Table 26: Outputs of Initialization_FSM

Signal	Description
Reset_n	Active low asynchronous reset: reset is active when reset_n is low, and inactive when reset_n is high. reset_n must be high during normal operation.
CA[13:0]	Command/Address Inputs: CA signals provide the command and address inputs according to the Command Truth Table in JEDEC standard section 4.1 Table 241
CS_n	Chip Select: All commands are masked when CS_n is registered high. For one cycle commands (CS_n=0), for two cycle commands (CS_n=0 for first cycle, CS_n=1 for second cycle).
Reset_Done	When 1 defines that initialization sequence has been executed.
Counter_Enable_Signals , Counter_Reset_Signals	Initialization parameters.

2.5.7.3 Operation

This block will be responsible for implementing finite state machine that controls initialization sequence for memory, we construct this finite state machine shown in Figure 15 based on timing diagram that describes initialization sequence in JEDEC79-5 standard section 3.3.1 and it's shown in Figure 18

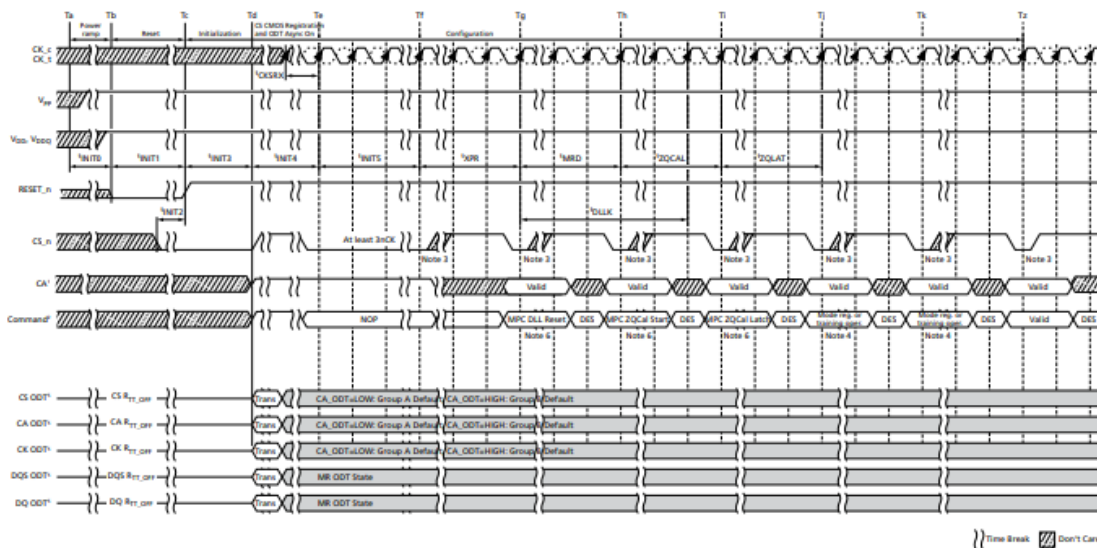


Figure 18: Reset and Initialization Sequence at Power-on Ramping, JEDEC Reference [5].

DDR5 SDRAM Memory Controller Design and Verification

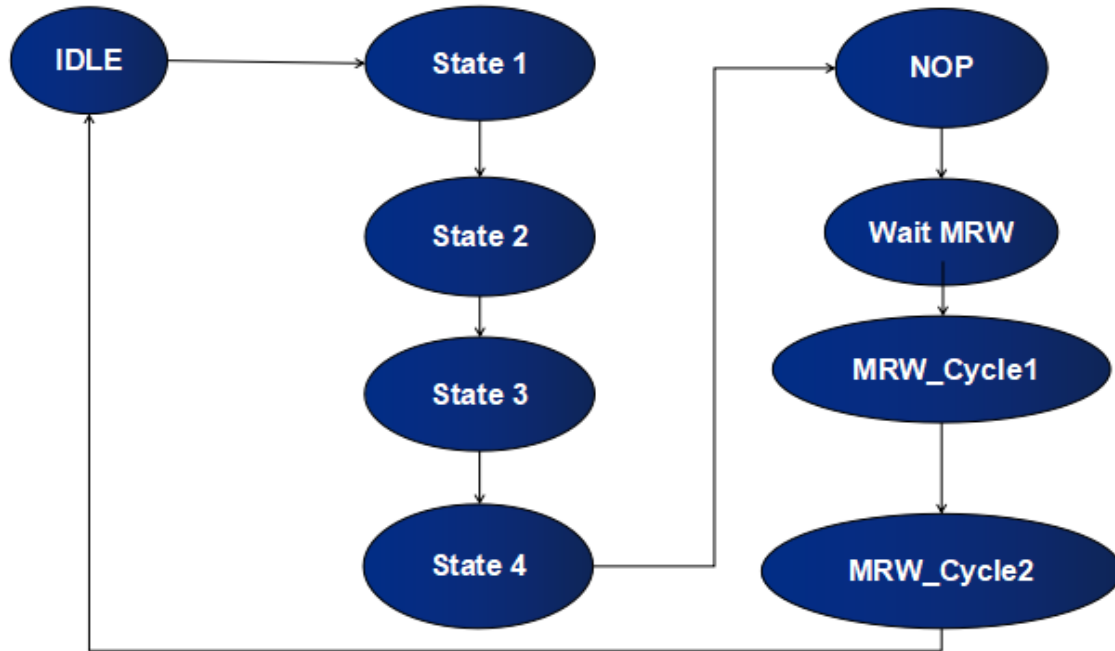


Figure 19: Initialization_FSM

Table 27: Description of each state in Initialization_FSM.

State	Description
IDLE	In this state, INIT_FSM_Enable is checked if it's asserted high or not, if it's asserted high, go to State 1 .
State 1	In this state, Some outputs like CS_n , CA and Reset_n takes some values for time interval tINIT1 as shown in Figure 14, then go to State 2 .
State 2	In this state, Some outputs like CS_n , CA and Reset_n takes some values for time interval tINIT1-tINIT2 as shown in Figure 14, then go to State 3 .
State 3	In this state, some outputs like CS_n, CA and Reset_n takes some values for time interval tINIT3 as shown in Figure 14, then go to State 4 .
State 4	In this state, some outputs like CS_n, CA and Reset_n takes some values for time interval tINIT4 as shown in Figure 14, then go to NOP state.
NOP	In this state, we wait for three clock cycles (NOP_Count) and also Some outputs like CS_n ,CA and Reset_n takes some values for this interval as shown in Figure 14 then go to Wait_MRW state.
Wait_MRW	In this state, some outputs like CS_n, CA and Reset_n takes some values for time interval tMRW (tXPR+tMRD+tZQCAL+tZQLAT) as shown in Figure 14, then go to MRW_Cycle1 state.
MRW_Cycle1	As Mode_Register_Write command from commands that are executed in two cycles, each cycle different information are sent to memory information of Mode_Register_Write command is sent in two states MRW_Cycle1 and MRW_Cycle2 this state is followed by MRW_Cycle2 state without checking any conditions.

DDR5 SDRAM Memory Controller Design and Verification

MRW_Cycle2	In this state, information of Mode_Register_Write command in the second cycle is sent to memory, then go to IDLE state without checking any conditions waiting another initialization request. In this state self-refresh timer is also enabled
-------------------	--

Table 28: Outputs of each state in Initialization_FSM

State	Outputs
IDLE	<ul style="list-style-type: none"> ➤ Reset_Done=0 ➤ tINIT1_Reset=0 ➤ tINIT2_Reset=0 ➤ tINIT3_Reset=0 ➤ tINIT4_Reset=0 ➤ tMRW_Reset = 0 ➤ NOP_Count_Reset=0
State 1	<ul style="list-style-type: none"> ➤ tINIT1_En=1 ➤ Reset_n=0 ➤ CS_n=don't care ➤ CA=don't care
State 2	<ul style="list-style-type: none"> ➤ tINIT2_En=1 ➤ tINIT1_Reset=1 ➤ tINIT1_En=0 ➤ Reset_n=0 ➤ CS_n=0 ➤ CA=don't care
State 3	<ul style="list-style-type: none"> ➤ tINIT3_En=1 ➤ tINIT2_Reset=1

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ tINIT2_En=0 ➤ Reset_n=1 ➤ CS_n=0 ➤ CA=don't care
State 4	<ul style="list-style-type: none"> ➤ tINIT3_Reset=1 ➤ tINIT4_En=1 ➤ tINIT3_En=0 ➤ Reset_n=1 ➤ CS_n=1 ➤ CA=All Ones
NOP	<ul style="list-style-type: none"> ➤ tINIT4_Reset=1 ➤ tINIT4_En=0 ➤ NOP_Count_En=1 ➤ CS_n=0 ➤ CA=NOP
Wait_MRW	<ul style="list-style-type: none"> ➤ NOP_Count_Reset=1 ➤ NOP_Count_En=0 ➤ tMRW_En=1
MRW_Cycle1	<ul style="list-style-type: none"> ➤ tMRW_Reset=1 ➤ tMRW_En=0 ➤ CS_n=0 ➤ CA=MRW_Cycle1

DDR5 SDRAM Memory Controller Design and Verification

MRW_Cycle2	<ul style="list-style-type: none"> ➤ CS_n=1 ➤ CA=MRW_Cycle2 ➤ Reset_Done=1,tREFI_En =1
-------------------	---

2.5.8 Self Refresh_FSM

2.5.8.1 Block Diagram

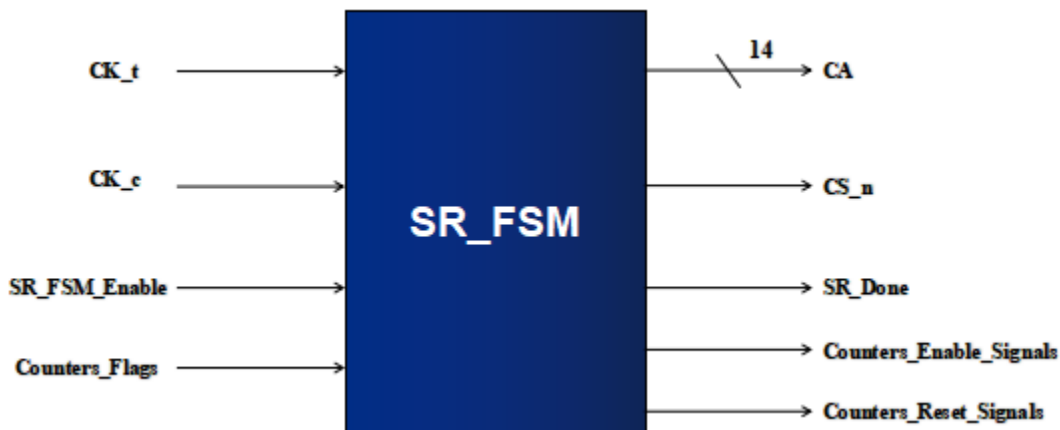


Figure 20: Block Diagram of Self-Refresh_FSM

2.5.8.2 List of Inputs and Outputs

Table 29: Inputs of Self Refresh_FSM

Signal	Description
CK_t, CK_c	Differential clock of DDR5 SDRAM, all address and control input signals are sampled on the crossing of the positive edge of CK_t and negative edge of CK_c
SR_FSM_Enable	Enable signal of Self Refresh_FSM comes from Command Decoder
Counter_Flags	Self-Refresh parameters

Table 30: Outputs of Self Refresh_FSM

Signal	Description
CS_n	Chip Select: All commands are masked when CS_n is registered high. For one cycle commands (CS_n=0),for

DDR5 SDRAM Memory Controller Design and Verification

	two cycle commands(CS_n=0 for first cycle,CS_n=1 for second cycle)
CA[13:0]	Command/Address Inputs: CA signals provide the command and address inputs according to the Command Truth Table in JEDEC standard section 4.1 Table 241
SR_Done	When 1 defines that self-refresh sequence has been executed
Counter_Enable_Signals, Counter_Reset_Signals	Self-Refresh parameters

2.5.8.3 Operation

This block will be responsible for implementing finite state machine that controls self-refresh sequence for memory, we construct this finite state machine shown in Figure 18 based on timing diagram that describes self-refresh sequence in JEDEC79-5 standard section 4.9 and it's shown in Figure 21

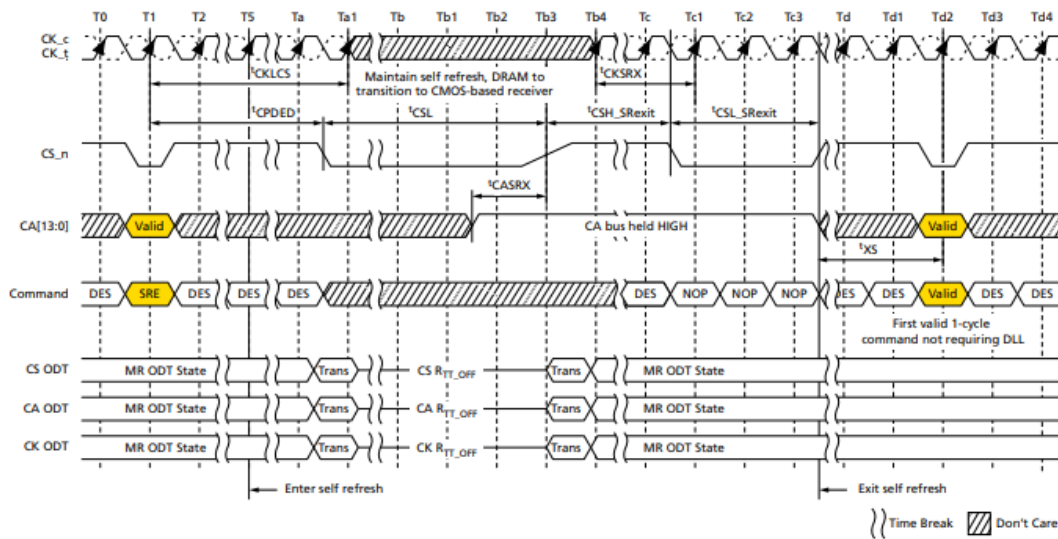


Figure 21: Self-Refresh Entry/Exit Timing with One-Cycle Exit Command, JEDEC Reference [5].

DDR5 SDRAM Memory Controller Design and Verification

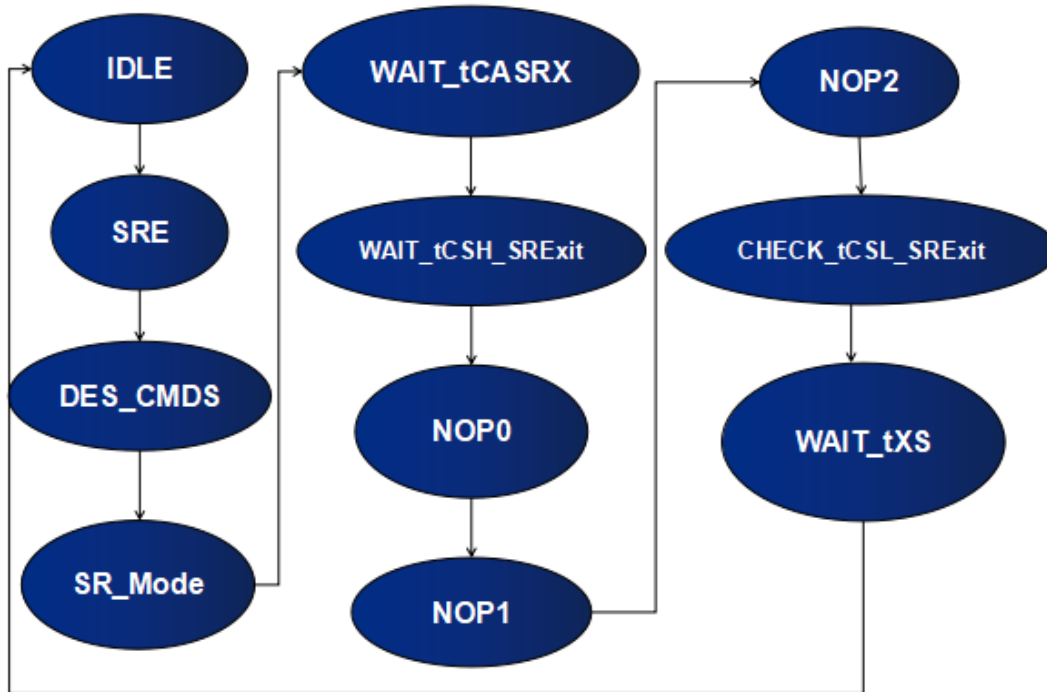


Figure 22: Self-Refresh_FSM

Table 31: Description of each state in Self Refresh_FSM

State	Description
IDLE	In this state, SR_FSM_Enable is checked if it is asserted high or not, if it's asserted high, go to SRE state.
SRE	It's self-refresh entry command state, in this state self-refresh entry command (SRE) is sent to memory then go to DES_CMDS state without checking any conditions.
DES_CMDS	In this state amount of number of Deselect Commands are sent to memory for time interval tCPDED as shown in Figure 17 then go to SR_Mode state.
SR_Mode	In this state, self-refresh operation is entered. Some outputs like CS_n and CA takes some values for time interval tCSL as shown in Figure 17. If tCSL is satisfied then go to WAIT_tCASRX state.
WAIT_tCASRX	In this state, some outputs like CS_n and CA takes some values for time interval tCASRX as shown in Figure 17. If tCASRX is satisfied then go to WAIT_tCSH_SRExit state
WAIT_tCSH_SRExit	In this state, Some outputs like CS_n and CA takes some values for time interval tCSH_SRExit as shown in Figure 17. If tCSH_SRExit is satisfied then go to NOP0 state

DDR5 SDRAM Memory Controller Design and Verification

NOP0	In this state, information of NOP command is sent to memory then it's followed by NOP1 state without checking any conditions.
NOP1	In this state, information of NOP command is sent to memory then it's followed by NOP2 state without checking any conditions.
NOP2	In this state, information of NOP command is sent to memory then it's followed by CHECK_tCSL_SRExit state without checking any conditions.
CHECK_tCSL_SRExit	In this state, self-refresh operation is excited by checking tCSL_SRExit as shown in Figure 17. If it's satisfied then go to WAIT_tXS state
WAIT_tXS	In this state, tCASRX is checked. If it is satisfied then go to IDLE state waiting timer of self-refresh expires to start another self-refresh operation.

Table 32: Outputs of each state in Self Refresh_FSM.

State	Outputs
IDLE	<ul style="list-style-type: none"> ➤ tXS_Reset = 1 ➤ tXS_En = 0 ➤ SR_Done = 0
SRE	<ul style="list-style-type: none"> ➤ tREFI_Reset = 1 ➤ tREFI_En = 0 ➤ CS_n = 0 ➤ CA = SRE ➤ tCPDED_En=1
DES_CMDS	<ul style="list-style-type: none"> ➤ CS_n = 1 ➤ CA = DES
SR_Mode	<ul style="list-style-type: none"> ➤ tCPDED_Reset=1 ➤ tCPDED_En = 0

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ tCSL_En=0
WAIT_tCASRX	<ul style="list-style-type: none"> ➤ tCSL_Reset=1 ➤ tCSL_En = 0 ➤ tCASRX_En=1 ➤ CA = All Ones
WAIT_tCSH_SRExit	<ul style="list-style-type: none"> ➤ tCASRX_Reset=1 ➤ tCASRX_En = 0 ➤ tCSH_SRExit_En=1 ➤ CS_n = 1
NOP0	<ul style="list-style-type: none"> ➤ tCSH_SRExit_Reset=1 ➤ tCSH_SRExit_En = 0; ➤ tCSL_SRExit_En=1 ➤ CS_n = 0 ➤ CA = NOP
NOP1	<ul style="list-style-type: none"> ➤ CS_n = 0 ➤ CA =NOP
NOP2	<ul style="list-style-type: none"> ➤ CS_n = 0 ➤ CA = NOP
CHECK_tCSL_SRExit	<ul style="list-style-type: none"> ➤ CS_n = 1
WAIT_tXS	<ul style="list-style-type: none"> ➤ tCSL_SRExit_Reset=1 ➤ tCSL_SRExit_En = 0 ➤ tXS_En = 1 <p>If tXS is satisfied</p> <ul style="list-style-type: none"> ➤ SR_Done = 1 , tREFI_En = 1

DDR5 SDRAM Memory Controller Design and Verification

❖ Assumption & Notes

As shown in Figure 17, that there is overlapping between tCSL and tCASRX. The higher priority for tCSL, it should be satisfied first then tCASRX is checked, if it's satisfied CS_n is raised high. to avoid making output depends on condition of tCASRX, we made two states SR_Mode to check that tCSL is satisfied then go to WAIT_tCASRX to check tCASRX if it's satisfied or not.

2.6 DESIGN ENHANCEMENT

There is a big difference between making the design work properly in its functionality only (the design is bit accurate and cycle accurate only) and making it fully synthesizable to model it with actual gates without any problems or violations.

There were many changes in the design to overcome synthesis problems and this is a brief of those changes:

- **Problem_1:** to generate CK_t, CK_c and CK_double, we used block called clock generator to generate them but the fact that this isn't a synthesizable block as shown in Figure 23:

- [Synth 8-6896] loop limit (65536) exceeded inside initial block, initial block items will be ignored [Clock_Generator.sv:17] (2 more like this)
- [Synth 8-6014] Unused sequential element Prev_Reset_reg was removed. [Command_Decoder.sv:86] (1 more like this)

Figure 23: Clock Generator isn't synthesizable

- ✓ **Solution_1:** remove Clock generator block and make these clocks as input and generate them by stimulus from environment.

- **Problem_2:** we were relying on the cross of the differential clock to sample any information but this couldn't be understood by synthesizer and causes problem of ambiguous clock triggering as shown in Figure 24:

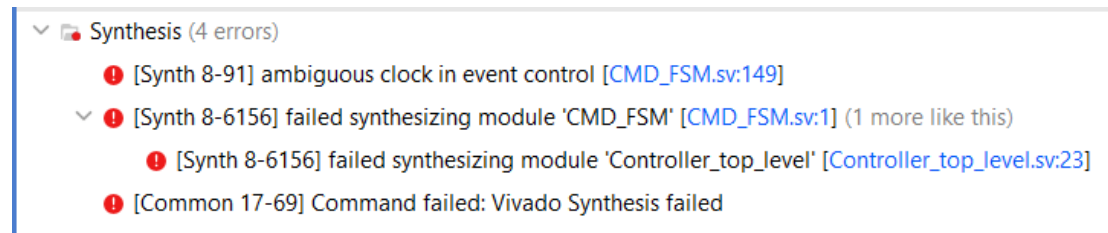


Figure 24: Problem of ambiguous clock triggering

- ✓ **Solution_2:** make it trigger with one clock only (not the cross of the differential clock), we got rid of one of the differential clock to get understood by the synthesizer and sample information with triggering edge of the other clock.

- **Problem_3:** we were doing READING_DATA and WRITING_DATA states by tasks which contain event blocking statements that is got used only on simulation and that weren't synthesizable.

DDR5 SDRAM Memory Controller Design and Verification

- ✓ **Solution_3:** exchange non-synthesizable tasks with synthesizable states in FSMs, FSM for reading operation called RD_FSM and FSM for writing operation called WR_FSM and made these FSMs as separate modules which are instantiated in Command_FSM block and are enabled in READING_DATA and WRITING_DATA states as we mentioned in Table 23. Let us discuss briefly these modules, their block diagrams as shown in Figure 21 and Figure 23 and state diagrams for these FSMs as shown in Figure 22 and Figure 24:

2.6.1 Read_FSM

2.6.1.1 Block Diagram

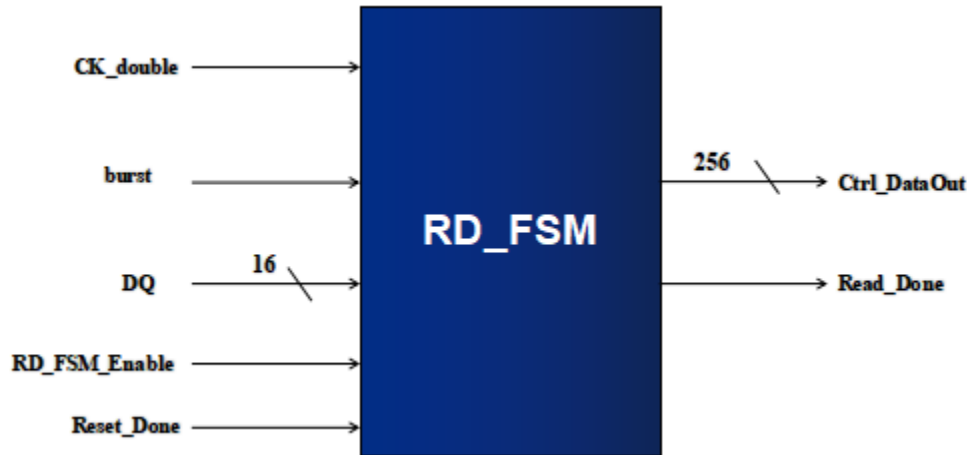


Figure 25: Block Diagram of Read_FSM

2.6.1.2 List of Inputs and Outputs

Table 33: Inputs of Read_FSM

Signal	Description
CK_double	Clock that has double frequency of SDRAM clock ,DQS signals will be generated with positive edge of this clock,also data from memory will be sampled also on the positive edge of this clock
burst	Flag signal when 1 defines burst length is 8 and when 0 defines burst length 16
DQ[15:0]	Data Input/Output: Bi-directional data bus
RD_FSM_Enable	Enable signal of Read FSM
Reset_Done	When 1 defines that initialization sequence has been executed

Table 34: Outputs of Read_FSM

Signal	Description
Ctrl_DataOut[255:0]	Data will be delivered from memory to CPU
Read_Done	Flag signal when 1 defines that reading operation has done

DDR5 SDRAM Memory Controller Design and Verification

2.6.1.3 Operation

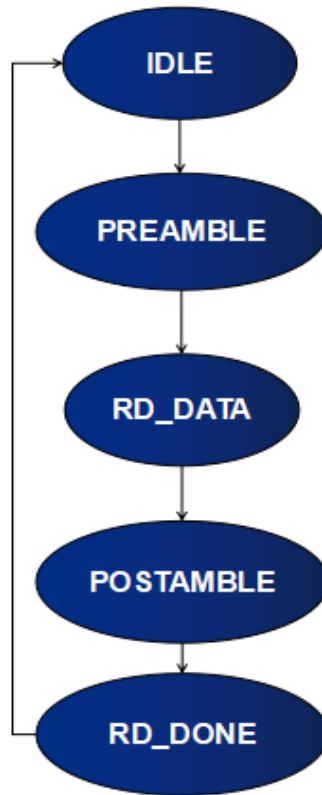


Figure 26: Read_FSM

Table 35: Description of each state in Read_FSM

State	Description
IDLE	In this state, RD_FSM_Enable is checked, if it's asserted high then go to PREAMBLE state.
PREAMBLE	We declared counter in RD_FSM block that counts two cycles from CK_double that represents interval of preamble and when this counter finishes, it outputs signal called PRE_Ctr_Done. After this signal is raised high this means that preamble phase has finished so we can read data by going to RD_DATA state.
RD_DATA	In this state reading data is done by transfer data from memory to controller with burst length that defined by burst signal, and there is internal signal when transfer is done it's raised high called Data_Ctr_Done then go to POSTAMBLE state.
POSTAMBLE	We declared counter in RD_FSM block that counts cycle from CK_double that represents interval of postamble and when this counter finishes, it outputs signal called POST_Ctr_Done. After this signal is raised high this means that postamble phase has finished so we can say that reading data has finished then go to RD_DONE state.

DDR5 SDRAM Memory Controller Design and Verification

RD_DONE	In this state, Read_Done signal is asserted high to indicate that reading has finished then go to IDLE state waiting another read operation.
----------------	---

Table 36: Outputs of each state in Read_FSM

State	Outputs
IDLE	<ul style="list-style-type: none"> ➤ Read_Done = 0 ➤ RD_Enable = 0 ➤ PRE_Ctr_Reset_n = 0 ➤ Data_Ctr_Reset_n = 0 ➤ POST_Ctr_Reset_n = 0
PREAMBLE	<ul style="list-style-type: none"> ➤ PRE_Ctr_Reset_n = 1
RD_DATA	<ul style="list-style-type: none"> ➤ Data_Ctr_Reset_n = 1 ➤ RD_Enable = 1 ➤ Ctrl_DataOut<=DQ
POSTAMBLE	<ul style="list-style-type: none"> ➤ POST_Ctr_Reset_n = 1 ➤ RD_Enable = 0
RD_DONE	<ul style="list-style-type: none"> ➤ Read_Done = 1

❖ Assumption & Notes

1. RD_Enable is internal signal that enables reading in RD_DATE state.
2. PRE_Ctr_Reset_n and POST_Ctr_Reset_n are reset signals for preamble and postamble counters respectively.
3. Data_Ctr_Reset_n is rest signal for counter that counts burst length.

DDR5 SDRAM Memory Controller Design and Verification

2.6.2 Write_FSM

2.6.2.1 Block Diagram

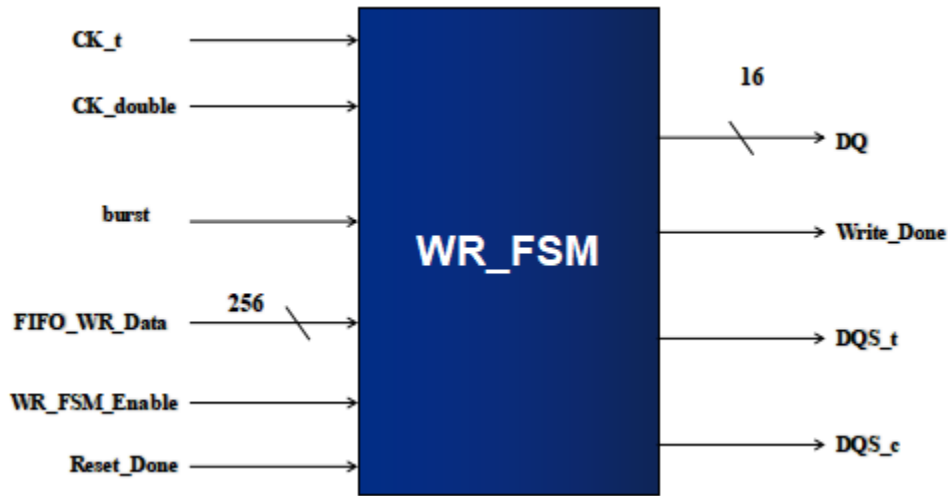


Figure 27: Block Diagram of Write_FSM

2.6.2.2 List of Inputs and Outputs

Table 37: Inputs of Write_FSM

Signal	Description
CK_t	One of differential clock of SDRAM DDR5
CK_double	Clock that has double frequency of SDRAM clock ,DQS signals will be generated with positive edge of this clock,also data from memory will be sampled also on the positive edge of this clock
burst	Flag signal when 1 defines burst length is 8 and when 0 defines burst length 16
FIFO_WR_Data[255:0]	Data that will be written in memory
WR_FSM_Enable	Enable signal of Write FSM
Reset_Done	When 1 defines that initialization sequence has been executed

Table 38: Outputs of Write_FSM

Signal	Description
DQ[15:0]	Data Input/Output: Bi-directional data bus
Write_Done	Flag signal when 1 defines that writing operation has done
DQS_t,DQS_c	Data Strobe: output with read data, input with write data. Edge-aligned with read data, centered in write data.DDR5 SDRAM supports differential data strobe only and does not support single-ended.

DDR5 SDRAM Memory Controller Design and Verification

2.6.2.3 Operation

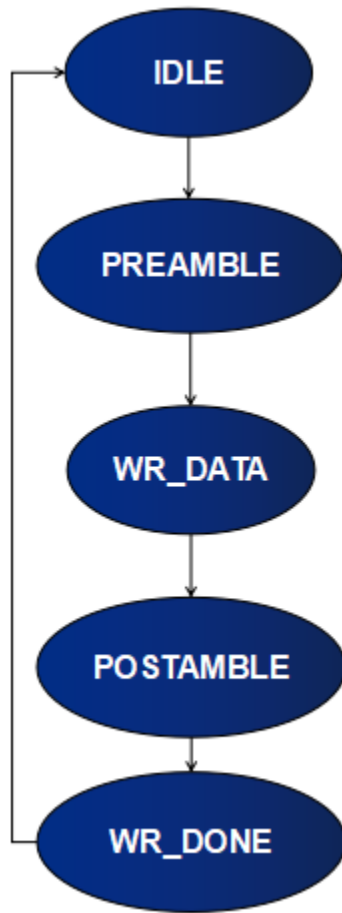


Figure 28: Write_FSM

Table 39: Description of each state in Write_FSM

State	Description
IDLE	In this state, WR_FSM_Enable is checked, if it's asserted high then go to PREAMBLE state.
PREAMBLE	We declared counter in RD_FSM block that counts two cycles from CK_double that represents interval of preamble and when this counter finishes, it outputs signal called PRE_Ctr_Done. After this signal is raised high this means that preamble phase has finished so we can write data by going to WR_DATA state.
WR_DATA	In this state writing data is done by transfer data from controller to memory with burst length that defined by burst signal, and there is internal signal when transfer is done it's raised high called Data_Ctr_Done then go to POSTAMBLE state.
POSTAMBLE	We declared counter in WR_FSM block that counts cycle from CK_double that represents interval of postamble and when this counter finishes, it outputs signal called POST_Ctr_Done. After this signal is

DDR5 SDRAM Memory Controller Design and Verification

	raised high this means that postamble phase has finished so we can say that writing date has finished then go to WR_DONE state.
WR_DONE	In this state, write_Done signal is asserted high to indicate that writing has finished then go to IDLE state waiting another write operation.

Table 40: Outputs of each state in Write_FSM

State	Outputs
IDLE	<ul style="list-style-type: none"> ➤ Write_Done = 0 ➤ WR_Enable = 0 ➤ PRE_Ctr_Reset_n = 0 ➤ Data_Ctr_Reset_n = 0 ➤ POST_Ctr_Reset_n = 0
PREAMBLE	<ul style="list-style-type: none"> ➤ PRE_Ctr_Reset_n = 1 ➤ DataIn_LD = 1
WR_DATA	<ul style="list-style-type: none"> ➤ PRE_Ctr_Reset_n = 1 ➤ Data_Ctr_Reset_n = 1 ➤ WR_Enable = 1 ➤ DQ_Transfer_En = 1 ➤ DQ<=Ctrl_DataOut
POSTAMBLE	<ul style="list-style-type: none"> ➤ PRE_Ctr_Reset_n = 1 ➤ POST_Ctr_Reset_n = 1 ➤ WR_Enable = 0 ➤ DQ_Transfer_En <= 0
WR_DONE	<ul style="list-style-type: none"> ➤ Write_Done = 1

DDR5 SDRAM Memory Controller Design and Verification

❖ Assumption & Notes

1. WR_Enable is internal signal that enables writing in RD_DATE state.
2. PRE_Ctr_Reset_n and POST_Ctr_Reset_n are reset signals for preamble and postamble counters respectively.
3. Data_Ctr_Reset_n is reset signal for counter that counts burst length.
4. DataIn_LD is signal that loads data from WR_Data_FIFO to register called DataIn.
5. DQ_Transfer_En is signal that enables transfer from DataIn register to DQ during WR_DATA state.

➤ **Problem_4:** due to the fact that there are many features the memory doing them and our controller handling them so there are many blocks drive the same signal based on what operation is handled like CA signal which leads to synthesis problem (multiple driven) as shown in Figure 29:



```
Synthesis (45 critical warnings)
  [Synth 8-6859] multi-driven net on pin CA_OBUF[13] with 1st driver pin 'init_fsm/CA_reg[13]/Q' [INIT_FSM.sv:145] (44 more like this)
  [Synth 8-6859] multi-driven net on pin CA_OBUF[13] with 2nd driver pin 'SR_FSM/CA_reg[13]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[13] with 3rd driver pin 'cmd_fsm/CA_reg[13]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[12] with 1st driver pin 'init_fsm/CA_reg[12]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[12] with 2nd driver pin 'SR_FSM/CA_reg[12]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[12] with 3rd driver pin 'cmd_fsm/CA_reg[12]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[11] with 1st driver pin 'init_fsm/CA_reg[11]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[11] with 2nd driver pin 'SR_FSM/CA_reg[11]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[11] with 3rd driver pin 'cmd_fsm/CA_reg[11]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[10] with 1st driver pin 'init_fsm/CA_reg[10]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[10] with 2nd driver pin 'SR_FSM/CA_reg[10]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[10] with 3rd driver pin 'cmd_fsm/CA_reg[10]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[9] with 1st driver pin 'init_fsm/CA_reg[9]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[9] with 2nd driver pin 'SR_FSM/CA_reg[9]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[9] with 3rd driver pin 'cmd_fsm/CA_reg[9]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[8] with 1st driver pin 'init_fsm/CA_reg[8]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[8] with 2nd driver pin 'SR_FSM/CA_reg[8]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[8] with 3rd driver pin 'cmd_fsm/CA_reg[8]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[7] with 1st driver pin 'init_fsm/CA_reg[7]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[7] with 2nd driver pin 'SR_FSM/CA_reg[7]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[7] with 3rd driver pin 'cmd_fsm/CA_reg[7]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[6] with 1st driver pin 'init_fsm/CA_reg[6]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[6] with 2nd driver pin 'SR_FSM/CA_reg[6]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[6] with 3rd driver pin 'cmd_fsm/CA_reg[6]/Q' [CMD_FSM.sv:490]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[5] with 1st driver pin 'init_fsm/CA_reg[5]/Q' [INIT_FSM.sv:145]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[5] with 2nd driver pin 'SR_FSM/CA_reg[5]/Q' [SR_FSM.sv:203]
  [Synth 8-6859] multi-driven net on pin CA_OBUF[5] with 3rd driver pin 'cmd_fsm/CA_reg[5]/Q' [CMD_FSM.sv:490]
```

Figure 29: Multiple Driven Problem.

✓ **Solution_4:** by putting MUX before multiple driven signals and select the right output based on select lines that come from a decoder that decodes internal signals. So we added to blocks in our design Selection Decoder and MUX used in Top Module before multiple driven signals, let us discuss Selection Decoder block.

DDR5 SDRAM Memory Controller Design and Verification

2.6.3 Selection Decoder

2.6.3.1 Block Diagram

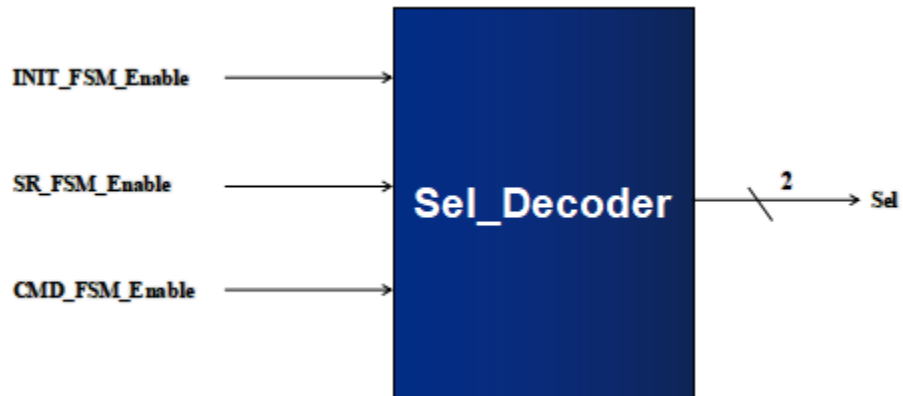


Figure 30: Block Diagram of Selection_Decoder.

2.6.3.2 List of Inputs and Outputs

Table 41: Inputs of Selection Decoder.

Signal	Description
INIT_FSM_Enable	Active high enable of Initialization_FSM
SR_FSM_Enable	Active high Enable of Self Refresh_FSM
CMD_FSM_Enable	Active high Enable of CMD_FSM

Table 42: Outputs of Selection Decoder.

Signal	Description
Sel[1:0]	Selection signal that will be selection signal for multiplexers that are placed before multiple driven signals

2.6.3.3 Operation

Table 43: Operation of Selection Decoder.

Enable Signal	Sel
INIT_FSM_Enable	0
SR_FSM_Enable	1
CMD_FSM_Enable	2

- **Problem_5:** inferring latches, we don't need latches in our design due to the complexity of calculating timings of latches at synthesis tools (checking timing violations is very complex).

DDR5 SDRAM Memory Controller Design and Verification

- ✓ **Solution_5:** we remove all latches and either return them to their equivalent combinational logic at the case of unintentional latches or exchange it with registers at the case of using it as storing element.
- **Problem_6:** sometimes synthesis tools remove registers due to they think that these registers will never be used.
- ✓ **Solution_6:** we added a load signal for registers that may be removed to make the tool understand that it actually used when this load gets activated then register a new value.
- **Problem_7:** clock cycle uncertainty between blocks in top module.
- ✓ **Solution_7:** we made a global initialization for all blocks and counters based on initialization feature of memory and also made counters to be aligned in counting with other blocks.
- **Problem_8:** Reset signals have not highest priority in counters and other blocks.
- ✓ **Solution_8:** we redefined all counters and blocks and chose to model counters with global reset, set priority counters to get instantiated with the optimized version of it after synthesis and that is by putting asynchronous reset at the top with the highest priority then even counts incrementing the counter when enabled or keeps the old one.
- **Problem_9:** There are bits in address vector may be removed by tool as it thinks that they won't be used and this due to that not all bits in Address vector, defined by the standard, is used in our design.
- ✓ **Solution_9:** we needed to redefine address vector to make all its bits get used from the design and not letting the tool to remove any signal with its own.
- **Problem_10:** there are non-synthesizable statements on System-Verilog HDL like wildcard equality operator "==="
- ✓ **Solution_10:** we exchanged all these statements with synthesizable statement like logical equality operator "==".

Some of these problems that we discussed are shown in Figure 31:

DDR5 SDRAM Memory Controller Design and Verification

▼ Synthesis (302 warnings)

- › [Synth 8-6014] Unused sequential element Prev_Reset_reg was removed. [Command_Decoder.sv:83] (28 more like this)
- › [Synth 8-87] always_comb on 'WR_Data_WR_En_reg' did not result in combinational logic [Command_Decoder.sv:206] (87 more like this)
- › [Synth 8-5788] Register DataOut_reg[15] in module RD_FSM is has both Set and reset with same priority. This may cause simulation mismatches. Consider rewriting code [RD_FSM.sv:288] (61 more like this)
- › [Synth 8-589] replacing case/wildcard equality operator === with logical equality operator == [CMD_FSM.sv:203] (4 more like this)
- › [Synth 8-327] inferring latch for variable 'WR_Data_WR_En_reg' [Command_Decoder.sv:206] (93 more like this)
 - [Synth 8-3936] Found unconnected internal register 'Command_reg' and it is trimmed from '7' to '3' bits. [Command_Decoder.sv:66]
- › [Synth 8-3331] design CMD_FSM has unconnected port Address[4] (1 more like this)
- › [Synth 8-3332] Sequential element (DQ_reg_130) is unused and will be removed from module WR_FSM. (15 more like this)
- › [Synth 8-264] enable of latch \cmd_fsm\WR_FSM_Enable_reg is always disabled (3 more like this)

Figure 31: Some Problems of Synthesis.

Finally, all problems were solved and the design get synthesized properly. And the schematic from synthesis tool is shown in Figure 32:

❖ Assumption & Notes

1. We used System Verilog HDL to describe our design.
2. Tools that we used in design through project Vivado ,Modelsim,etc.
3. There are changes in design based on bugs that verification found will be discussed later.

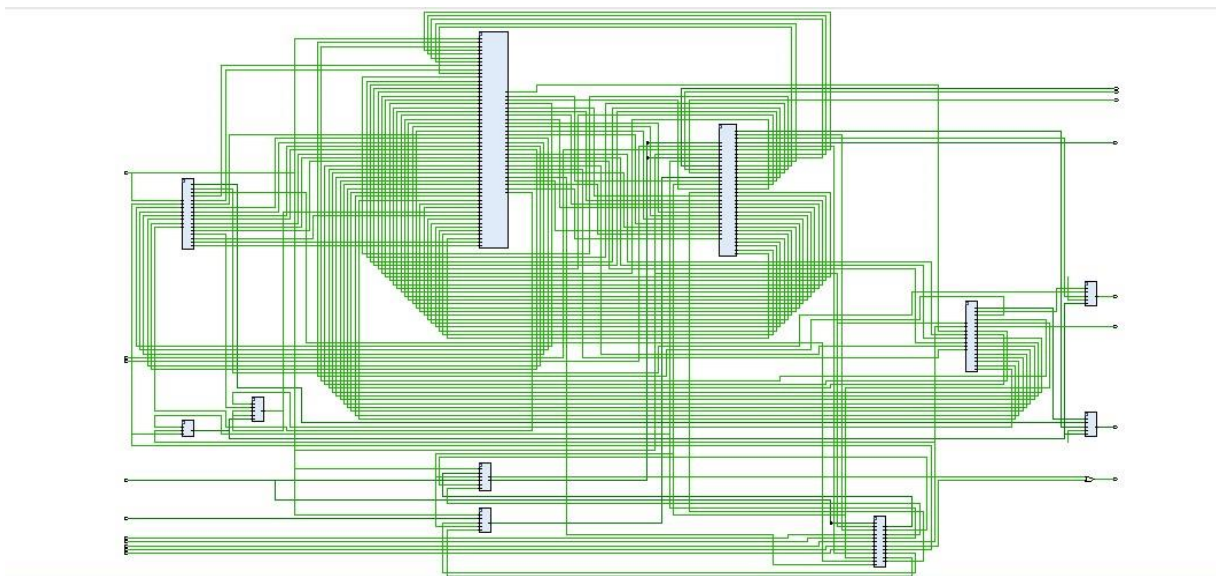


Figure 32: Schematic from Synthesis Tool (Vivado).

DDR5 SDRAM Memory Controller Design and Verification

Chapter 3: Introduction to Cocotb

3.1 INTRODUCTION

Modern system-on-chip (SoC) designs have been evolving towards heterogeneous compositions of general purpose and specialized computing fabrics as Dennard scaling has ended and Moore's law has slowed. This heterogeneity makes the already difficult work of SoC design and verification much more difficult. Multiple generations of open-source hardware modelling frameworks have attempted to address the growing complexity of hardware design and verification. Comprehensive, productive, and open-source verification procedures that decrease our necessary to build completely validated hardware blocks are a critical missing component in the open-source hardware ecosystem. Verification of open-source hardware has numerous substantial hurdles as compared to closed-source hardware. Closed source hardware, for starters, is typically owned and maintained by firms with specialized verification teams. These verification engineers often have a lot of expertise with constraint-based random testing using commercial System Verilog simulators utilizing a universal verification methodology (UVM). Open-source hardware teams, on the other hand, typically use an agile test-driven design method borrowed from the open-source software community, in which the designer is also responsible for writing the tests. Furthermore, due to the high learning curve and limited support in existing open-source tools, open-source hardware teams seldom employ the UVM-based method. Instead of replicating closed-source hardware testing frameworks, the open-source hardware industry deliberately needs an alternate way for verifying open-source hardware. The top-down approach offered by UVM does not work well for complex multimedia IP blocks like image signal processing pipelining, video codec, neural processing unit etc. due to the algorithmic/system architecture complexity. An SoC chain can contain more than 20 blocks, which a verification testbench is expected to handle. There is a need for SoC DV to be able to take a portion of the IP DV environment and be able to re-run valid semi-randomized scenarios at SoC level. To fully address SoC-level verification, a solution must extend from UVM and allow for vertical (IP to SoC) reuse and horizontal (verification engine portability) reuse. A solution must provide a way to capture, share, and automatically amplify use cases to speed test-case creation and leverage fast verification engines.

3.2 BACKGROUND

Design Verification is a process in which a design is compared against a given design specification before tape-out. This happens along with the development of the design and can start from the time the design architecture definition is completed. The main goal of verification is to ensure functional correctness of the design. However, with increasing design complexities, the scope of verification is also evolving to include much more than functionality. This includes verification of performance and power targets, security and safety aspects of design and complexities with multiple asynchronous clock domains. Simulation of the design model (RTL) remains the primary vehicle for verification while a lot of other methodologies like formal property verification, power-aware simulations, emulation/FPGA prototyping, static and dynamic checks, etc. are also used for efficiently verifying all aspects of design. The Verification process is considered very critical as part of design life cycle as any serious bugs in design not discovered before tape-out can lead to the need of newer steppings and increasing the overall cost of design process.

DDR5 SDRAM Memory Controller Design and Verification

3.2.1 Functional Verification

The known as functional verification. Functional verification does not confirm the correctness of the design specification and instead assumes that it is correct. It is one of the most difficult steps in the IC design cycle and the primary cause of IC re-spin. The main objectives are: Functional correctness of individual IPs, Internal module communication, External module communication, End to end functional paths, Clock and reset circuits, Power up and down sequence, Complete integration of all IPs. Different types of Functional Verification methods are shown in Figure 33.

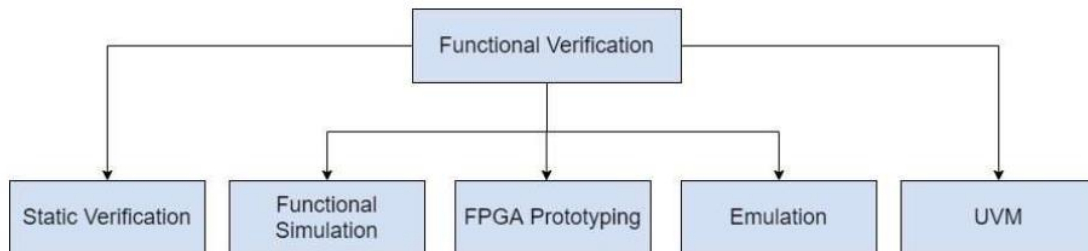


Figure 33: Types of Functional Verification

1) **Static Verification:** It is the process of checking a design against some predefined rules without running it. It enables validation of design at an early stage, without any stimulus or setup, and is thus performed early in the IC design cycle, that is, as soon as the RTL code is available. It doesn't do any timing checks. The earlier a bug is discovered, the easier it is to fix it. The goal of static verification is to decrease the verification effort at the RTL level.

2) **Functional Simulation:** The process of simulating a design's functional behavior in software is known as functional simulation. It is not useful in software development because it does not account for the timing delays of internal logic or interconnects. The goal of simulation is to validate the individual IPs or blocks of the IC. Functional simulation does not allow for system-level verification.

3) **FPGA Prototyping:** FPGA prototyping is the process of testing the functionality of an integrated circuit (IC) on FPGAs. With the increasing complexity of ICs and the increasing demand to reduce IC time to market, FPGA prototyping remains a critical solution. The goal of FPGA prototyping is to ensure that the design works as expected when driven with live data and that all of its external interfaces are operational.

4) **Emulation:** Emulation, also known as pre-silicon validation, is the process of testing the system's functionality on a hardware device known as an emulator. An emulator can handle both system-level and RTL designs (written in C, C++, or SystemC) (in Verilog or VHDL). Simulators take much longer to run than emulators. A design that takes days to simulate will only take hours to emulate. Emulation is used to find issues in system level design using live data, to verify system integration and to develop embedded software.

5) **Universal Verification Methodology (UVM):** UVM is a well-defined set of coding guidelines with a well-defined testbench structure. It's written in SystemVerilog and comes with a SystemVerilog base class library for creating advanced reusable verification components. It was

DDR5 SDRAM Memory Controller Design and Verification

created with significant guidance and input from Mentor by the Accellera Systems Initiative, an EDA standards body. IPs are extremely complex, and fully verifying them takes time. The standard test benches are not reusable, so verification engineers must build them from scratch. Due to time constraints, a verification methodology is highly recommended. UVM has a fixed testbench architecture, which makes the testbench highly reusable and saves time.

3.2.2 Switching to Python

SystemVerilog is a fairly complex programming language. The SystemVerilog specification is almost a thousand pages long. There are 221 keywords in the language, compared to 83 in C++. It's a powerful tool, but it takes some time to master. UVM has comparable concerns with complexity. There are numerous ways to accomplish the same task. Again, highly powerful, but difficult to master. Ergo, SV-UVM is powerful but complicated. So, hardware description languages are kept for designing whereas for developing testbenches, a high-level, general-purpose language with object-oriented programming is considerably more beneficial. Thus, Cocotb was created.

3.3 DESIGN VERIFICATION USING COCOTB

Cocotb automatically connects to a variety of HDL simulators (such as Icarus, Modelsim, Questasim, and others) and allows you to control the signals in your design straight from Python. The whole testbench may be written in Python, and automation and randomization are simple to implement, resulting in increased productivity. Cocotb does not necessitate the use of any additional RTL code. In the simulator, the top level is instantiated as the Design Under Test. Python is used to provide stimulation to the DUT's inputs and monitor the outputs. Given that it does not necessitate knowledge of HDLs, it can be of great help to those who are unfamiliar with it. Python is also an object-oriented scripting language. Cocotb has certain significant advantages over HDL testing techniques since it uses Python for verification:

- Python is an extremely productive language that allows one to write code quickly
- Python makes it simple to connect to other languages.
- Python contains a large library of pre-existing code that can be reused.
- Python is an interpreted language, which means that tests can be modified and rerun without having to recompile the design or exit the simulator GUI.
- Python is widely used; significantly more engineers are familiar with it than SystemVerilog or VHDL.

DDR5 SDRAM Memory Controller Design and Verification

3.3.1 Architecture of Cocotb

A normal Cocotb testbench does not necessitate any additional RTL code. Without any wrapper code, the Design Under Test (DUT) is instantiated as the simulator's top level. Cocotb applies stimuli to the DUT's inputs (or lower in the hierarchy) and monitors the outputs directly from Python. Cocotb acts as a bridge between the simulator and Python as shown in Figure 34 [9]. Verilog Procedural Interface (VPI) or VHDL Procedural Interface (VHDLPI) is used (VHPI).

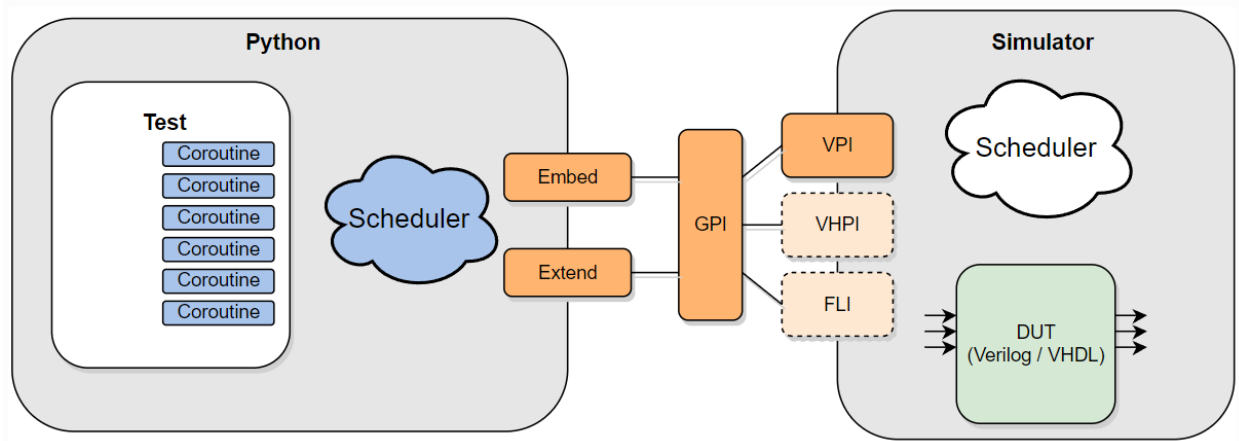


Figure 34: Architecture of Cocotb.

A test is merely a Python function. The await keyword indicates when control of execution should be returned to the simulator. A test can start numerous coroutines, permitting separate execution flows. Python testbench code has the ability to [10]:

- Traverse the DUT hierarchy and update values.
- Wait for the simulation timer to run out.
- Wait for a signal's rising or falling edge.

3.3.2 Design Methodology

The cocotb framework is made to be a goal-directed design verification tool. The following steps are included in the python-based verification flow [11].

- 1) Capture the IP-level actions needed to create a desired use case, if not already captured.
- 2) Compose the desired use case in text format.
- 3) Use cocotb for vector generation: cocotb allows constrained randomization through which all the parameters of the IP core can be randomized.
- 4) Verify the resulting vectors on a golden reference: These vectors can be run on a C test design and the validity of vectors can be checked.

DDR5 SDRAM Memory Controller Design and Verification

3.3.3 Cosimulation

It is the independent simulation of the design and testbench. Communication is accomplished using VPI/VHPI interfaces, which are represented by cocotb ‘triggers’. The simulation time does not advance while the Python function is running. When a trigger is delivered, the testbench suspends execution until the triggered condition is met before restarting execution. Some triggers available are [9]:

- Timer (time, unit): Waits for a given amount of simulation time to pass before acting.
- Edge(signal): Waits for a signal’s state to change (rising or falling edge).
- RisingEdge(signal): Waits for a signal’s rising edge.
- FallingEdge(signal): Waits for a signal’s falling edge.
- ClockCycles(signal, num): Waits for a certain number of clocks to cycle (transitions from 0 to 1).

3.4 COCOTB COVERAGE

3.4.1 Functional Coverage in SystemVerilog

In SystemVerilog a fundamental coverage unit is a *coverpoint*. It contains several bins and each bin may contain several values. Every *coverpoint* is associated with a variable or signal. At sampling event, the *coverpoint* variable value is compared with each defined bin. If there is a match, then the number of hits of the particular bin is incremented. *Coverpoints* are organized in *covergroups*, which are specific class-like structures. A single *covergroup* may have several instances and each instance may collect coverage independently. A *covergroup* requires sampling, which may be defined as a logic event (e.g., a positive clock edge). Sampling may also be called implicitly in the testbench procedural code by invoking a *sample()* method of the *covergroup* instance. A bin may be also defined as an *ignore_bins*, which means its match does not increase a coverage count, or an *illegal_bins*, which results in error when hit during the test execution.

Another coverage construct in SystemVerilog is a *cross*. It automatically generates a Cartesian product of bins from several *coverpoints*. It is a useful feature simplifying the functional coverage generation. As it may be difficult or unnecessary to cover all the cross-bins, some of them may be excluded from the analysis. This is possible using the *binsof ... intersect* syntax.

The most important limitations of the SystemVerilog functional coverage features are:

- straightforward bins matching criteria – only satisfied by equality or inclusion relation;
- bins may be only constants or transitions (possibly wildcard);
- flat coverage structure – cover groups cannot contain other cover groups, which would correspond better to a verification plan scheme;

DDR5 SDRAM Memory Controller Design and Verification

- not possible to get the detailed coverage information in real time (e.g., when a specific bin was hit).

3.4.2 Functional Coverage with Cocotb-coverage

The general assumptions for the architecture of the functional coverage features are as follows:

- functional coverage structure should better match a real verification plan;
- its syntax should be more flexible, but a separation between coverage and executable code should be maintained;
- features for analyzing the coverage during test execution should be added or extended;
- coverage primitives should be able to monitor testbench objects at a higher level of abstraction.

The implemented mechanism is based on the idea of decorator design pattern. In Python, a decorator syntax is readable and easy to use. Instead of sampling coverage items by an additional method, decorators are by default invoked at each decorated function call. As it is easy to create functions in Python (for example anonymous functions can be created as lambda expressions<lambda> – single-line function definitions), this is a convenient solution. The coverage structure is based on a prefix tree (a *trie*). The main coverage primitive is a *CoverItem*, which corresponds to a SystemVerilog *covergroup*. *CoverItem* may contain other *CoverItems*<*CoverItem*> or objects extending *CoverItems*<*CoverItem*> base class, which are *CoverPoints*<*CoverPoint*>, *CoverCrosses*<*CoverCross*> or arbitrary new, user-defined types. *CoverItems*<*CoverItem*> are created automatically, the user defines only *CoverPoint* or *CoverCross* primitives (the lowest level nodes in the trie). Each created primitive has a unique ID – a dot-separated string. This string denotes the position of an object in the coverage trie. For example, a *CoverPoint a.b.c* is a member of the *a.b* *CoverItem*, which is then a member of the *a* *CoverItem*. The structure of the coverage tree is presented below in figure 35.

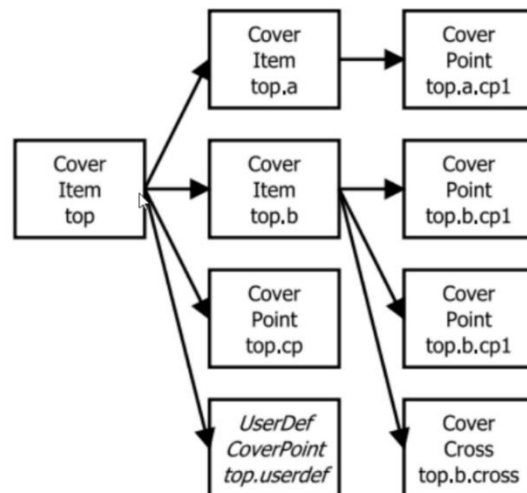


Figure 35: An example of the coverage tree structure [12].

DDR5 SDRAM Memory Controller Design and Verification

3.4.3 Constrained Random Verification Features in SystemVerilog

SystemVerilog users may define random variables using the *rand/randc* modifier. Calling *randomize()* function on a class instance (object) results in picking random values of the defined random variables, satisfying given constraints. Also a *with* modifier can be used together with *randomize()* which allow for appending additional constraints dynamically. Constraints are defined in a special section in the class named *constraint*. They describe a range values that a single variable may have or a relation between variables. It is also possible to define solution ranges with weights (using *dist* modifier). The *solve ... before* is an additional construction which organizes variable randomization order.

Constraints are unique constructs of SystemVerilog. They are class members, but they are not functions or objects. Basic operations can be performed on constraints, such as enable/disable or inheritance. Soft constraints have been introduced in SystemVerilog 2012. They are resolved only when it is possible to satisfy them together with all other hard constrains. Every SystemVerilog simulator must implement a constraint solver. Although many open-source constraint solvers are available, testbench designers cannot use them, as they have no control over the simulator engine. The most important limitations of the existing constrained randomization features are related to their fixed syntax.

In cocotb-coverage, it is assumed that a constraint may be any callable object – an arbitrary function or a class with `__call__` method. It allows for creating various functionalities quite easily and manipulating them in a flexible way [12].

3.4.4 Constrained Random Verification Features in cocotb-coverage

The main assumption for the constrained randomization features was to provide only a flexible API, and let the testbench designer to adjust it depending on project needs. There is an open-source based hard constraint solver used by this framework: python-constraint.

The general idea of Cocotb-coverage is that all classes that intended to use randomized variables should extend the base class `Randomized`. Afterwards, random variables and their ranges should be defined. Constraints are just arbitrary functions with only one requirement: their argument names must match class member names. It is possible to define two types of constraints:

functions that return a True/False value, corresponding to SystemVerilog hard constraints;

functions that return a numeric value, corresponding to a variable's distribution (or cross-distribution) which also may be used as soft constraints.

The `Randomized` class API consists of the following functions:

- `add_rand(var, domain)<add_rand>` - specifies `var` as a randomized variable taking values from the domain list;
- `add_constraint(cstr)<add_constraint>` - adds a constraint function to the solver;

DDR5 SDRAM Memory Controller Design and Verification

- `del_constraint(ctr)<del_constraint>` - removes a constraint function from the solver;
- `solve_order(vars0, vars1 ...)<solve_order>` - optionally specifies the order of randomizing variables (can be used for problem decomposition or in case some random variables must be fixed before randomizing the others);
- `pre_randomize` - function called before `randomize/randomize_with`, corresponding to similar function in SV;
- `post_randomize` - function called after `randomize/randomize_with`, corresponding to similar function in SV;
- `randomize()` - main function that picks random values of the variables satisfying added constraints;
- `randomize_with(ctr0, ctr1 ...)<randomize_with>` - similar to `randomize()`, but satisfies additional given constraints.

A more complex example is presented below. The class `TripleInt` contains three unsigned integer members, `y` and `z` are randomized. The first defined constraint combines all variables (random and non-random). The second constraint defines a triangular distribution for variable `z`. It is achieved by defining a function that has its maximum in the middle of the variable range (for solution $z = 500$). The third one is a cross-distribution of variables `y` and `z`. The weight function defines higher probability for solutions with higher difference between both variables. The last one is a kind of a soft constraint – very low probability is set for condition $x > y$, which means that solutions satisfying $x \leq y$ will be strongly preferred.

```
class TripleInt(crv.Randomized):
    def __init__(self, x):
        crv.Randomized.__init__(self)
        self.x = x # this is a non-random value, determined at class instance creation
        self.y = 0
        self.z = 0
        add_rand(y, list(range(1000))) # 0 to 999
        add_rand(z, list(range(1000))) # 0 to 999
        add_constraint(lambda x, y, z: x+y+z==1000) # hard constraint
        add_constraint(lambda z: 500 - abs(500-z)) # triangular distribution of z variable
        add_constraint(lambda y, z: 100 + abs(y-z)) # multi-dimensional distribution
        add_constraint(lambda x, y: 0.01 if (y > x) else 1) # soft constraint
```

It is assumed that only one hard constraint and one distribution may be associated with each set of random variables. So, for the example presented above, it is possible to define no more than six constraint functions: separately for variables `y` and `z` and both (*y and z*). It means that constraints may be overwritten, for example by `randomize_with()` function arguments.

3.5 CODE COVERAGE

Code Coverage testing determines how much code is tested. Code coverage is a metric that describes the extent to which the program's source code has been tested. It is given by the Eqn. 1:

DDR5 SDRAM Memory Controller Design and Verification

$$\text{Code Coverage} = \frac{\text{Number of lines of code excuted}}{\text{Total number of lines of code}} * 100 \% \quad (1)$$

There are several coverage types, which are as follows [13]:

3.5.1 Statement coverage/ Line coverage

Statement coverage, often known as line coverage, is the simplest to comprehend sort of coverage. Statement coverage measures how many statements/lines are covered in the simulation.

3.5.2 Block/ Segment coverage

The nature of the statement and block coverage seems to be similar. The distinction is that block coverage takes into account branching blocks of if/else, case branches, wait, while, for, and so on. The dead code (lines which never get executed) is revealed by analyzing block coverage.

3.5.3 Conditional coverage

Conditional coverage, also known as expression coverage, shows how variables or expressions in conditional statements are assessed. Only expressions using logical operators are taken into account. Conditional coverage is the ratio of number of cases checked to the total number of instances present.

3.5.4 Branch coverage

Branch coverage, also known as decision coverage, reports the true or false of conditions such as if-else, case, and ternary operator statements. Decision coverage for an ‘if’ statement will report if the ‘if’ statement is examined in both true and false instances, even if a ‘else’ statement does not exist.

3.5.5. Toggle coverage

It ensures how many times variables and nets are toggled (flipping between logic high and logic low). Toggle coverage is just the ratio of toggled nodes to total nodes.

3.5.6. Path coverage

Due to conditional statements such as if-else, a different path is generated in the design, diverting the flow of input to the specific path. Path coverage is regarded to be more comprehensive than branch coverage since it can detect flaws in the order of operations.

3.5.7. FSM coverage

As it works on the design’s behavior, it is the most complex sort of code coverage. In a finite state machine, this evaluates how often states are visited, transited, and how many sequences are covered.

DDR5 SDRAM Memory Controller Design and Verification

Chapter 4: Block Level Verification

4.1 GOALS AND OVERVIEW

The goal of this chapter is to write a block level verification plan to ensure the functionality of each block, report the critical bugs of each block and how it was fixed, measure the functional coverage and the code coverage of each block to ensure the completeness of the written testbenches, which are written in python using Cocotb Coverage library as mentioned in chapter3.

4.2 BLOCK LEVEL TESTBENCH ARCHITECTURE

The test bench architecture is based on Self-checking coverage-Driven Constraint Random-Based Functional Verification Methodology, the function of each block as follows:

Generator: generates constrained random test cases.

Driver: drives the test cases to the Device under test and the Reference Model concurrently.

Reference Model: provides the expected output according to the current testcase.

Checker: compares the predicted output with the DUT output.

Scoreboard: prints the failed test cases and the passed ones.

Coverage: samples the test inputs to collect the features that have been tested.

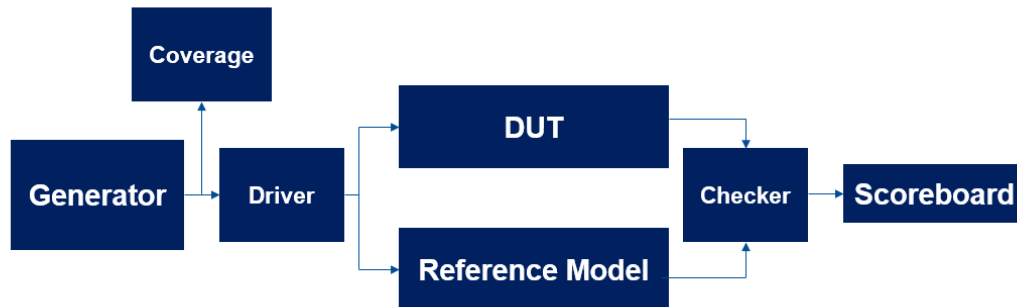


Figure 36: Block Level Verification Environment.

4.3 COMMAND DECODER VERIFICATION PLAN

4.3.1 Functional Coverage Plan

The important features that should be covered to ensure the correctness of the Command decoder functionality are the following:

1. Reset.
2. Self-Refresh.
3. Reset and Self_Refresh/command in the same time to ensure the priority of reset.

DDR5 SDRAM Memory Controller Design and Verification

4. self-refresh and command in the same time to ensure the priority of self-Refresh.
5. Write.
6. Read.
7. Write with AP.
8. Read with AP.
9. Write Burst.
10. Read Burst.
11. Write Burst with AP.
12. Read Burst with AP.
13. Write after Read in same Bank Group.
14. Write after Read in different Bank Group.

```

Command_Decoder_Coverage = coverage_section (
    CoverPoint("top.Read", vname="Ctrl_Read_DUT", bins = [1, 0]),
    CoverPoint("top.Write", vname="Ctrl_Write_DUT", bins = [1, 0]),
    CoverPoint("top.Reset", vname="Ctrl_Reset_DUT", bins = [1, 0]),
    CoverPoint("top.Burst", vname="Ctrl_Burst_DUT", bins = [1, 0]),
    CoverPoint("top.CMD_Done", vname="CMD_Done_DUT", bins = [1, 0]),
    CoverPoint("top.Reset_Done", vname="Reset_Done_DUT", bins = [1, 0]),
    CoverPoint("top.tREFI", vname="tREFI_DUT", bins = [1, 0]),
    CoverPoint("top.Auto", vname="Ctrl_Auto_DUT", bins = [1, 0]),
    CoverPoint("top.SR_Done", vname="SR_Done_DUT", bins = [1, 0]),
    CoverPoint("top.CMD", vname="Command_DUT", bins = [0, 1, 2, 3, 4,5,6,7]),
    CoverCross("top.Enable_FSM", items = ["top.Reset", "top.tREFI", "top.Reset_Done", "top.SR_Done", "top.CMD_Done"]),
    CoverCross("top.Commands", items = ["top.Read", "top.Write", "top.Burst", "top.Auto"])
)
    
```

Figure 37: Command decoder coverage section written in python.

4.3.2 Test Cases







Table 44: Test cases of command decoder.

Test Item	Test Case	Expected Result	Covered	Bug Free
Reset	<ul style="list-style-type: none"> ➤ Ctrl_Reset signal is asserted high for only one clock cycle then low. 	<ul style="list-style-type: none"> ➤ INIT_FSM_Enable signal should be asserted high. ➤ Memory_Busy Signal should be asserted high. 	✓	✓
Reset	<ul style="list-style-type: none"> ➤ Reset_Done is signal is asserted high for only one clock cycle then low. 	<ul style="list-style-type: none"> ➤ INIT_FSM_Enable signal should be asserted low. ➤ Memory_Busy Signal should be asserted low. 	✓	✓
Self-Refresh	<ul style="list-style-type: none"> ➤ tREFI signal is asserted high for only one clock cycle then low. ➤ Ctrl_Reset signal is asserted low. 	<ul style="list-style-type: none"> ➤ SR_FSM_Enable signal should be asserted high. ➤ Memory_Busy Signal should be asserted high. 	✓	✓





DDR5 SDRAM Memory Controller Design and Verification


Self-Refresh	<ul style="list-style-type: none"> ➤ SR_Done is asserted high for only one clock cycle then low. 	<ul style="list-style-type: none"> ➤ SR_FSM_Enable signal should be asserted low. ➤ Memory_Busy Signal should be asserted low. 	✓	✓
Write	<ul style="list-style-type: none"> ➤ Ctrl_Write signal is asserted high for only one clock cycle. ➤ Ctrl_Read signal is asserted low for only one clock cycle. ➤ Ctrl_Burst is asserted low for only one clock. ➤ Ctrl_Auto is asserted low for only one clock cycle. ➤ Bank_Group signal is asserted to a certain value 'x' for only one clock cycle. ➤ Data_Transfer_Write is asserted to a certain value for only one clock cycle. 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ WR_Data_WR_En signal should be asserted high ➤ CMD signal should be asserted '001'. ➤ First_Command should be asserted high. 	✓	✓
Read	<ul style="list-style-type: none"> ➤ Ctrl_Read signal is asserted high. ➤ Ctrl_Write signal is asserted low. ➤ Ctrl_Burst is asserted low. ➤ Ctrl_Auto is asserted low. ➤ Bank_Group signal is asserted to a certain value 'x'. 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ CMD signal should be asserted '000'. ➤ Same_Bank_Group should be asserted high. 	✓	✓
Write With AP	<ul style="list-style-type: none"> ➤ Ctrl_Write signal is asserted high. ➤ Ctrl_Read signal is asserted low. ➤ Ctrl_Burst is asserted low. ➤ Ctrl_Auto is asserted low. ➤ Bank_Group signal is asserted to a certain value 'y'. 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ WR_Data_WR_En signal should be asserted high ➤ CMD signal should be asserted '011'. ➤ Same_Bank_Group should be asserted low. 	✓	✓

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ Data_Transfer_Write is asserted to a certain value. 			
Read With AP	<ul style="list-style-type: none"> ➤ Ctrl_Read signal is asserted high. ➤ Ctrl_Write signal is asserted low. ➤ Ctrl_Burst is asserted low. ➤ Ctrl_Auto is asserted high. ➤ Bank_Group signal is asserted to a certain value. 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ CMD signal should be asserted '010'. 		
Write Burst	<ul style="list-style-type: none"> ➤ Ctrl_Write signal is asserted high. ➤ Ctrl_Read signal is asserted low. ➤ Ctrl_Burst is asserted high. ➤ Ctrl_Auto is asserted low. ➤ Bank_Group signal is asserted to a certain value. ➤ Data_Transfer_Write is asserted to a certain value. 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ WR_Data_WR_En signal should be asserted high ➤ CMD signal should be asserted '101'. 		
Read Burst	<ul style="list-style-type: none"> ➤ Ctrl_Read signal is asserted high. ➤ Ctrl_Write signal is asserted low. ➤ Ctrl_Burst is asserted high. ➤ Ctrl_Auto is asserted low. ➤ Bank_Group signal is asserted to a certain value. 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ CMD signal should be asserted '100'. 		

DDR5 SDRAM Memory Controller Design and Verification

<p>Write Burst with AP</p>	<ul style="list-style-type: none"> ➤ Ctrl_Write signal is asserted high. ➤ Ctrl_Read signal is asserted low. ➤ Ctrl_Burst is asserted high. ➤ Ctrl_Auto is asserted high. ➤ Bank_Group signal is asserted to a certain value. ➤ Data_Transfer_Write is asserted to a certain value. ➤ 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ WR_Data_WR_En signal should be asserted high ➤ CMD signal should be asserted '111'. 		
<p>Read Burst with AP</p>	<ul style="list-style-type: none"> ➤ Ctrl_Read signal is asserted high. ➤ Ctrl_Write signal is asserted low. ➤ Ctrl_Burst is asserted high. ➤ Ctrl_Auto is asserted high. ➤ Bank_Group signal is asserted to a certain value. 	<ul style="list-style-type: none"> ➤ CMD_FIFO_WR_En signal should be asserted high. ➤ CMD signal should be asserted '110'. 		

 **Note:** all signal is asserted at positive edge the clock CK_t in order to be sampled and lasts for only on clock cycle then changes.

4.3.3 Reported Bugs



Bug #1:

FSM Enables are asserted high for only one cycle although it should be high until a FSM done signal is asserted high, for example: Initialization FSM Enable is asserted high for only one cycle, although it should be high until Reset_Done signal (which comes from the initialization FSM when it finishes) is asserted high.

DDR5 SDRAM Memory Controller Design and Verification



Modification:

Initialization `_FSM_Enable` is stored in a `prev_Reset` Register in order to modify the condition that assert the Initialization FSM Enable high based on this register for illustration:

When `Ctrl_Reset` is asserted high for only one cycle: `INIT_FSM_Enable` will be asserted high as

```
INIT_FSM_Enable=! Reset_Done;
```

When `Ctrl_Reset` is asserted low at the next clock cycle:

`INIT_FSM_Enable` will be asserted high as

```
INIT_FSM_Enable= prev_Reset;
```

When the initialization FSM finishes it will assert `Reset_Done` low for only one cycle:

`INIT_FSM_Enable` will be asserted low as

```
INIT_FSM_Enable=! Reset_Done;
```

When `Reset_Done` is asserted low at the next clock cycle:

`INIT_FSM_Enable` will be asserted low as

```
INIT_FSM_Enable= prev_Reset;
```

```
if(Ctrl_Reset || Reset_Done)
begin
    INIT_FSM_Enable=!Reset_Done ;
    Memory_Busy=!Reset_Done;
    SR_FSM_Enable=0;
    CMD_FSM_Enable=0;
end
else if(Ctrl_Reset==0 && Reset_Done==0)
begin
    INIT_FSM_Enable=Prev_Reset ;
    Memory_Busy=Prev_Reset;
    SR_FSM_Enable=0;
    CMD_FSM_Enable=0;
end
```

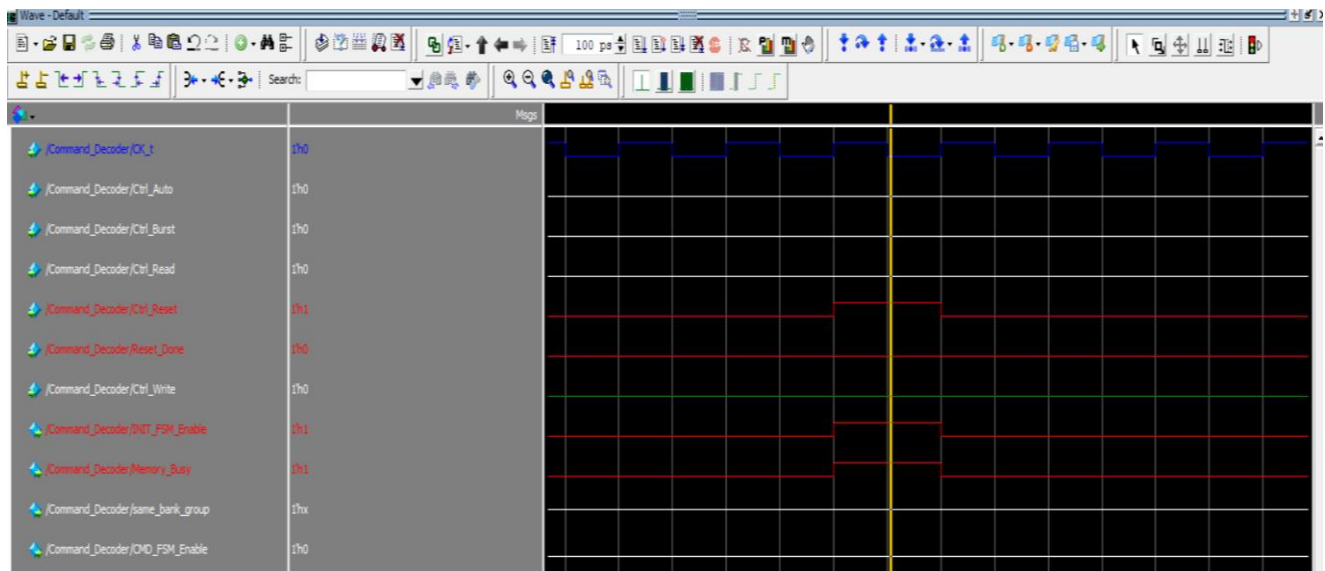


Figure 38: Initialization FSM Enable is asserted high for only one cycle.

DDR5 SDRAM Memory Controller Design and Verification

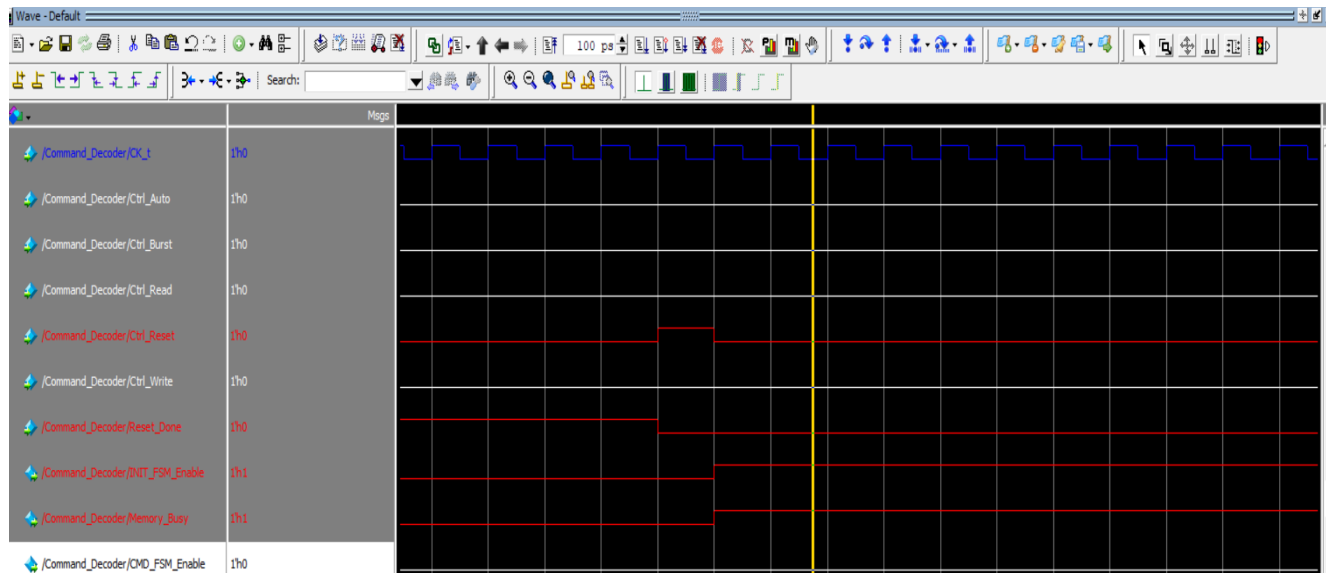


Figure 39: Initialization FSM Enable after modification.

```

top.Write : <cocotb_coverage.coverage.CoverPoint object at 0x000000003D96940>, coverage=2,

    BIN 1 : 10000
    BIN 0 : 10000

number of randomized testcases 19999
Test Passed: Command_Decoder_Test
Passed 1 tests (0 skipped)
*****
** TEST                PASS/FAIL  SIM TIME(NS)  REAL TIME(S)  RATIO(NS/S) **
*****
** Command_decoder.Command_Decoder_Test  PASS          1999.96      433.35       4.62 **
*****

*****
**                                ERRORS : 0                                **
*****

**                                SIM TIME : 1999.96 NS                                **
**                                REAL TIME : 433.43 S                                **
**                                SIM / REAL TIME : 4.61 NS/S                                **

```

Figure 40: Command Decoder Test Summary from Questasim.

DDR5 SDRAM Memory Controller Design and Verification

4.3.4 Functional Coverage Results

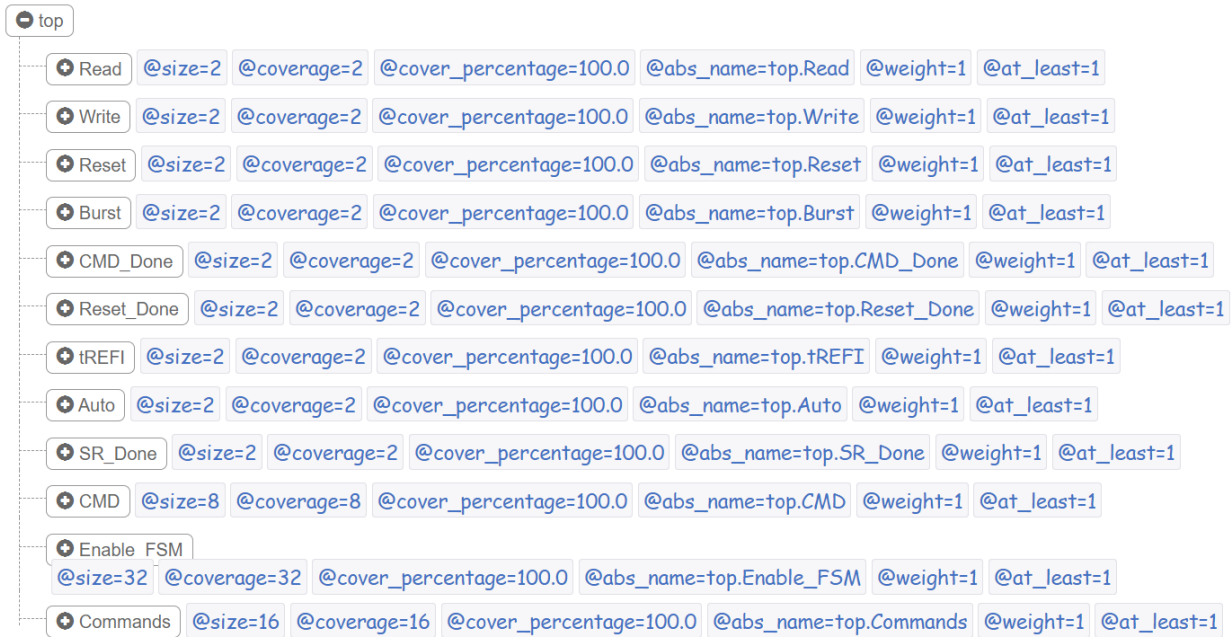


Figure 41: Command decoder Functional Coverage XML Report.

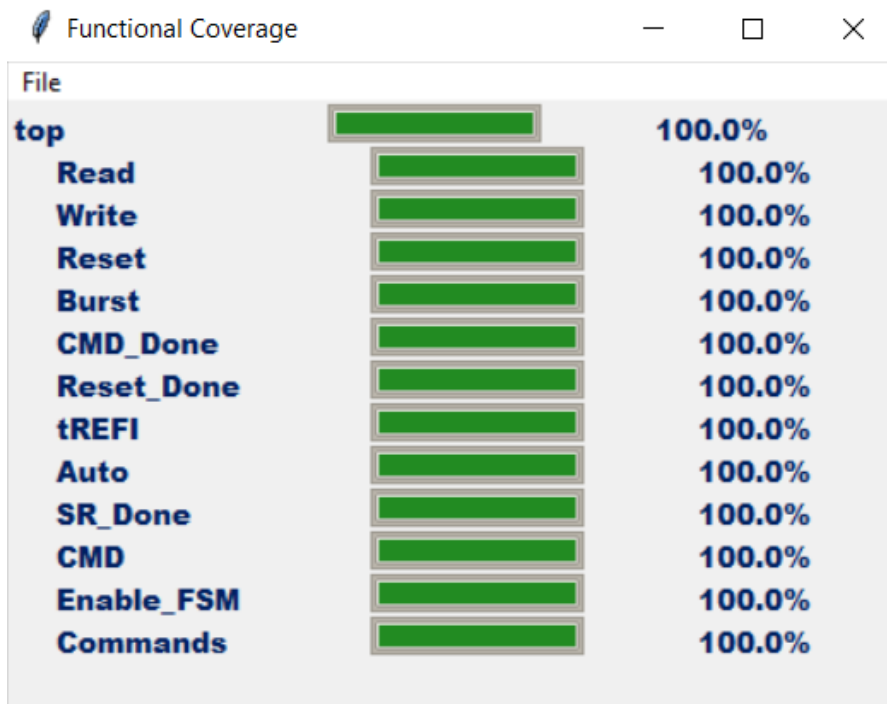


Figure 42: Command decoder Functional Coverage from Coverage Viewer.

DDR5 SDRAM Memory Controller Design and Verification

4.3.5 Code Coverage Results

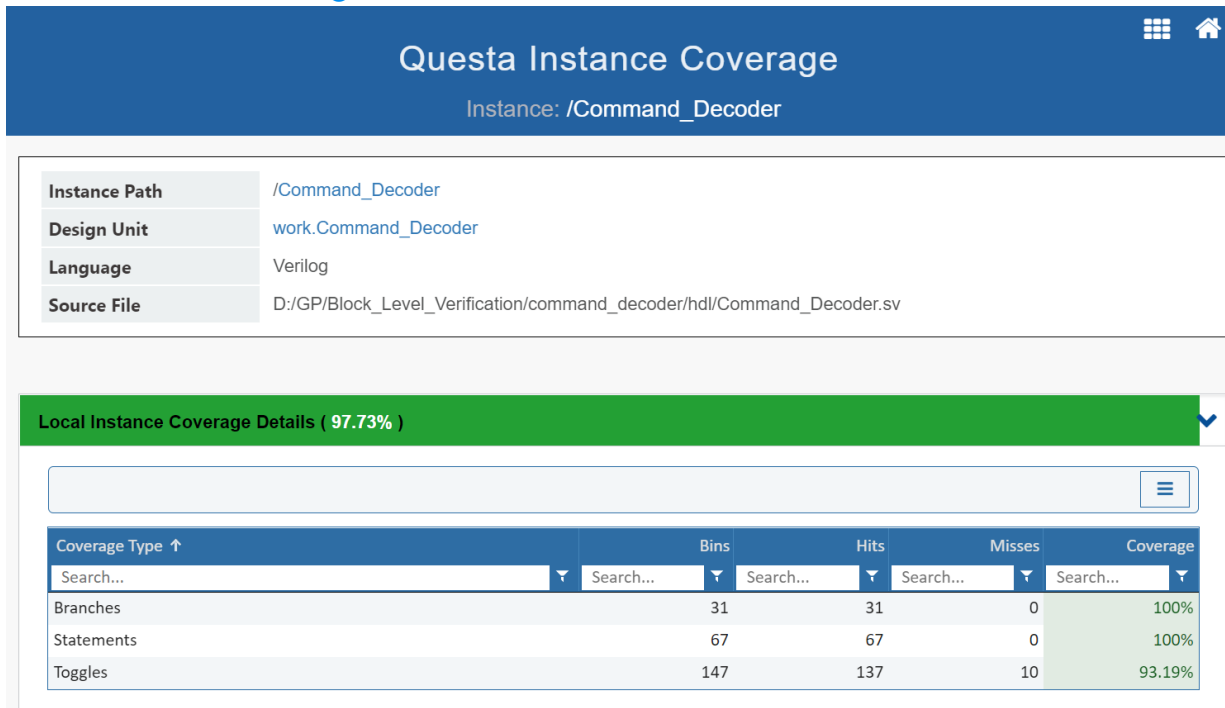


Figure 43: Command decoder Code Coverage Summary from Questasim.

4.4 COMMAND FINITE STATE MACHINE

CMD FSM is responsible for **handling the sequence of operation that should be done in order to execute write or read operations** with or without Auto precharge in same or different bank group address as implemented in 2.5.6 as specified in JESD79-5 section 3.1 and providing the appropriate status signal to SDRAM as specified in JESD79-5 section 4.1 Table 241.

4.4.1 Functional Coverage Plan

The important features that should be covered to ensure the correctness of the Command FSM functionality are the following:

1. All types of write command (write, write Burst, write with AP, write burst with AP).
2. All types of read command (read, read Burst, read with AP, read burst with AP).
3. Two consecutive writes (write, write with AP) in same bank group.
4. Two consecutive writes (write, write with AP) in different bank group.
5. Two consecutive reads (read, read with AP) in same bank group.
6. Two consecutive reads (read, read with AP) in different bank group.
7. Read after write in same bank group.
8. Read after write in different bank group.
9. Write after read in same bank group.
10. Write after read in different bank group.





DDR5 SDRAM Memory Controller Design and Verification

4.4.2 Test Cases







Table 45: Test cases of command FSM.

Test Item	Test Case	Expected Result	Covered	Bug Free
Write	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '001'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A cycles) outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. 	✓	✓
Write Burst	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '100'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A-8) cycles outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. 	✓	✓
Write with AP	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '011'. ➤ First_Command signal is asserted high. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. 	✓	✓



DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A) cycles outputs. ➤ WRITE_DONE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs 		
Write Burst with AP	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '111'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A-8) cycles outputs. ➤ WRITE_DONE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs 		
Read	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '000'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A Cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs 		





DDR5 SDRAM Memory Controller Design and Verification

<p>Read Burst</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '100'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A-8) cycles outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs 		
<p>Read with AP</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '010'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs 		
<p>Read Burst with AP</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '110'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs 		



DDR5 SDRAM Memory Controller Design and Verification

		<ul style="list-style-type: none"> ➤ WAIT_READ_DONE (A-8) cycles outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs 		
<p>write after write in same Row address</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '001'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted High. ➤ CMD signal is asserted '011'. ➤ First_Command signal is asserted Low. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A cycles) outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A cycles) outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. 		

DDR5 SDRAM Memory Controller Design and Verification

<p>Write after write in different Row address</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '001'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted low. ➤ CMD signal is asserted '011'. ➤ First_Command signal is asserted Low. ➤ all counter flags are asserted High. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A) cycles outputs. ➤ WRITE_DONE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A) cycles) outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. 		
<p>read after read in same Row address</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '000'. ➤ First_Command signal is asserted high. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs 		



DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted High. ➤ CMD signal is asserted '010'. ➤ First_Command signal is asserted Low. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A Cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A Cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs 		
<p style="text-align: center;">read after read in different Row address</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '000'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted low. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs 		



DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ CMD signal is asserted '010'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted Low. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs 		
Read after write in same Row address	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '001'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted High. ➤ CMD signal is asserted '000'. ➤ First_Command signal is asserted Low. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A cycles) outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A Cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs 	✓	✓
Read after write in different Row address.	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. 	<ul style="list-style-type: none"> ➤ IDLE outputs. ➤ WAIT_ACT outputs. ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. 	✓	✓

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ CMD signal is asserted '001'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted low. ➤ CMD signal is asserted '000'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted Low. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A) cycles outputs. ➤ WRITE_DONE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A) cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs 		
<p>Write after read in same Row address.</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '000'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A) Cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs 		

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted High. ➤ CMD signal is asserted '001'. ➤ First_Command signal is asserted Low. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. ➤ WAIT_WRITE_DONE (A cycles) outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. 		
<p>Write after read in different Row address</p>	<ul style="list-style-type: none"> ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted Low. ➤ CMD signal is asserted '000'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted high. ➤ Ctrl_ADD signal is asserted a certain value. ➤ Wait until CMD_Done is asserted high. ➤ CMD_FSM_Enable signal is asserted high. ➤ Same_Bank_Group signal is asserted low. ➤ CMD signal is asserted '001'. ➤ First_Command signal is asserted high. ➤ all counter flags are asserted Low. 	<ul style="list-style-type: none"> ➤ IDLE outputs ➤ WAIT_ACT outputs ➤ ACT_CYCLE1 outputs ➤ ACT_CYCLE2 outputs ➤ BANK_ACTIVE outputs ➤ WAIT_READ outputs ➤ READING_CTCLE1 outputs ➤ READING_CYCLE2 outputs ➤ WAIT_READ_LATENCY outputs ➤ READING_DATA outputs ➤ WAIT_READ_DONE (A cycles) outputs ➤ READ_DONE outputs ➤ BANK_ACTIVE outputs ➤ PRECHARGE outputs ➤ WAIT_tRP outputs ➤ IDLE outputs ➤ ACT_CYCLE1 outputs. ➤ ACT_CYCLE2 outputs. ➤ BANK_ACTIVE outputs. ➤ WAIT_WRITE outputs. ➤ WRITING_CTCLE1 outputs. ➤ WRITING_CYCLE2 outputs. ➤ WAIT_WRITE_LATENCY outputs. ➤ WRITING_DATA outputs. 		

DDR5 SDRAM Memory Controller Design and Verification

	<ul style="list-style-type: none"> ➤ Ctrl_ADD signal is asserted a certain value. 	<ul style="list-style-type: none"> ➤ WAIT_WRITE_DONE (A cycles) outputs. ➤ WRITE_DONE outputs. ➤ BANK_ACTIVE outputs. 		
--	--	---	--	--

Note: In consecutive operations, the red colored statements represent the operations of the first command, while the blue colored statements represent the operations of the second command.

4.4.3 Reported Bugs



Bug #1:

CMD FSM gets stuck at wait_write_done state while executing write operation, and at wait_read_done state while executing read operation as well.

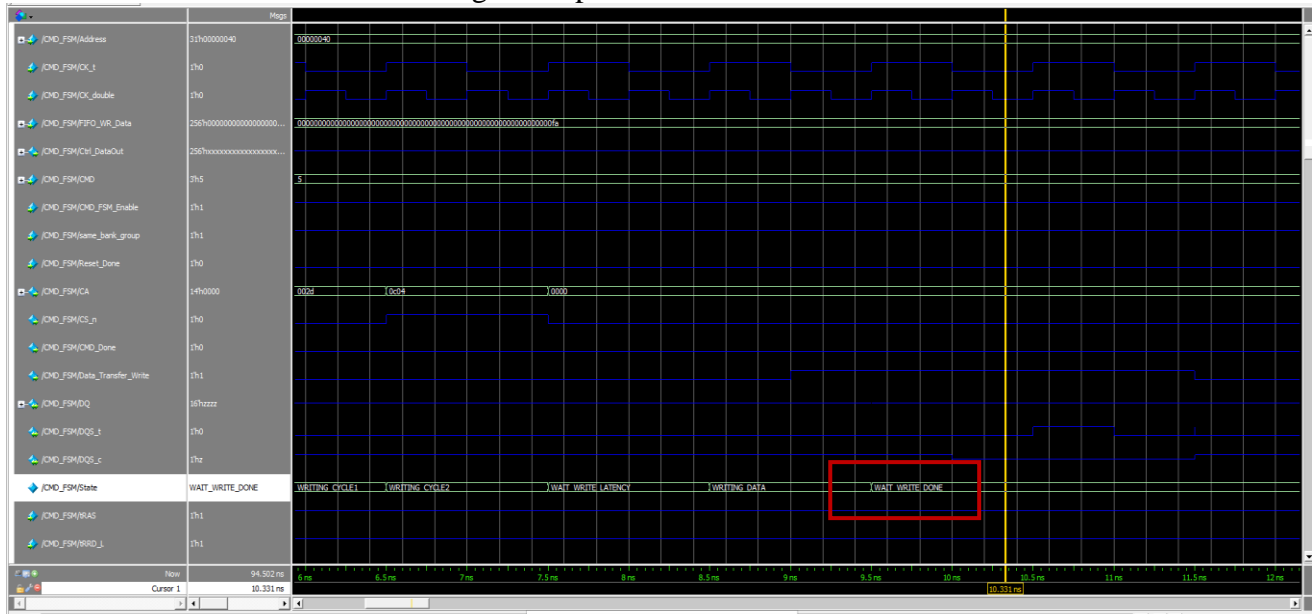


Figure 44: Single write operation stuck at wait write done state.

DDR5 SDRAM Memory Controller Design and Verification



Modification:

After tracing signals, **write_Done signal** which is the status signal that FSM determine the next state based on it, **was found to be asserted high and returned low before the positive edge of the CK_t**, so CMD FSM couldn't sample it, so it stuck at this state.

As **write_Done signal** comes from a **WR FSM** which instantiated in this block but its operating frequency is **double** the frequency of the CMD FSM, **write_Done signal** is asserted **high for two clock cycles of the double clock** by adding extra state (WR_DONE2) which WR FSM goes to it unconditionally, the same solution was done to **Read_Done signal** in RD FSM.

```
WR_DONE1:
begin
    Write_Done = 1'b1;
end
WR_DONE2:
begin
    Write_Done = 1'b1;
end
default:
begin
    Write_Done = 1'b0;
end
endcase

end: output_logic
```

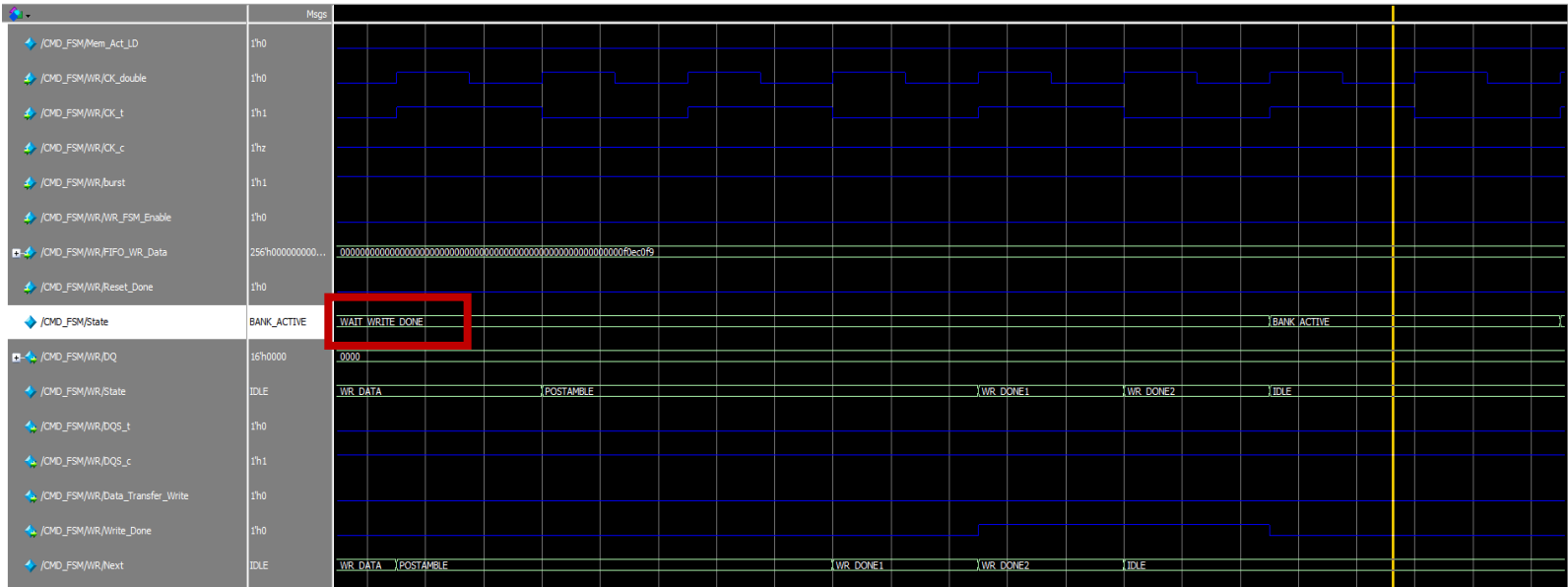


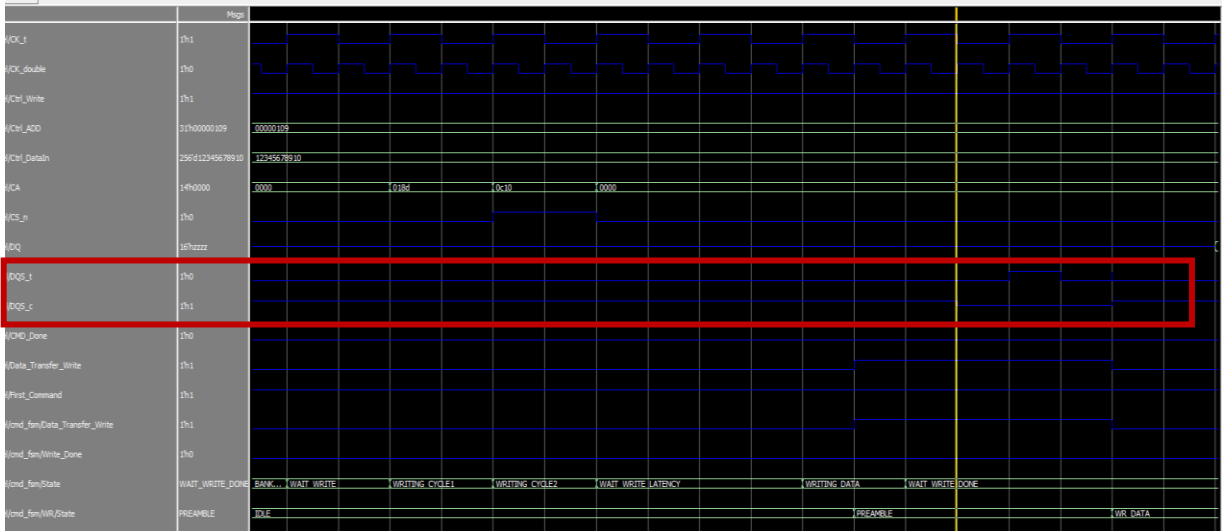
Figure 45: Single write operation after modifying CMD FSM.

DDR5 SDRAM Memory Controller Design and Verification



Bug #2: DQS, DQT preambles aren't working properly.

Figure 46: DQS, DQT preambles aren't working properly.



Modification: The always block which generates DQS, is changed.

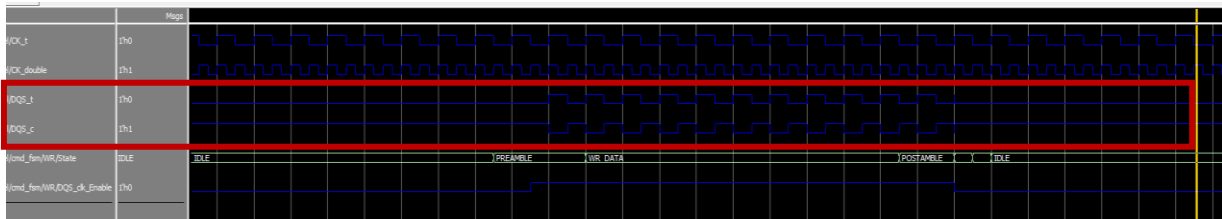


Figure 47: DQS, DQT preambles are working properly after modification.

DDR5 SDRAM Memory Controller Design and Verification



Bug #3: CMD FSM gets stuck at wait_tRP state.

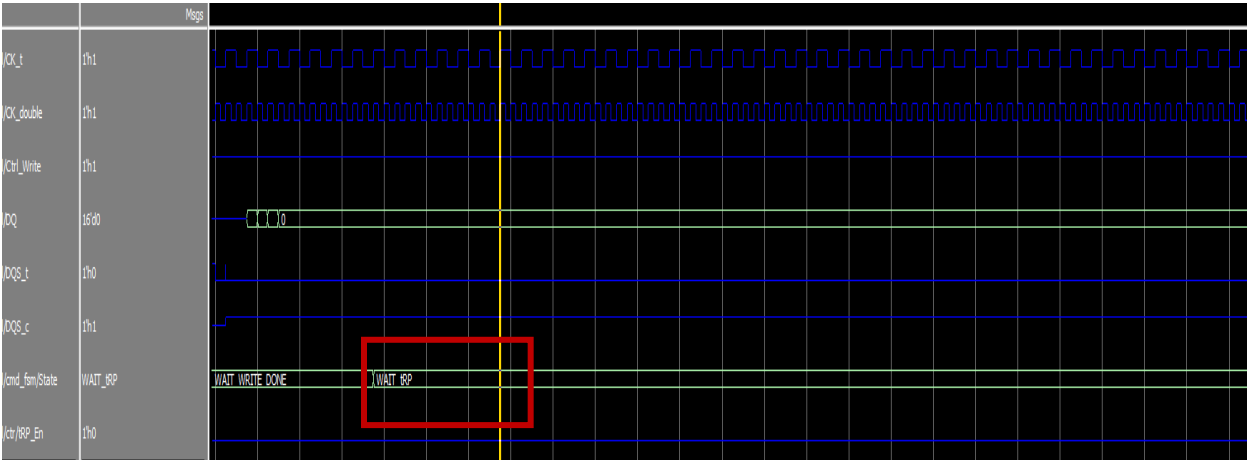


Figure 48: CMD FSM gets stuck at wait_tRP state.



Modification: it was a trivial error in the next state decoder, as next state was wait_tRP at the two branches of the condition on tRP signal.

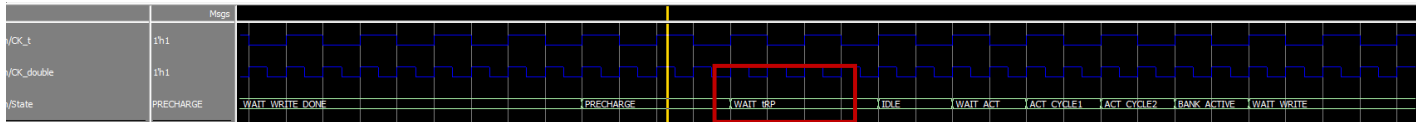


Figure 49: CMD FSM works properly after modification.

DDR5 SDRAM Memory Controller Design and Verification

4.4.4 Functional Coverage Results

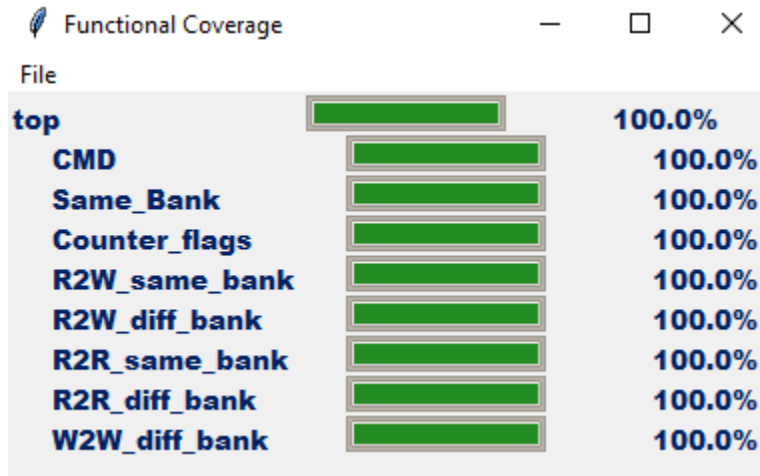


Figure 50: Command FSM Functional Coverage from Coverage Viewer.

4.4.5 Code Coverage Results

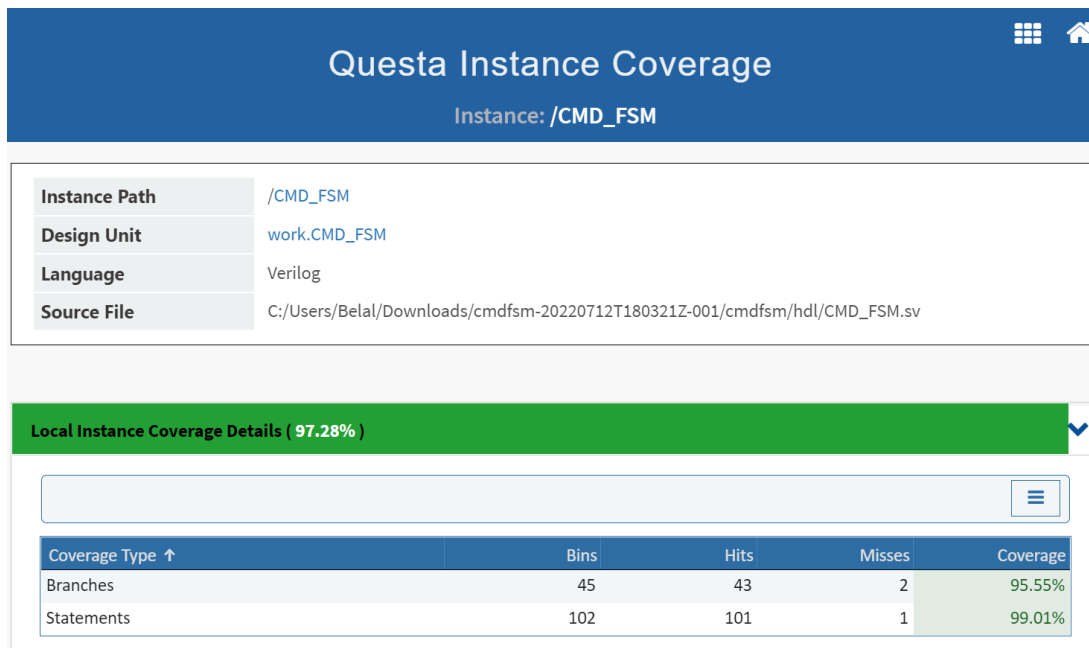


Figure 51: Command FSM Code Coverage Summary from Questasim.

DDR5 SDRAM Memory Controller Design and Verification

4.5 SELF_REFRESH FINITE STATE MACHINE

One of the main limitations of dynamic CMOS circuits such as SDRAMs is the signal integrity issues, so it needs refreshing every certain period to defeat signal integrity issues affected by reading from the cell or leakage current by adjusting and updating its internal average periodic refresh interval, as needed, based on its own temperature sensor (does not require any external control), after the interval time passed, tREF counter will be asserted high and the command decoder will assert SR_FSM_Enable high to enable only the SR_FSM, which is responsible for **doing the sequence of operations** which specified in in JESD79-5 section 4.6, that should be done **in order to complete self-Refreshing**.

4.5.1 Functional Coverage Plan

The important features that should be covered to ensure the correctness of the Self_Refresh FSM functionality are the following:

1. Cover point to cover SR_FSM_Enable.
2. Cover Points to Cover Counter_Flags to grantee that FSM is behaving correctly in case of they are high and low.

4.5.2 Test Cases

Table 46: Test cases of Self_Refresh FSM.

Test Item	Test Case	Expected Result	Covered	Bug Free
Self_Refresh	<ul style="list-style-type: none"> ➤ SR_FSM_Enable signal is asserted high. ➤ all counter flags are asserted high. 	<ul style="list-style-type: none"> ➤ The sequence of CA and CS_n as shown in Figure 55 specified in JESD79-5. 	✓	✓

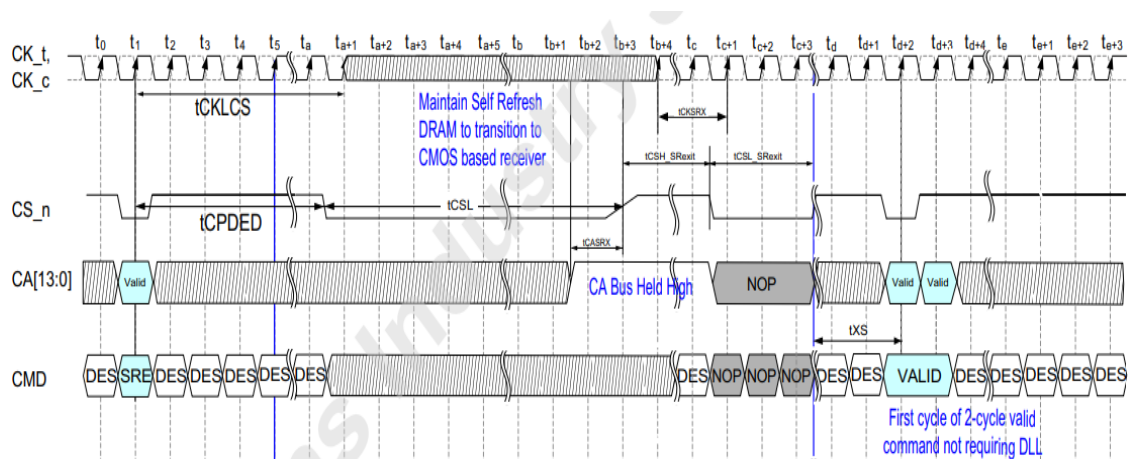


Figure 52: Self_Refresh sequence as specified in JESD79-5 [5].

DDR5 SDRAM Memory Controller Design and Verification

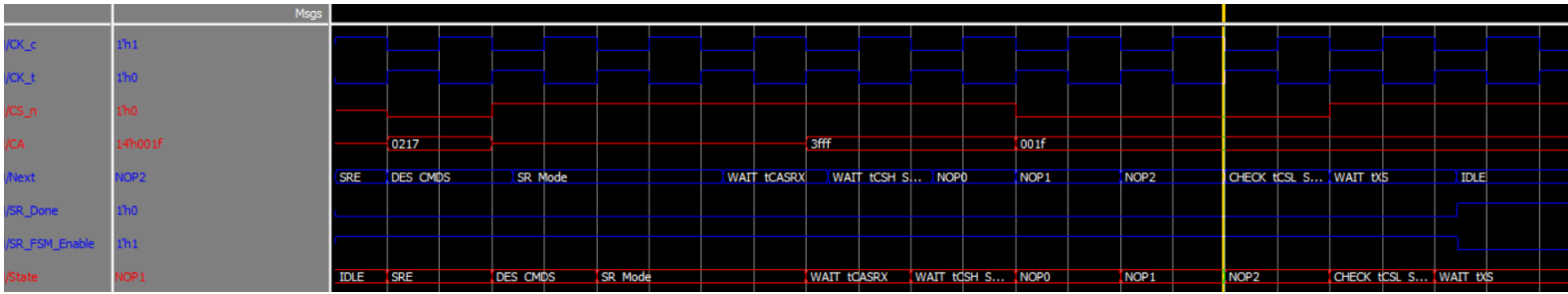


Figure 53: implemented Self_Refresh waveform.

4.5.3 Reported Bugs

Free of Bugs 

4.5.4 Functional Coverage Results

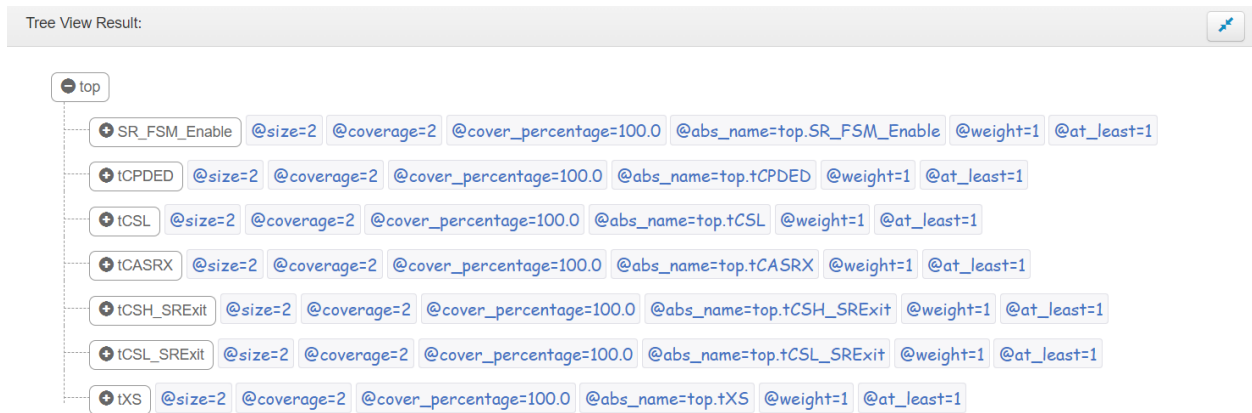


Figure 54: Self-Refresh FSM Functional Coverage XML Report.

DDR5 SDRAM Memory Controller Design and Verification

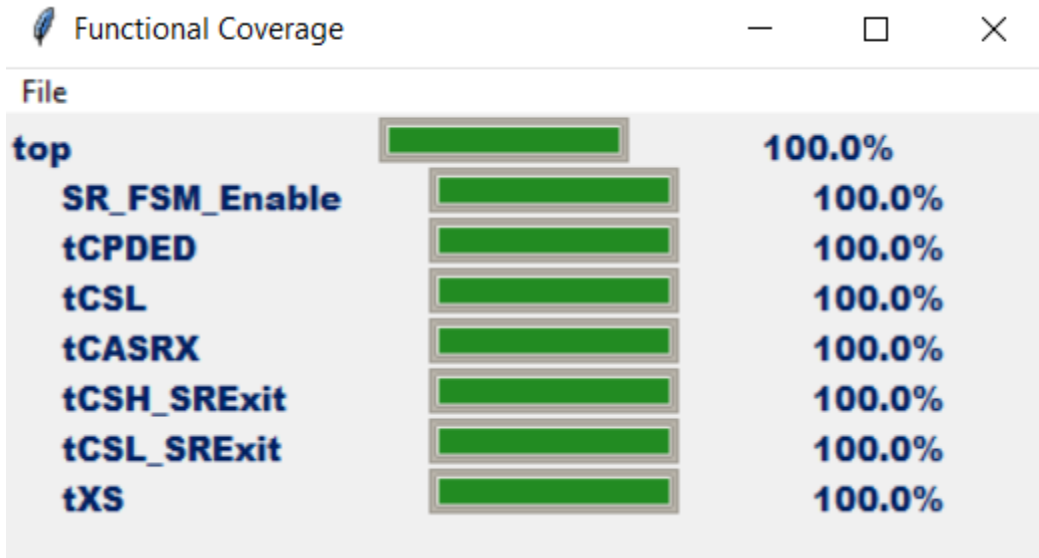


Figure 55: Self-Refresh FSM Functional Coverage from Coverage Viewer.

4.5.6 Code Coverage Results

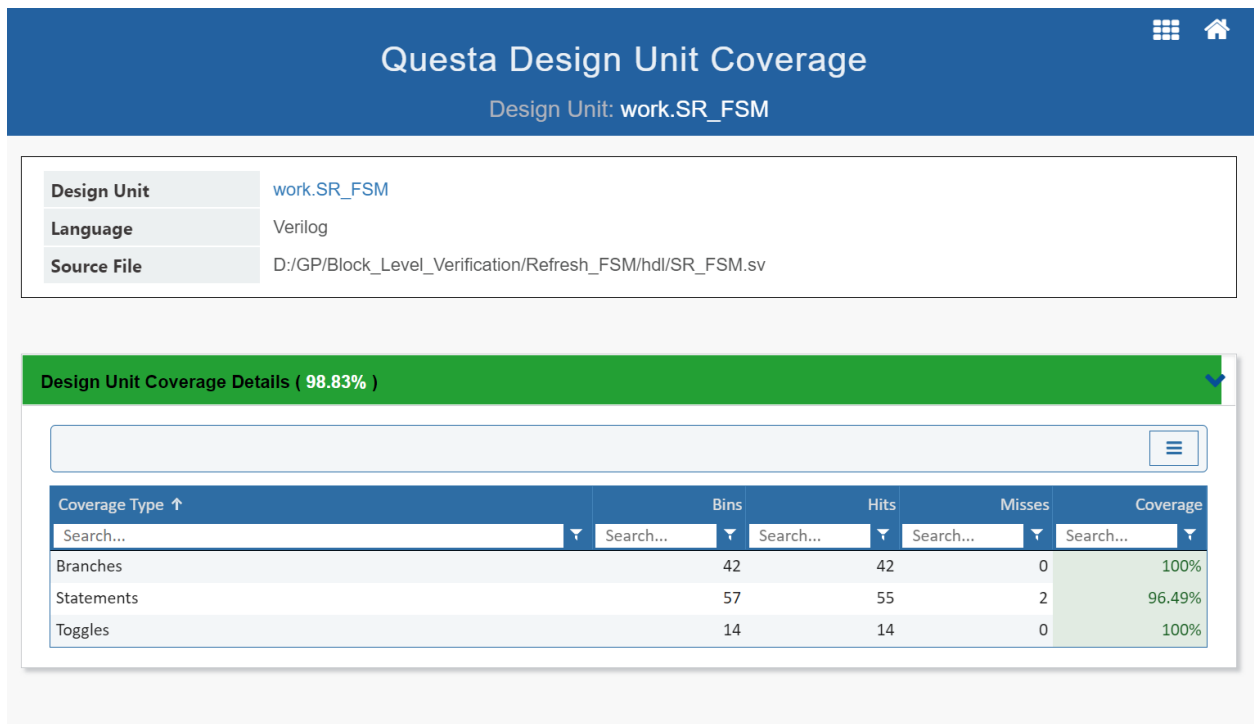


Figure 56: Self-Refresh FSM Code Coverage Summary from Questasim.

DDR5 SDRAM Memory Controller Design and Verification

4.6 INITIALIZATION FINITE STATE MACHINE

Initialization FSM is responsible for doing Set of Sequences as specified in JESD79-5 section 3.3 that should be done in order to power on in a well-known state, starting from power on and ending with loading the mode registers with the default values.

4.6.1 Functional Coverage Plan

The important features that should be covered to ensure the correctness of the

Initialization FSM functionality are the following:

1. Cover point to cover INIT_FSM_Enable.
2. Cover Points to Cover Conter_Flags to grantee that FSM is behaving correctly in case of they are high and low.

4.6.2 Test Cases

Table 47: Test cases of Initialization FSM.

Test Item	Test Case	Expected Result	Covered	Bug Free
Initialization	<ul style="list-style-type: none"> ➤ INIT_FSM_Enable signal is asserted high. ➤ all counter flags are asserted high. 	<ul style="list-style-type: none"> ➤ The sequence of CA and CS_n as shown in Figure 3 specified in JESD79-5. 	✓	✓

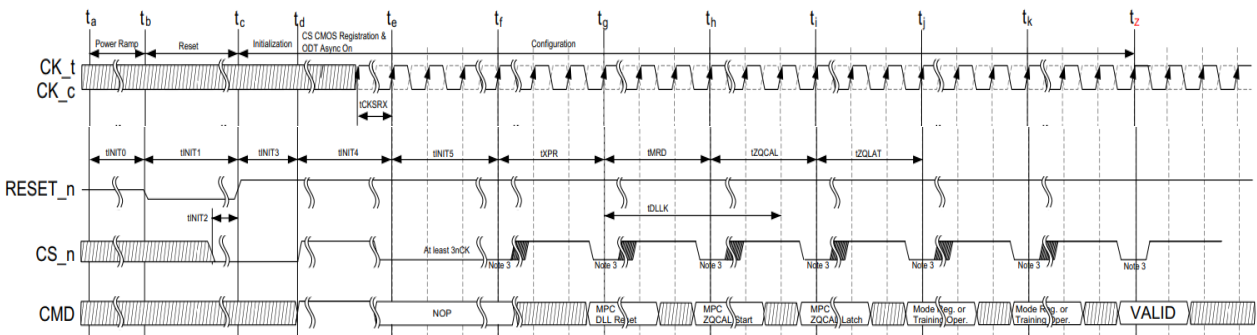


Figure 57: Initialization sequence as specified in JESD79-5 [5].

DDR5 SDRAM Memory Controller Design and Verification

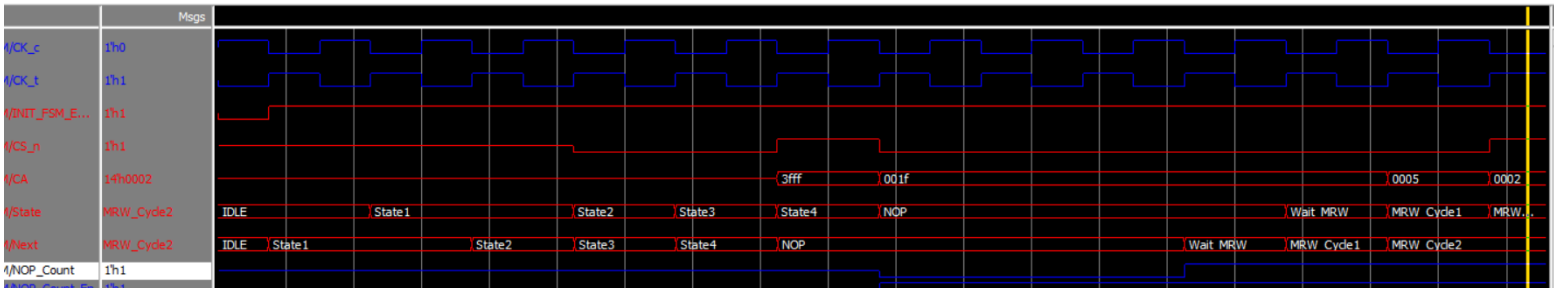


Figure 58: implemented initialization waveform.

4.6.3 Reported Bugs

FREE OF BUGS



4.6.4 Functional Coverage Results

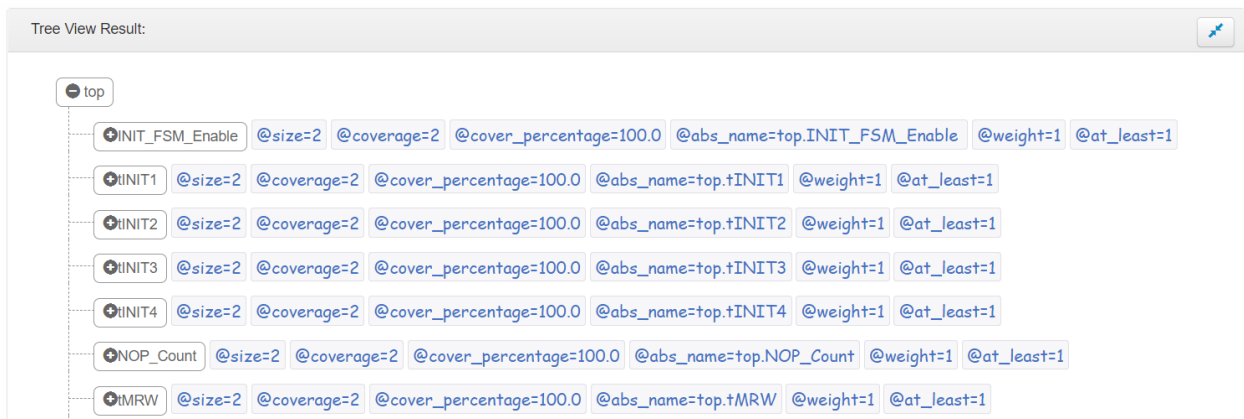


Figure 59: Initialization FSM Functional Coverage XML Report.

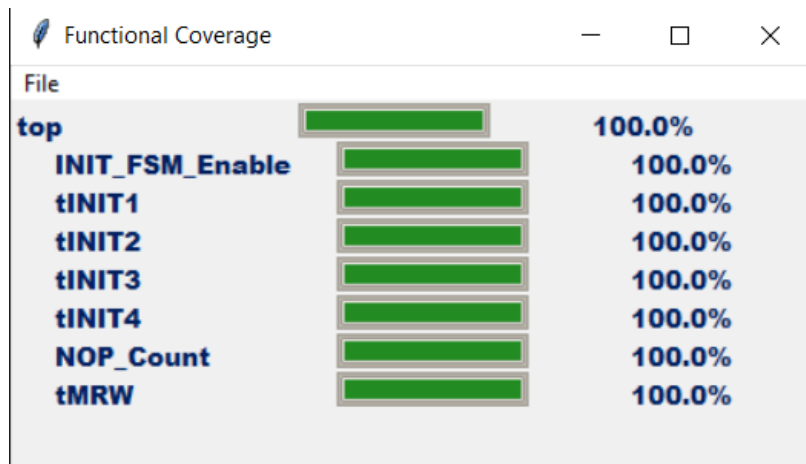


Figure 60: Initialization FSM Functional Coverage from Coverage Viewer.

DDR5 SDRAM Memory Controller Design and Verification

4.6.5 Code Coverage Results

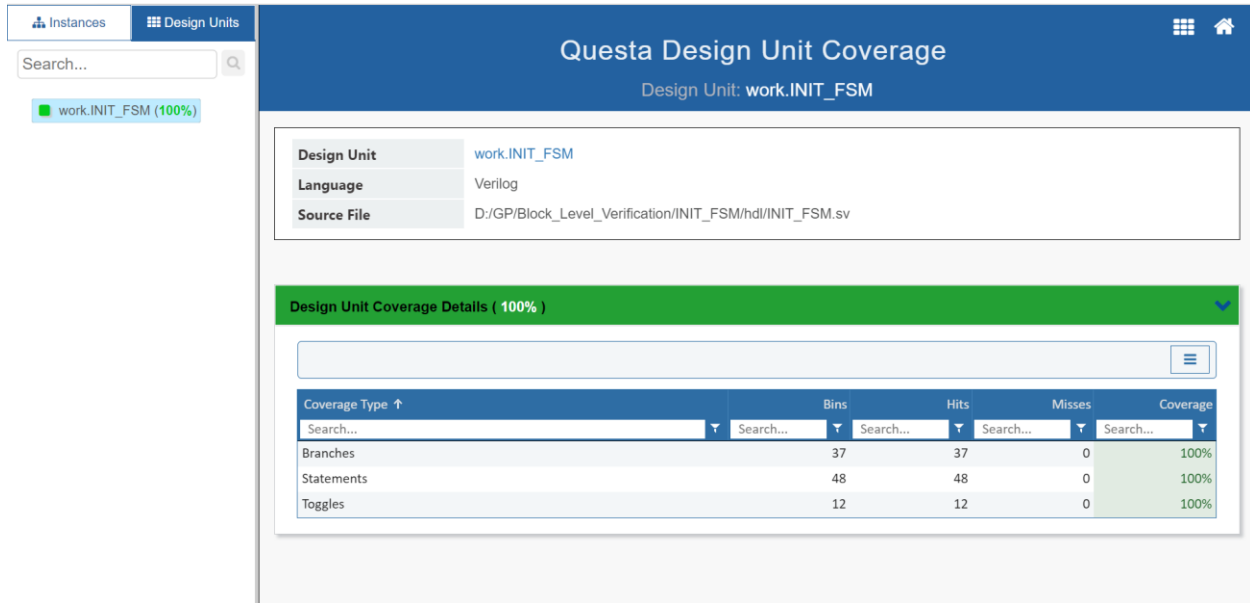


Figure 61: Initialization FSM Code Coverage Summary from Questasim.

4.7 COUNTERS

4.7.1 Functional Coverage Plan

The important features that should be covered to ensure the correctness of the

Counter functionality are the following:



1. Cover point to cover Counter Enable.
2. Cover point to cover Reset.
3. Cross Coverage Points to cover Reset and Enable at the same time.

4.7.2 Test Cases

Table 48: Test cases of counters.

Test Item	Test Case	Expected Result	Covered	Bug Free
Enable	<ul style="list-style-type: none"> ➤ Counter_Enable signal is asserted high. ➤ Counter_Reset signal is asserted Low. 	<ul style="list-style-type: none"> ➤ After a specific number of clock cycles according to value of timing parameter Counter Flag will be asserted High. 	✓	✓

DDR5 SDRAM Memory Controller Design and Verification

Reset	<ul style="list-style-type: none"> ➤ Counter_Enable signal is asserted low. ➤ Counter_Reset signal is asserted High. 	<ul style="list-style-type: none"> ➤ Counter Flag will be asserted low immediately. 		
--------------	--	---	---	---

4.7.3 Reported Bugs

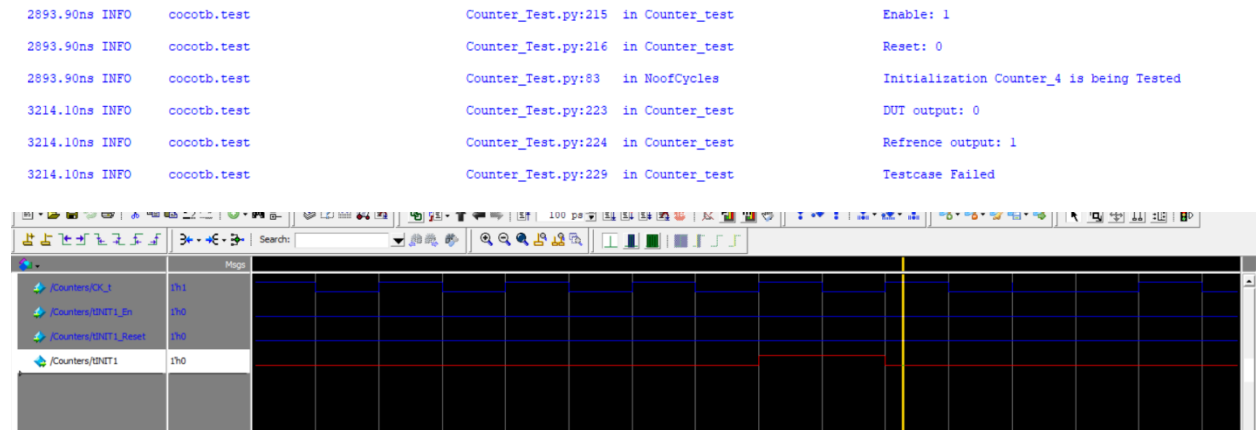


Figure 62: Counter Flag is asserted High for only one Clock Cycle.

4.7.4 Functional Coverage Results

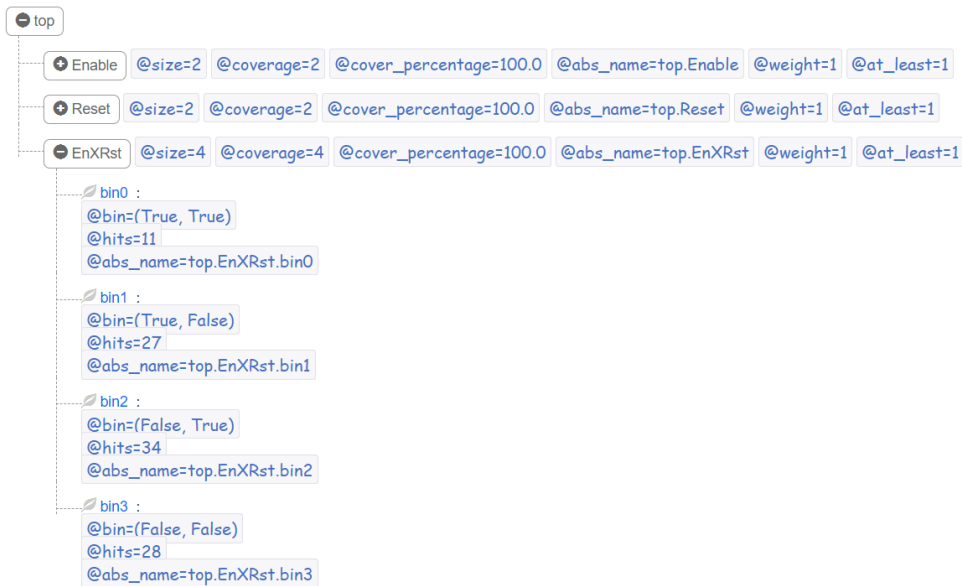


Figure 63: Counters Functional Coverage XML Report.

DDR5 SDRAM Memory Controller Design and Verification

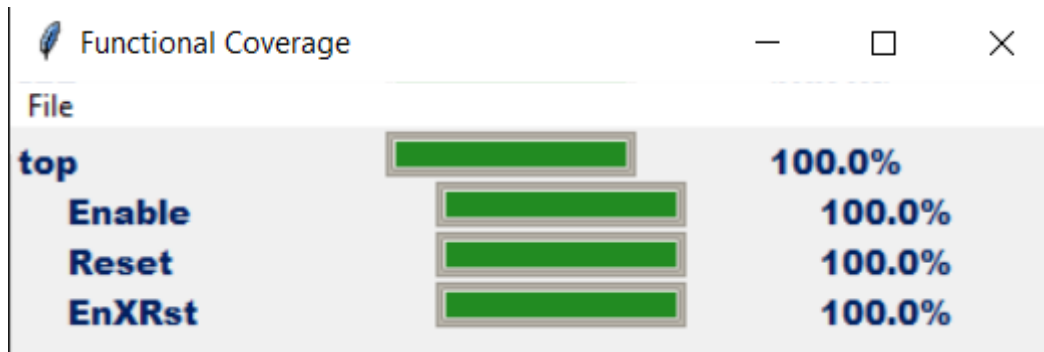


Figure 64: Counters Functional Coverage from Coverage Viewer.

4.7.5 Code Coverage Results

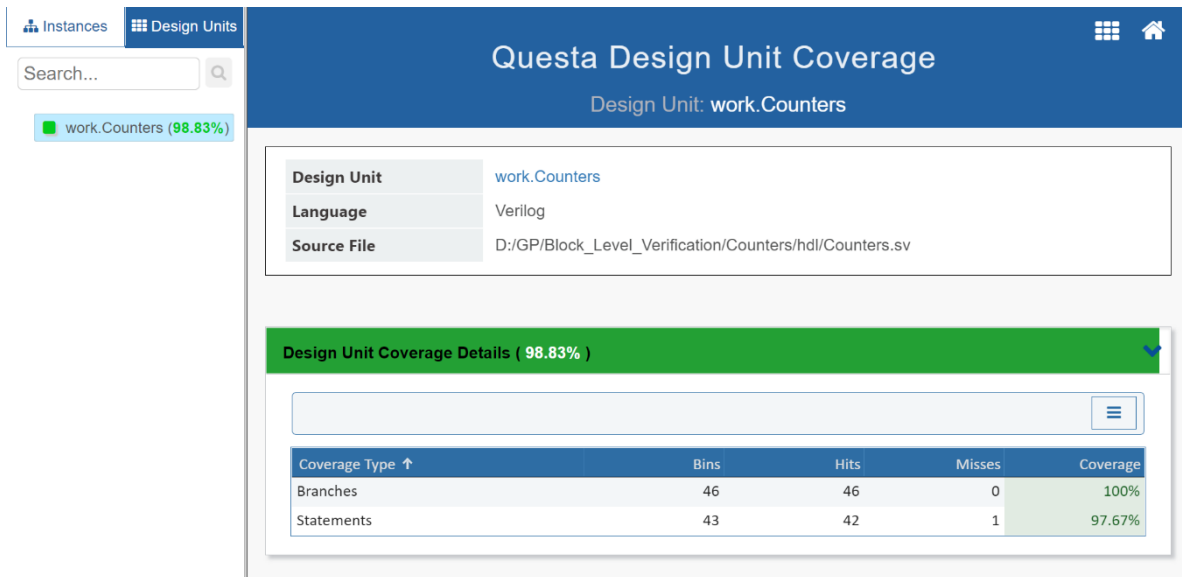


Figure 65: Counters Code Coverage Summary from Questasim.

```
# 76872.70ns INFO cocotb.test Counter_Test.py:83 in N
oofCycles Initialization Counter_4 is being Tested
#
# 76873.70ns INFO cocotb.test Counter_Test.py:208 in C
ounter_test DUT output: 0
#
# 76873.70ns INFO cocotb.test Counter_Test.py:209 in C
ounter_test Refrence output: 0
#
# 76873.70ns INFO cocotb.test Counter_Test.py:211 in C
ounter_test Testcase Passed
#
# 76873.70ns INFO cocotb.test Counter_Test.py:217 in C
ounter_test no of Passed Testcases: 100
#
# 76873.70ns INFO cocotb.test Counter_Test.py:219 in C
ounter_test no of Failed Testcases: 0
#
# 76873.71ns INFO cocotb.regression regression.py:364 in _
score_test Test Passed: Counter_test
#
# 76873.71ns INFO cocotb.regression regression.py:487 in _
```

Figure 66: Counters Test Summary from Questasim.

Chapter 5: UVM VS COCOTB

5.1 INTRODUCTION

In order to compare between UVM verification and COCOTB verification we choose block from our design which is the FIFO and test it with UVM and COCOTB.

5.1.1 UVM verification

A key concept for any modern verification methodology is the layered test bench. Although this process may seem to make the test bench more complex, it actually helps to make the task easier by dividing the code into smaller pieces that can be developed separately. The proposed UVM-based verification architecture for WR_Data_FIFO block is shown in Figure 67 [14].

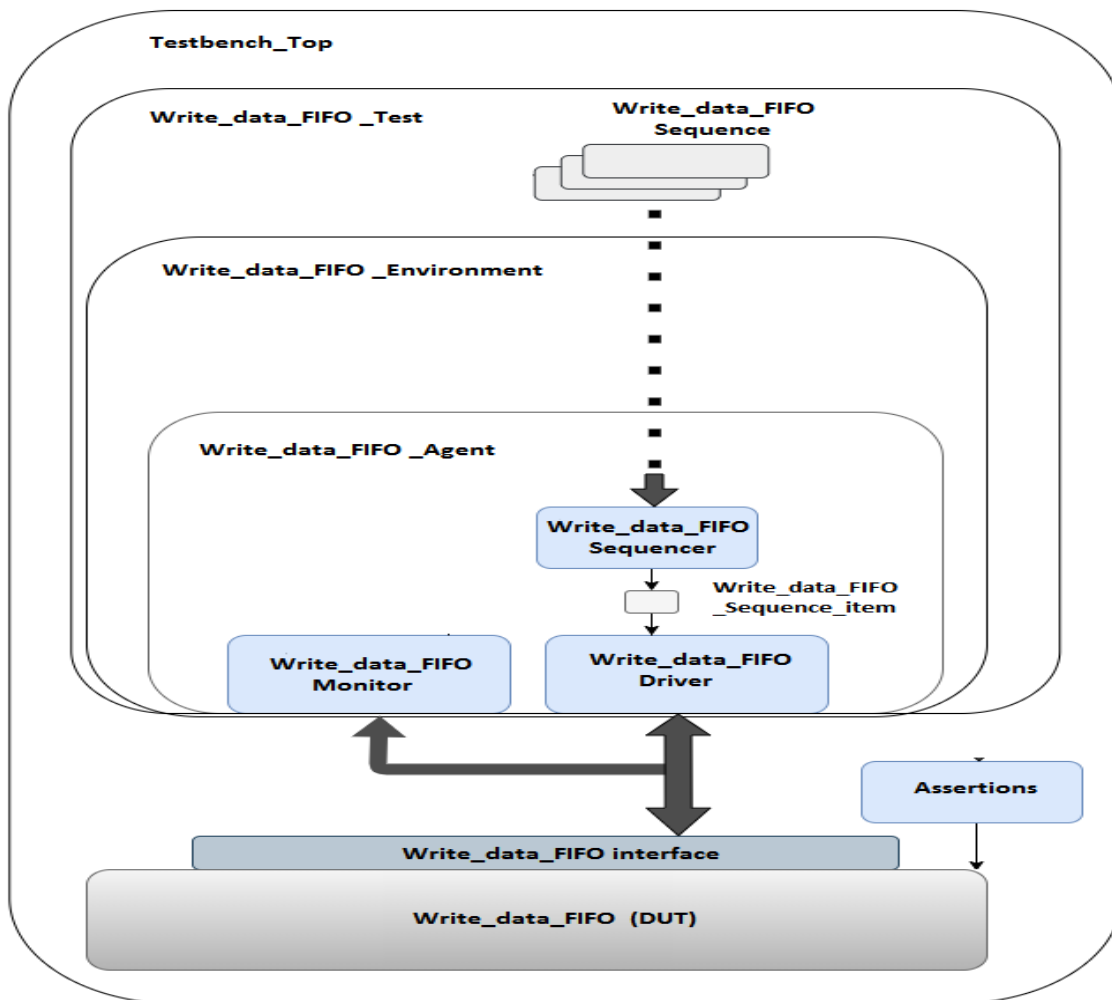


Figure 67: UVM environment for WR_Data_FIFO block [14].

DDR5 SDRAM Memory Controller Design and Verification

5.1.2 UVM verification environment components

UVM verification environment is composed of different components such as [15]:

- **Driver** which drives the DUT's inputs as it runs single commands, such as bus read or write.
- **Monitor** which is driven by the DUT's output as it that takes signal transitions and groups them together into commands.
- **Assertions** which cross the command/signal layer, as they look at individual signals.
- **Sequencer** which takes sequence items from a sequence and passes them to the driver.
- **Functional coverage** measures the progress of all tests in fulfilling the verification plan requirements.

5.2 VERIFICATION PLAN

First step in verification process is to make the verification plan which is derived from the hardware specification and contains a description of what features need to be exercised and the techniques to be used. These steps may include directed or random testing and assertions.

So, we make verification plan for testing FIFO which is to:

- Ensure that we can write and read from FIFO and make sure that it follows first input data is the first output data
- Ensure that full flag is high when FIFO is full and ensure that empty flag is high when FIFO is empty
- Ensure that read, write pointer, full, empty flag are reset when Reset is high.

Then we make testing scenarios and ensure that our tests provide 100% coverage of the entire verification plan.

5.2.1 Testing scenarios:

1. Write 16 data packets until the FIFO is full then read all 16 data packets and compare whether the data is same as what have been written previously.
2. Write and read data randomly for 50 times.
3. Using system Verilog assertions for:
 - Asserting for that Read pointer, write pointer, FIFO counter, full flag and empty flag are now reset when Reset is high.
 - Asserting for that FIFO full flag is high when FIFO has no space to write in.
 - Asserting for writing in a full FIFO and FIFO full flag is high.
 - Asserting for that FIFO empty flag is high when all data have been read.
 - Asserting for trying to read from empty FIFO and FIFO empty flag is high.

DDR5 SDRAM Memory Controller Design and Verification

5.3 RESULTS FROM UVM VERIFICATION

There were some few errors that we have discovered from results of simulation such as:

```
# RD_En is high
# UVM_INFO C:/fifo/FIFO_UVM.sv(328) @ 580: uvm_test_top.f_env.f_scb [Read Data] examdata: 3a FIFO_WR_Data: 0 empty: 0
# ----- Fail! -----
# ----- Check empty -----
```

Figure 68: 1st example of error from simulation.

The solution of this error was editing the size of Read Pointer to be 4 bits as FIFO depth is 16.

```
# WR_En is high
# UVM_INFO C:/fifo/FIFO_UVM.sv(323) @ 310: uvm_test_top.f_env.f_scb [write Data] WR_En: 1 RD_En: 0 Ctrl_DataIn: b5 full: 0
# 310: Assertion Failed: The design failed the fifo not full condition.
# UVM_INFO C:/fifo/FIFO_UVM.sv(285) @ 320: uvm_test_top.f_agt.f_seqr@@f_seq [f_sequence] ***** Generate 16 Read REQs *****
# 320: Assertion Failed: The design failed the fifo not full condition.
```

Figure 69: 2nd example of error from simulation.

The solution of this error was editing the condition for FIFO full flag to be high only if FIFO counter equals 16. Etc.

We solved all design errors and the final UVM Report Summary and Coverage is shown in Figure 70.

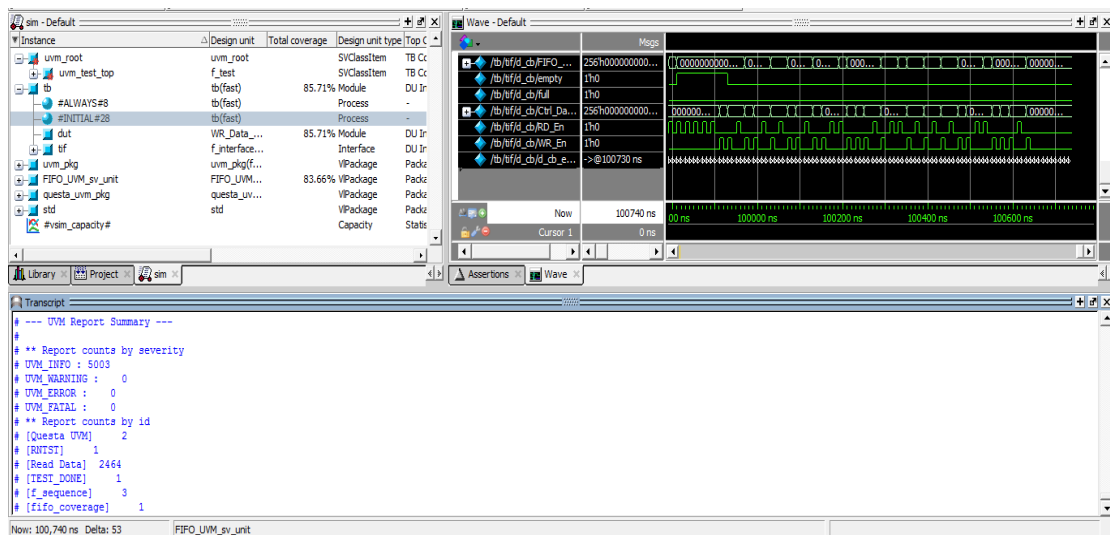


Figure 70: UVM Report Summary and Coverage

DDR5 SDRAM Memory Controller Design and Verification

Also, all assertions are passed as shown in Figure 71. And the assertions coverage is shown in Figure 72.

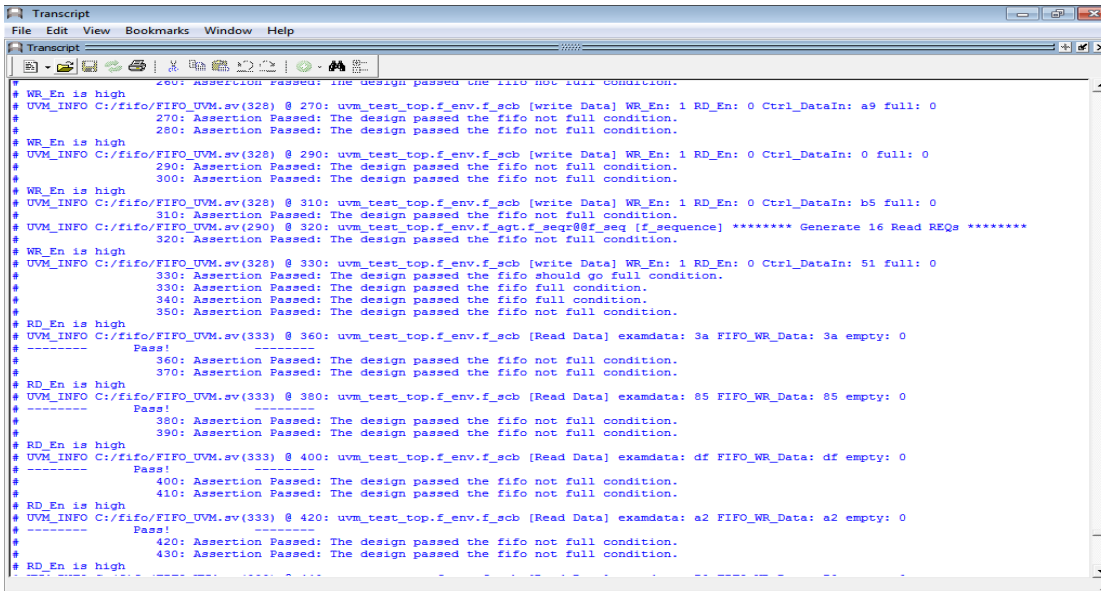


Figure 71: Assertions and UVM results.

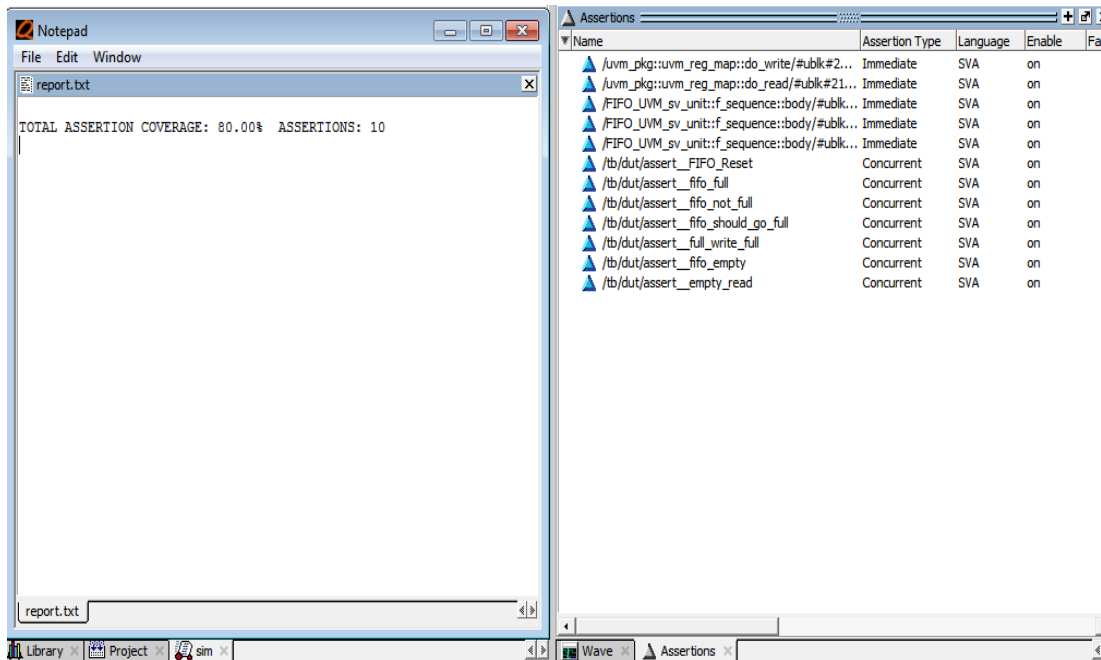


Figure 72: The assertions coverage.

DDR5 SDRAM Memory Controller Design and Verification

5.4 RESULTS FROM COCOTB VERIFICATION

We did verification for WR_Data_FIFO block also with COCOTB and we get the same errors results as that we got from UVM verification as shown in Figure and the results are shown in Figure 73.

```
0.65ns INFO Data written to fifo: C4776B1B
0.85ns INFO Data written to fifo: 71FEB625
1.05ns INFO Data written to fifo: 4B62C794B
1.25ns INFO Data written to fifo: 7D562376F
1.45ns INFO Data written to fifo: 58699AC0A
1.65ns INFO Data written to fifo: 6F0B6AB17
1.85ns INFO Data written to fifo: 2AA7827C9
2.05ns INFO Data written to fifo: 5D78CC64E
2.25ns INFO Data written to fifo: 52EC776F8
2.45ns INFO Data written to fifo: A941C580
2.65ns INFO Data written to fifo: 78AB95DAD
2.85ns INFO Data written to fifo: 42CE2789F
3.05ns INFO Data written to fifo: 5BD0123C9
3.25ns INFO fifo_test failed

Traceback (most recent call last):
  File "C:\Users\Belal\Downloads\fifo\final\tests\fifo_test.py", line 114, in fifo_test
    assert(data == fifo_model.pop())
AssertionError: assert 0 == 25580231248
+ where 25580231248 = <built-in method pop of collections.deque object at 0x000000006D74A00>()
+ where <built-in method pop of collections.deque object at 0x000000006D74A00> = deque([24645805001, 17932908703, ...])

3.25ns INFO *****
** TEST STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
*****
** fifo_test.fifo_test FAIL 3.25 0.01 259.87 **
*****
```

Figure 73: Result of errors from COCOTB.

After solving all design errors, the final COCOTB Report which shows that tests are passed is shown in Figure 74.

```
# 976.65ns INFO Data NOT written, fifo FULL!
# 976.85ns INFO Data read from fifo: F9CAA964
# 977.05ns INFO Data written to fifo: 3D7191419
# 977.25ns INFO Data NOT written, fifo FULL!
# 977.45ns INFO Data NOT written, fifo FULL!
# 977.65ns INFO Data read from fifo: 28654EB9D
# 977.85ns INFO Data read from fifo: 4385B1C38
# 978.05ns INFO Data written to fifo: 4FC1FD279
# 978.25ns INFO Data read from fifo: 7C8A43FD4
# 978.45ns INFO Data written to fifo: 280F2FBF6
# 978.65ns INFO Data read from fifo: 24131BEAF
# 978.85ns INFO Data written to fifo: 688CF0D11
# 979.05ns INFO Data read from fifo: 7540373D9
# 979.25ns INFO Data written to fifo: 793DE148E
# 979.45ns INFO Data written to fifo: 196169EA2
# 979.65ns INFO Data NOT written, fifo FULL!
# 979.85ns INFO Data NOT written, fifo FULL!
# 980.05ns INFO Data read from fifo: 4876CC345
# 980.25ns INFO Data written to fifo: 4A4728542
```

DDR5 SDRAM Memory Controller Design and Verification

```
1000.05ns INFO top : <cocotb_coverage.coverage.CoverItem object at 0x0000000006c379d0>, coverage=8, size=8
1000.05ns INFO top.fifo_full : <cocotb_coverage.coverage.coverPoint object at 0x0000000006c37b20>, coverage=2, size=2
1000.05ns INFO     BIN True : 2138
1000.05ns INFO     BIN False : 2862
1000.05ns INFO top.rw : <cocotb_coverage.coverage.coverPoint object at 0x0000000006c37910>, coverage=2, size=2
1000.05ns INFO     BIN True : 1168
1000.05ns INFO     BIN False : 3832
1000.05ns INFO top.rwxfull : <cocotb_coverage.coverage.coverCross object at 0x0000000006c37b50>, coverage=4, size=4
1000.05ns INFO     BIN (True, True) : 498
1000.05ns INFO     BIN (True, False) : 670
1000.05ns INFO     BIN (False, True) : 1640
1000.05ns INFO     BIN (False, False) : 2192
1000.05ns INFO fifo_test passed
1000.05ns INFO *****
** TEST                STATUS SIM TIME (ns) REAL TIME (s) RATIO (ns/s) **
*****
** fifo_test.fifo_test    PASS          1000.05          3.16          316.33 **
*****
** TESTS=1 PASS=1 FAIL=0 SKIP=0          1000.05          3.18          314.78 **
*****
```

Figure 74: COCOTB verification results

Functional coverage percentage is 100% as shown in Figure 75 and Code coverage percentage is 99.09% as shown in Figure 76.

```
▼<top abs_name="top" size="8" coverage="8" cover_percentage="100.0">
  ▼<rw size="2" coverage="2" cover_percentage="100.0" abs_name="top.rw" weight="1" at_least="1">
    <bin0 bin="True" hits="1168" abs_name="top.rw.bin0"/>
    <bin1 bin="False" hits="3832" abs_name="top.rw.bin1"/>
  </rw>
  ▼<fifo_full size="2" coverage="2" cover_percentage="100.0" abs_name="top.fifo_full" weight="1"
  at_least="1">
    <bin0 bin="True" hits="2138" abs_name="top.fifo_full.bin0"/>
    <bin1 bin="False" hits="2862" abs_name="top.fifo_full.bin1"/>
  </fifo_full>
  ▼<rwxfull size="4" coverage="4" cover_percentage="100.0" abs_name="top.rwxfull" weight="1"
  at_least="1">
    <bin0 bin="(True, True)" hits="498" abs_name="top.rwxfull.bin0"/>
    <bin1 bin="(True, False)" hits="670" abs_name="top.rwxfull.bin1"/>
    <bin2 bin="(False, True)" hits="1640" abs_name="top.rwxfull.bin2"/>
    <bin3 bin="(False, False)" hits="2192" abs_name="top.rwxfull.bin3"/>
  </rwxfull>
</top>
```

Figure 75: COCOTB Functional coverage report

DDR5 SDRAM Memory Controller Design and Verification

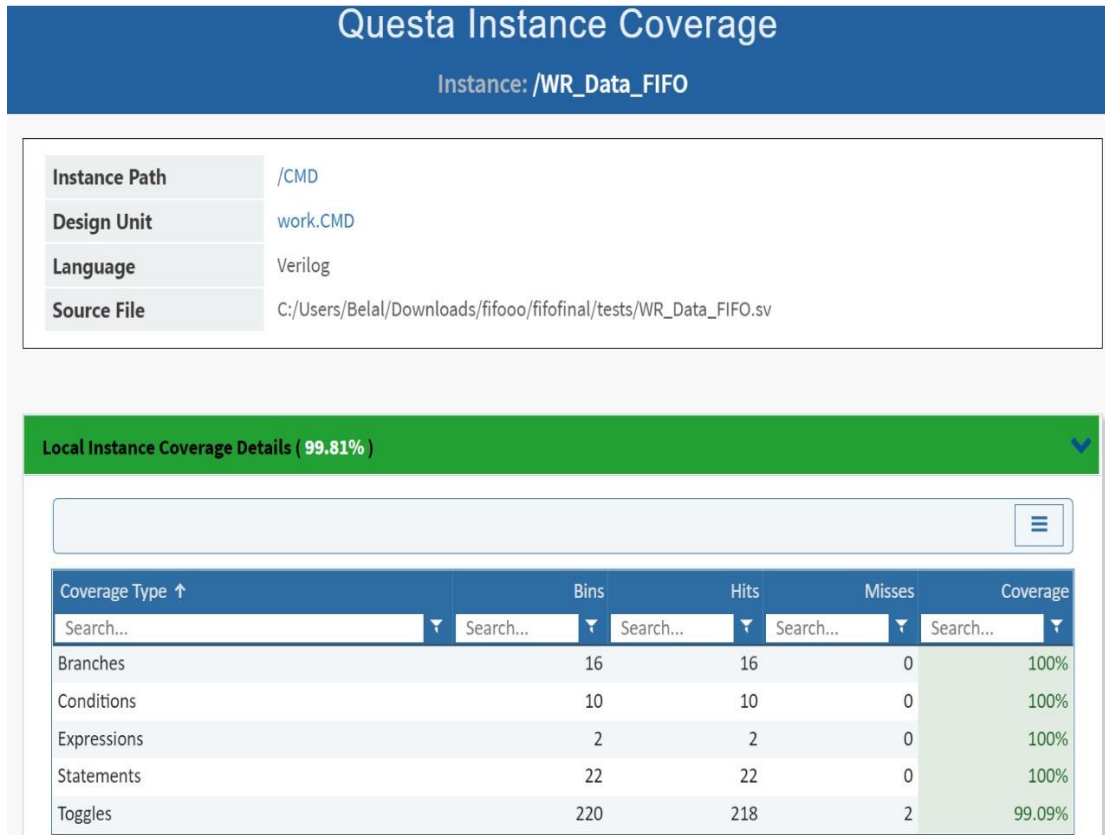


Figure 76: COCOTB Code coverage report

5.5 HISTORY OF VERIFICATION METHODS:

In order to understand why using COCOTB we should first know the History of verification methods. Traditionally, when faced with the task of verifying the directed tests method is used by writing stimulus vectors that exercise the features in the DUT. Then simulating the DUT with these vectors and manually reviewing the resulting log files and waveforms to make sure the design work properly. The problem with the directed tests method is when the design complexity doubles, it takes twice as long to complete or requires twice as many people to implement it.

So, the second method which is Constrained-Random Stimulus is used as constrained-random test bench is now finding bugs faster than the many directed ones as shown in Figure 77 [13]. Another advantage is that directed test finds the bugs you expect to be in the design, whereas a random test can find bugs you never anticipated. When using random stimuli, you need functional coverage to measure verification progress. Furthermore, once you start using automatically generated stimuli, you need an automated way to predict the results generally a scoreboard or reference model. Building the test bench infrastructure, including self-prediction, takes a significant amount of work. A layered test bench helps you control the complexity by breaking the problem into manageable pieces.

DDR5 SDRAM Memory Controller Design and Verification

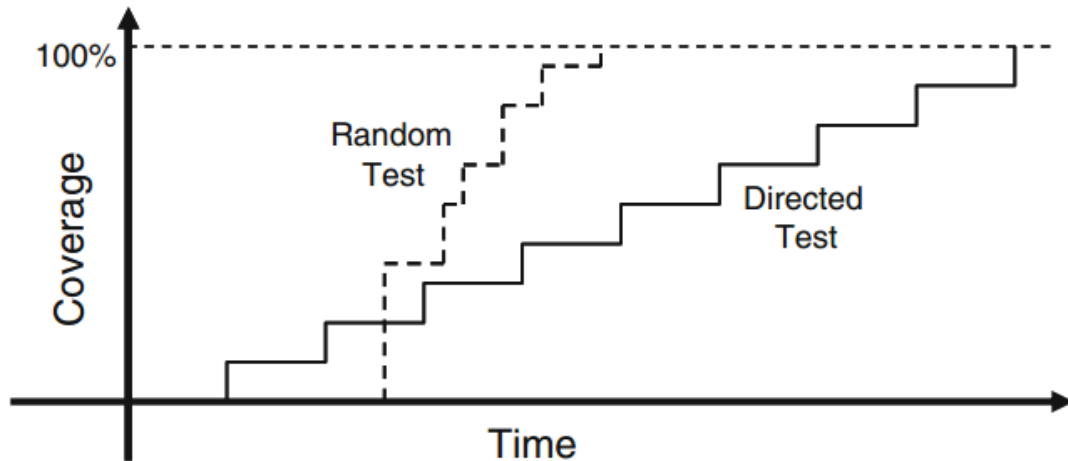


Figure 77: Constrained-random test progress over time vs. directed testing [13].

As higher-level language concepts (like OOP) are useful when writing layered and complex test benches so they added higher level programming features to a hardware description language (System Verilog). So UVM (Universal Verification Methodology) libraries written in System Verilog.

The SV/UVM approach is powerful, but complicated. Since the Verification test benches are software, not hardware problem so the COCOTB's developers tried a different approach which is using a high-level, general-purpose language (Python) for developing test benches.

5.6 TRADEOFFS BETWEEN USING UVM AND COCOTB:

- 1) The SV/UVM approach is powerful, but complicated as system Verilog has around 250 keywords and its reference has around 1300 pages as shown in Figure 78 [16].
- 2) COCOTB's developers, Chris Higgs and Stuart Hodgson, tried a different approach:
 - Keep the hardware description languages for what they're good at—design!
 - Use a high-level, general-purpose language for developing test benches.
 - Object oriented programming is much more natural in general purpose languages!
 - They picked Python as their language of choice:
Python is simple (only 35 keywords) and easy to learn, but very powerful.
 - The Python reference is around 160 pages.
 - Python has a large standard library and a huge ecosystem; lots of existing libraries.
 - Python is well documented and popular: lots of resources online.
 - The Python test bench can read or change the value of any internal signal.
 - COCOTB can be used for post-synthesis simulations too!
 - Tests can call other methods and functions, just like normal Python

DDR5 SDRAM Memory Controller Design and Verification

- 3) Also, now there are developers making the Universal Verification Methodology (UVM) implemented in Python to get advantage of reusable components which is provided by UVM.

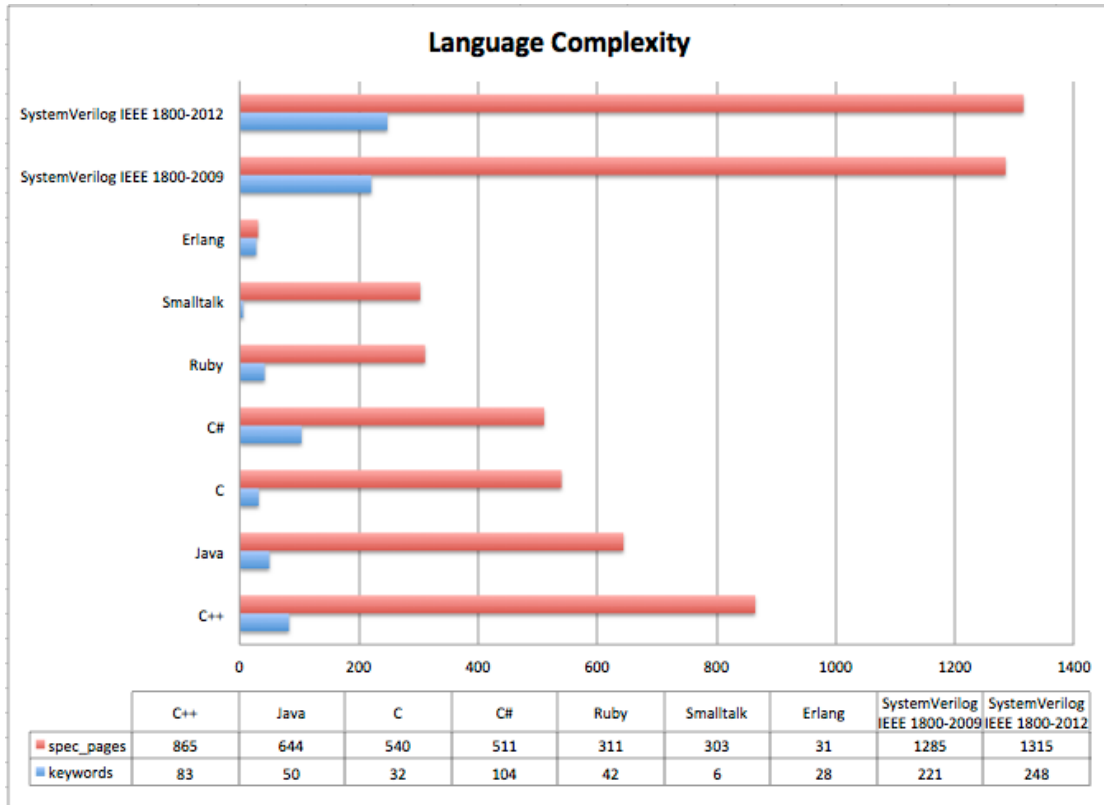


Figure 78: Programming languages Complexity.

5.7 FUTURE OF PYTHON IN VERIFICATION:

Here are two studies from Siemens were made in 2020. first study is about FPGA verification language adoption next twelve months which shows that using of python in verification has increased over the years and it exceeds 20% in usage with respect to other languages as shown in Figure 79 [17].

Second study is about ASIC/IC verification language adoption next twelve months which shows that using of python in verification has increased over the years and it reaches around 30% in usage with respect to other languages as shown in Figure 80 [17].

DDR5 SDRAM Memory Controller Design and Verification

FPGA Verification Language Adoption Next Twelve Months

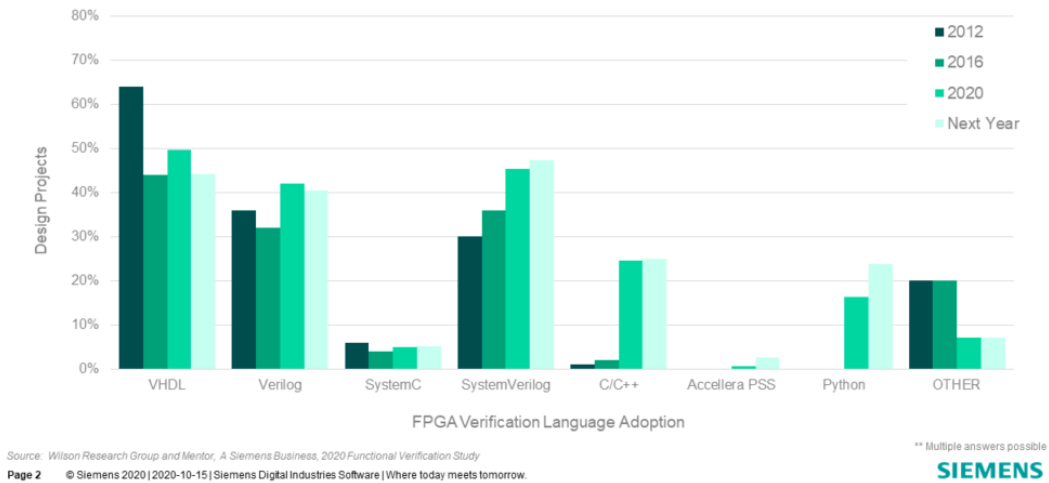


Figure 79: FPGA verification language adoption next twelve months [17].

ASIC/IC Verification Language Adoption Next Twelve Months

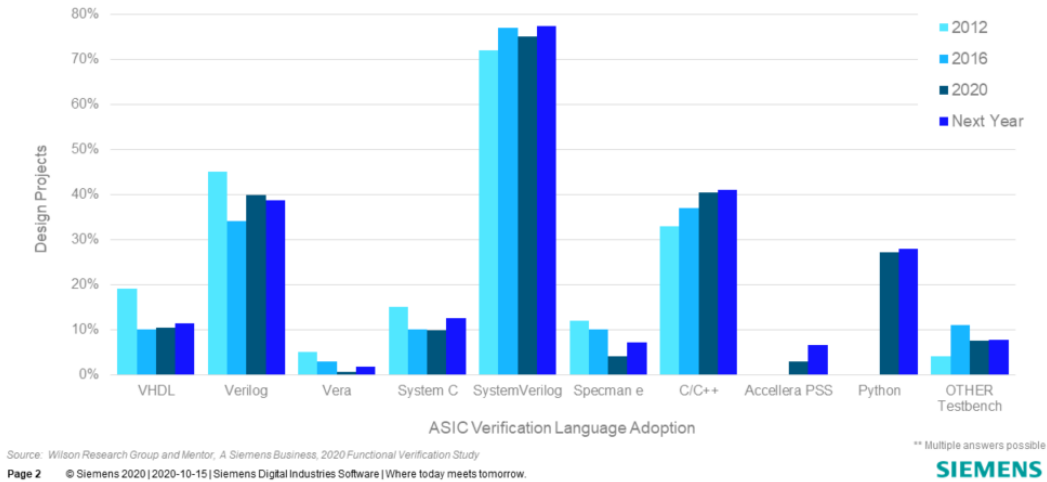


Figure 80: ASIC/IC verification language adoption next twelve months [17].

So, it is clear that using of python in verification is increasing and after implementing UVM in Python it is predictable that using of python will increase more.

DDR5 SDRAM Memory Controller Design and Verification

Chapter 6: Top Level Verification

6.1 VERIFICATION ENVIRONMENT:

The environment is the same as environment of block level verification except new item is added called slave model as shown in Figure 81 [18]:

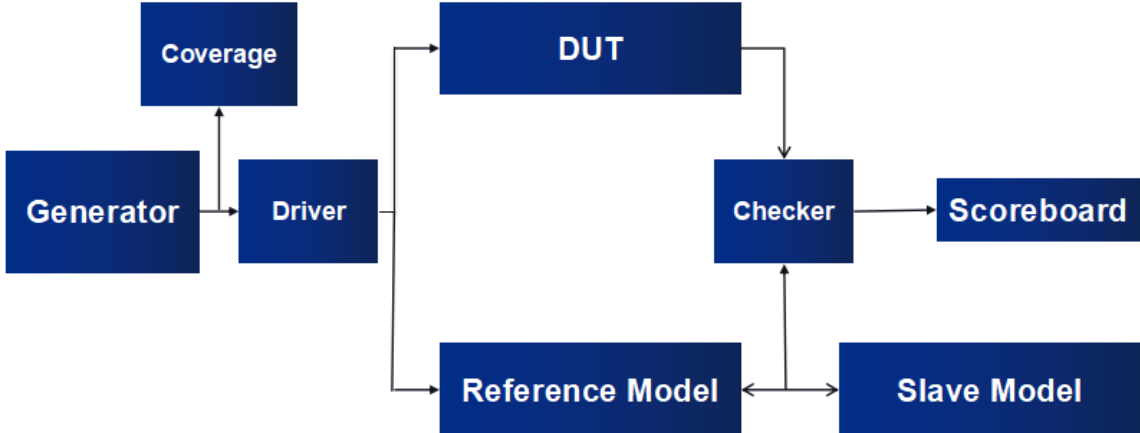


Figure 81 : Verification Environment of Top-Level Verification [18].

Slave model is added to verification environment in order to check the functionality of the controller. It represents model of DDR5 SDRAM Memory to captures the Read/Read_With_AutoPrecharge/Read_Burst/Read_Burst_AutoPrecharge/Write/Write_With_AutoPrecharge/Write_Burst/Write_Burst_AutoPrecharge/ACT/Precharge request from the controller. The Memory Module concatenates the requested address for write or read from Read/Write/ACT commands which are sent to the memory module. For the Write command the write data which is captured from DQ and DQS pins is stored in an associate array in required address. For the Read command the memory module send read data by DQ and DQS pins to controller. For the Precharge command the memory module deactivate the row in requested bank. So, by The Memory Module which is shown in Figure 82 we can compare the data sent from generator to the controller is equal to data received.

DDR5 SDRAM Memory Controller Design and Verification

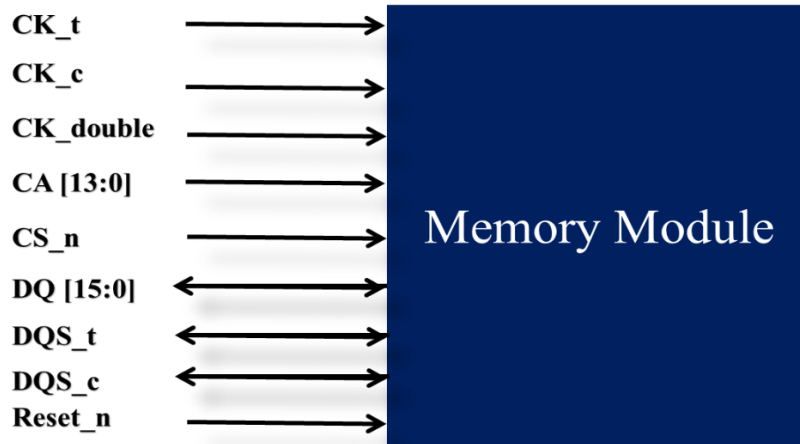


Figure 82: Block Diagram of Memory Module

6.2 FUNCTIONAL COVERAGE PLAN:

The important features that should be covered to ensure the correctness of the Top-Level functionality are the following:

1. Reset.
2. Self-Refresh.
3. Reset and Self_Refresh/command in the same time to ensure the priority of reset.
4. All types of write command (write, write Burst, write with AP, write burst with AP).
5. All types of read command (read, read Burst, read with AP, read burst with AP).
6. Two consecutive writes (write, write with AP) in same bank group.
7. Two consecutive writes (write, write with AP) in different bank group.
8. Two consecutive reads (read, read with AP) in same bank group.
9. Two consecutive reads (read, read with AP) in different bank group.
10. Read after write in same bank group.
11. Read after write in different bank group.
12. Write after read in same bank group.
13. Write after read in different bank group.
14. All types of write command (write, write Burst, write with AP, write burst with AP).
15. All types of read command (read, read Burst, read with AP, read burst with AP).
16. Two consecutive writes (write, write with AP) in same bank group.
17. Two consecutive writes (write, write with AP) in different bank group.
18. Two consecutive reads (read, read with AP) in same bank group.
19. Two consecutive reads (read, read with AP) in different bank group.
20. Read after write in same bank group.
21. Read after write in different bank group.
22. Write after read in same bank group.
23. Write after read in different bank group.
24. Multiple read after multiple write in different bank group.
25. Multiple read after multiple write in different bank group.

DDR5 SDRAM Memory Controller Design and Verification

6.3 REPORTED BUGS:



Bug #1: First Command

First command after initialization stuck at WAIT_ACT state in Command_FSM as shown in Figure 83. This happened as in this state timing between consecutive active commands is checked, so it won't be satisfied as this is the first command and there are not previous commands to count these timings.

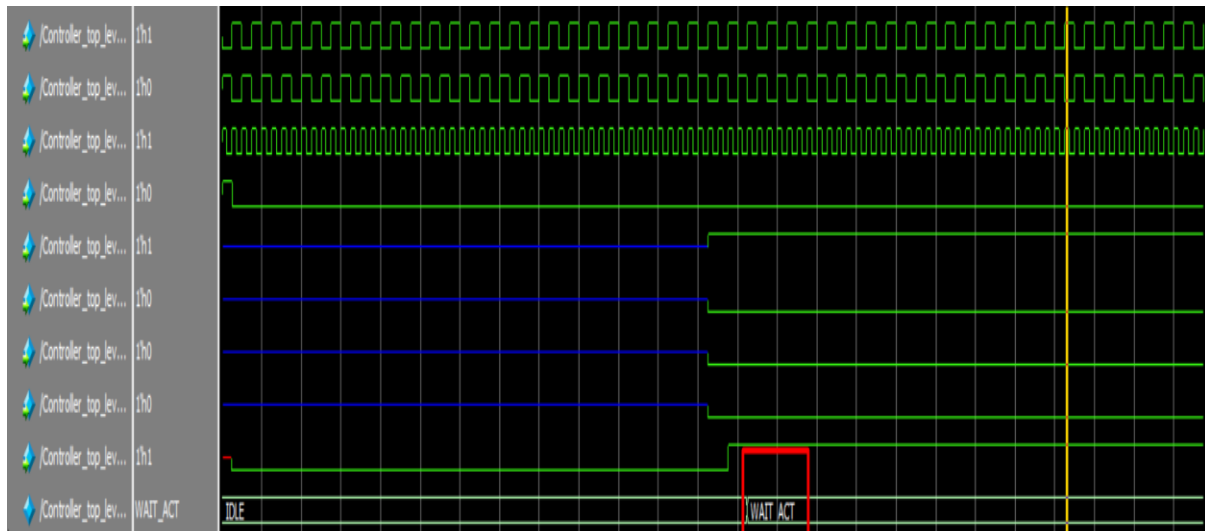


Figure 83:Top Module is stuck at WAIT_ACT state



Modification: This problem is not related only to timing of active commands; it's also related to read and write timing parameters. We solved this problem by adding status register to store 1 after initialization indicating that the command will be executed later is the first command and no need for checking timing between consecutive commands in Command_FSM.



Bug #2: Same_Bank_Group Signal

The value of Same_Bank_Group signal changes each cycle as controller receives commands from CPU each clock cycle, so we lose this information as it's not stored like commands, addresses and data in FIFO which leads strange performance.

DDR5 SDRAM Memory Controller Design and Verification



Modification: Same_Bank_Group signal is also stored with command and address in Command_Address_FIFO.



Bug #3: Read Words with length less than burst length as shown in Figure 84 due to mistake counters which are used in Read_FSM block.

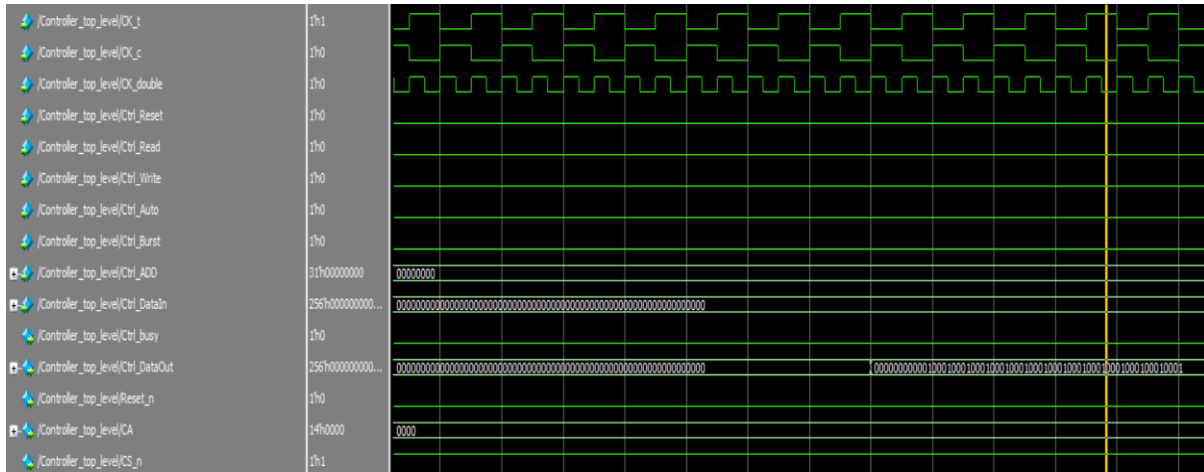


Figure 84: Read words with length less than burst length



Modification: Counters in Read_FSM are fixed with correct values



Bug #4: The Memory Module and the Controller can drive DQ and DQS pins in Write or Read operations so this produces a bug as DQ and DQS pins have now multiple drivers.



Modification:

The solution for this bug is by using 3-State Buffer or more commonly a Tri-state Buffer as shown in Figure 85. A Tri-state Buffer can be thought of as an input-controlled switch with an output that can be electronically turned “ON” or “OFF” by means of an external “Control” or “Enable” (EN) signal input. This control signal can be either logic “0” or a logic “1” type signal resulting in the Tri-state Buffer being in one state allowing its output to operate normally producing the required output or in another state where its output is blocked or disconnected.

DDR5 SDRAM Memory Controller Design and Verification

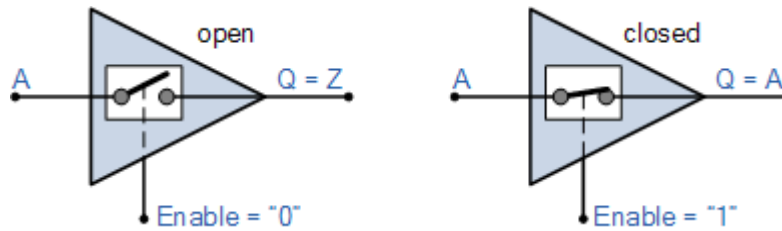


Figure 85: Tri-state Buffer.

When activated into its third state it disables or turns “OFF” its output producing an open circuit condition that is neither at a logic “HIGH” or “LOW”, but instead gives an output state of very high impedance, High-Z, or more commonly Hi-Z.

6.4 FUNCTIONAL COVERAGE RESULTS:



Figure 86:Functional Coverage Report of Top Level

DDR5 SDRAM Memory Controller Design and Verification

6.5 CODE COVERAGE RESULTS:

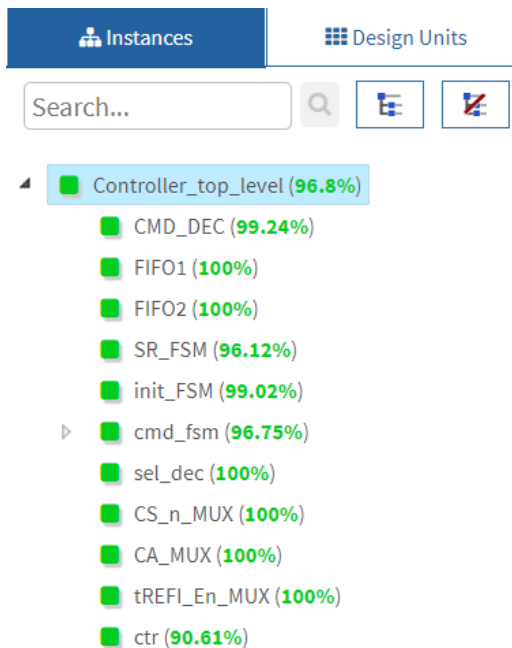


Figure 87: Code Coverage Report of Top Level

6.6 FINAL RESULTS

6.6.1 Initialization Sequence

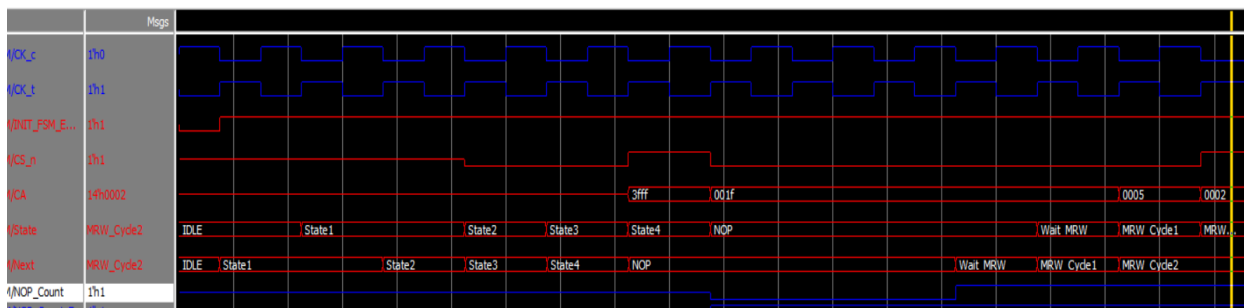


Figure 88: Waveform of Initialization Sequence from Top Level

DDR5 SDRAM Memory Controller Design and Verification

6.7 QUESTA VERIFICATION IP

6.7.1 Overview

After we have verified our design using our verification environment, as a sanity check we used Questa Verification IP (QVIP) DDR5 Memory Model to verify our design. The DDR5 Questa memory model provides the infrastructure to create models of various DDR5 memory devices that you can connect to a memory controller designs under test (DUTs). it includes parameterized SystemVerilog modules that you instantiate in a Verilog or UVM test bench and connect it to a design under test (DUT).it provides various APIs for configuration, callback, backdoor, and other operations, which you can use in a test as required. For example, use the configuration API `set_delay` to change certain delay timing values of the memory model during runtime. To load a memory image for initialization or at any other point in the test case, use a backdoor or memory access API. During runtime, the instantiated memory model responds to the signal-level protocol for front-door access. The model responds to the APIs and provides full functionality and timing accuracy for the supported memory device.it also includes built-in assertions, performance statistics collection, and transaction logging features to identify issues. These debug features abstract the memory accesses to high-level transactions, which makes it easier to analyze the data. The typical memory model flow is shown in Figure 92 [19] :

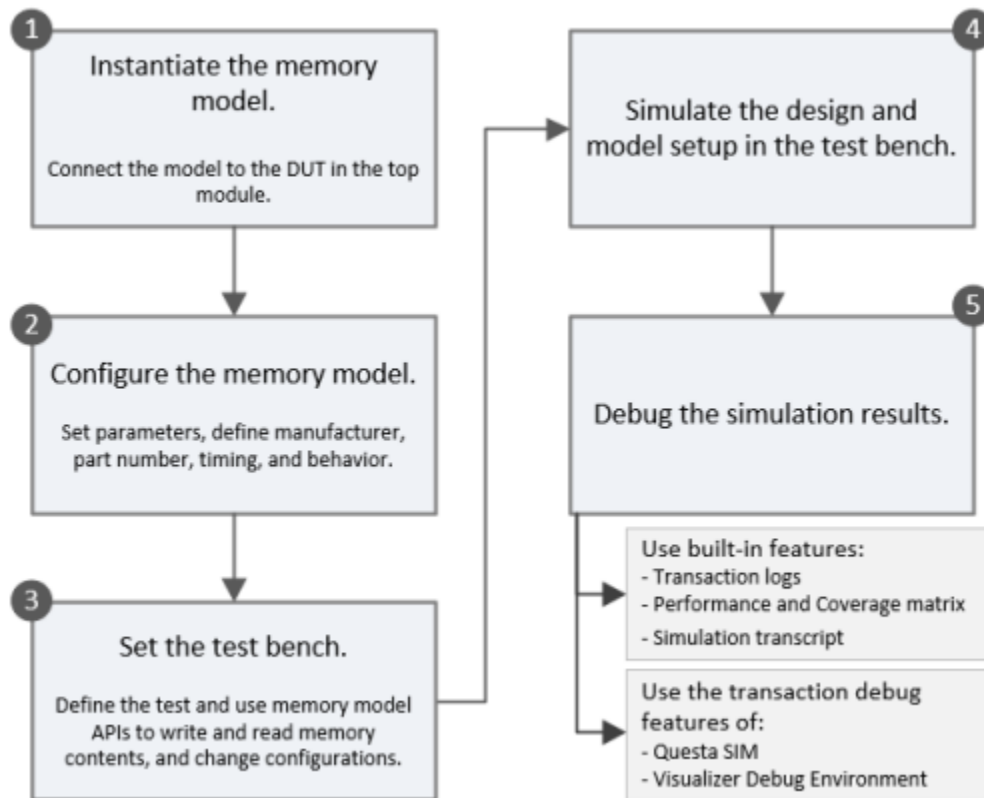


Figure 92: Questa Memory Model Flow

DDR5 SDRAM Memory Controller Design and Verification

6.7.2 Memory Model Components

The DDR5 Questa memory model is based on the Questa Memory Library architecture. This architecture supports features to instantiate and configure a model in your test bench, and debug functional verification issues. The main components of the memory model are shown in Figure 93 [19]:

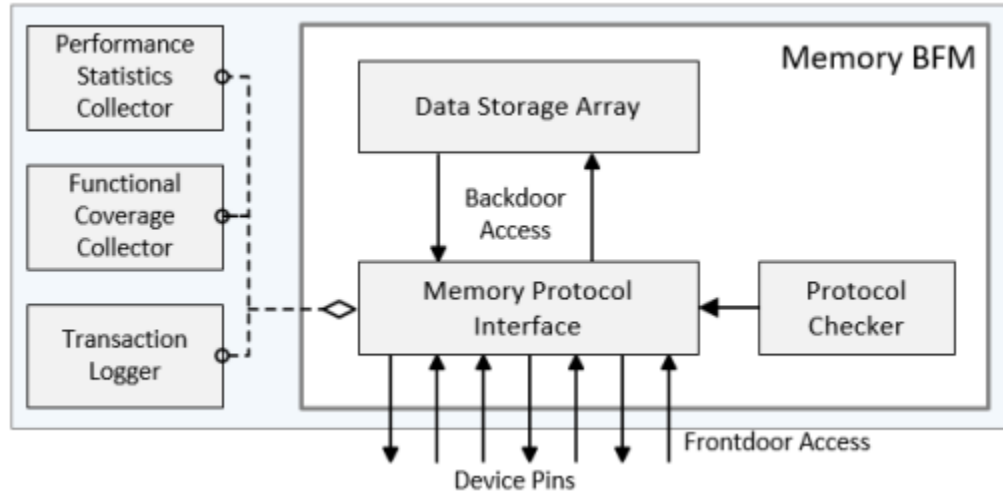


Figure 93: Memory Model Components

The description of each component is shown in Table 49 [19]:

Table 49: Description of Memory Model Components

Component	Description
Memory Bus Functional Model (BFM)	Implements a transaction-based bus function model with a SystemVerilog interface
Data Storage Array	Acts as the memory of the model
Front-door Access	Implements the interface over the memory protocol to access the contents of the memory At runtime, access to the memory takes place using the signals of the protocol front-end interface.
Backdoor Access	Loads or unloads the contents of the memory At runtime, data is transferred in and out of the storage array using the backdoor and memory access APIs.

DDR5 SDRAM Memory Controller Design and Verification

Protocol Checker	Implements runtime assertion checks to determine how the memory protocol responds to transactions
Performance Statistics Collector	Collects statistics of various performance parameters
Functional Coverage Collector	Collects information about DUT coverage at the transaction-level model
Transaction Logger	Logs transaction details in a file This component is disabled by default. You can enable the component when you configure the memory model for your test bench

6.7.3 Configurations

We used the graphical user interface of QVIP Configurator (Configurator) to configure and create the model, which can then be instantiated in an existing test bench. Configurator also provides the option to create the complete test bench where the memory controller and the memory model instance are connected and configured according to requirements. We connected Memory Model with our DUT as shown in Figure and configured Memory Model by choosing configuration of device x16 and speed 3.2 GHZ as shown in Figure 94 [19]:

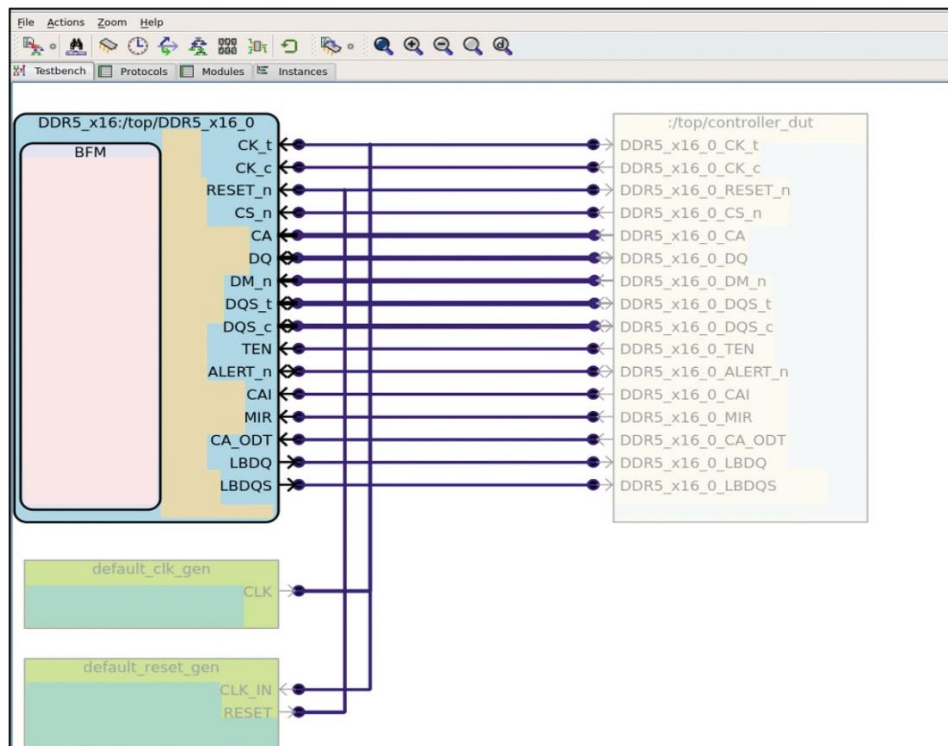


Figure 94: Test bench using QVIP Configurator

DDR5 SDRAM Memory Controller Design and Verification

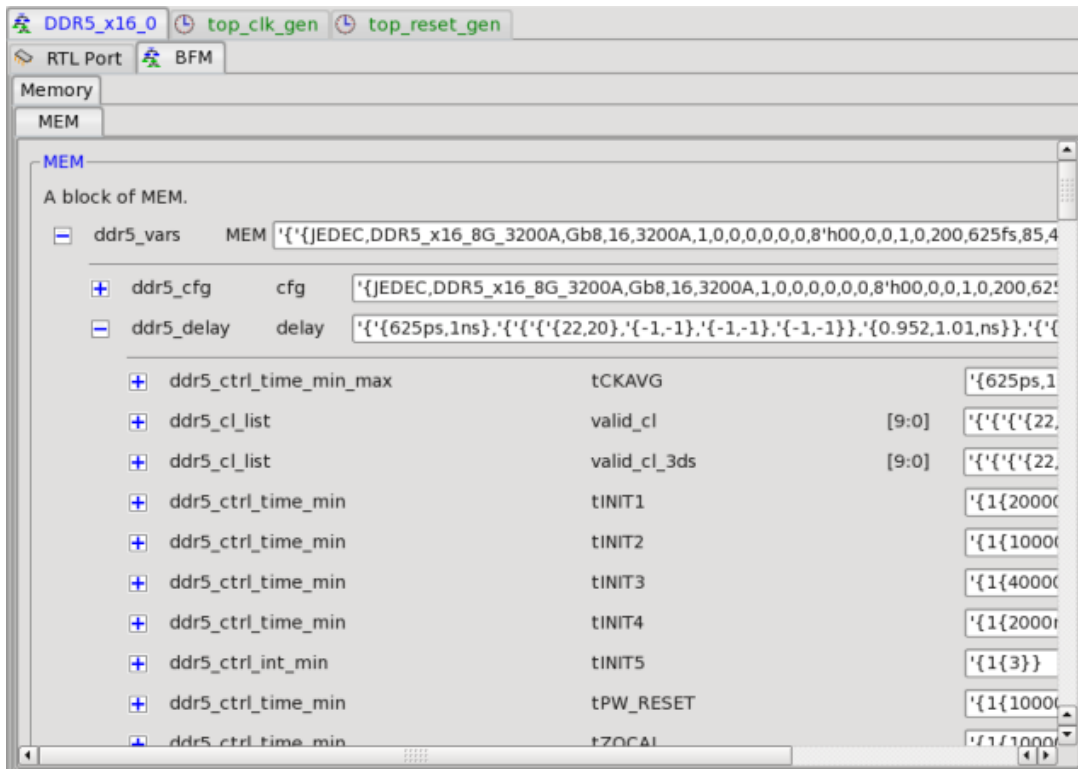


Figure 95 :Memory Model Configurations

6.7.3 Results

QVIP reported bugs, we will present some of them:



Bug #1: DQ should be high “Z “during self-refresh, however is observed as” X”.



Modification: This is intentional bug made by us, we replaced Z with X as Questa simulation tool reports error in compiling when it founds Z.

DDR5 SDRAM Memory Controller Design and Verification



Bug #2: Minimum Value of timer tCSH_SREXIT is “13 ns” however, timer is configured as “10 ns”.



Modification: We edited the value of this parameter in Counters block with the correct value.



Bug #3: Any valid command like read or write for bank that is not activated can be issued only after tRP (interval for precharge of current row) has been elapsed.



Modification: we solved this bug by founding that value of tRP is not as configured by tool, so we edited with value configured by tool.

DDR5 SDRAM Memory Controller Design and Verification

Chapter 7: Conclusions and Future Work

7.1 CONCLUSION

In this thesis, we have seen the need of new DDR protocol (DDR5) and its applications in industry. We have discussed basic features of DDR5 SDRAM Memory controller through design of blocks that implemented these features and verified the functionality of this design on blocks level and top module level and yielded to results that meet performance needed by JEDEC79-5 standard. We also have discussed a comparison between verification using UVM and Python based verification (COCOTB).

7.2 FUTURE WORK

This section provides ideas for further research and extension to the thesis work proposed. These are possibilities to improve the DDR5 SDRAM Memory Controller design and verification in order to increase the functionality.

- Adding optional features mentioned in section 2.1.2 to design of controller.
- Performing full system emulation to test DDR5 SDRAM Memory Controller design by porting on FPGA.
- Enhancing the test environment by adding more constraints random test cases.
- Measuring coverage by Questa Verification IP.
- Modelling the existing verification environment of top level to a class based UVM environment.
- Publish a paper on comparison between UVM and COCOTB.

DDR5 SDRAM Memory Controller Design and Verification

REFERENCES

- [1] Lukas Steiner, Dr.-Ing. Matthias Jung Exploration of DDR5 with the Open-Source Simulator DRAMSys 2021, Available: <https://ieeexplore.ieee.org/abstract/document/9399718>
- [2] Nitin, Bhagwath, et al. "DDR5 design challenges." 2018 IEEE 22nd Workshop on Signal and Power Integrity (SPI). IEEE, 2018. Available: <https://ieeexplore.ieee.org/abstract/document/8401666/>
- [3] <https://news.skhynix.com/why-ddr5-is-the-industrys-powerful-next-gen-memory/> accessed at September 2021.
- [4] Native interface Standard, chapter 3 LogiCORE IP UltraScale Architecture-Based FPGAs Memory Interface Solutions v4.2. Available: https://www.xilinx.com/content/dam/xilinx/support/documents/ip_documentation/ultrascale_memory_ip/v1_4/pg150-ultrascale-memory-ip.pdf
- [5] DDR5 JEDEC Standard Oct. 2021 Available: <https://www.jedec.org/standards-documents/docs/jesd79-5a>
- [6] Taraate, V. (2022). Case Study: FIFO Design. In: Digital Logic Design Using Verilog. Springer, Singapore. https://doi.org/10.1007/978-981-16-3199-3_23
- [7] Jain, Abhishek Kumar, Scott Lloyd, and Maya Gokhale. "Microscope on memory: MPSoC-enabled computer memory system assessments." 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2018.
- [8] Islam, Md Ashraful, Md Yeasin Arafath, and Md Jahid Hasan. "Design of DDR4 SDRAM controller." 8th International Conference on Electrical and Computer Engineering. IEEE, 2014.
- [9] Cocotb's documentation, <https://docs.cocotb.org/en/stable/>.
- [10] S. Jiang, Y. Ou, P. Pan, K. Cheng, Y. Zhang and C. Batten, "PyH2: Using PyMTL3 to Create Productive and Open-Source Hardware Testing Methodologies," in IEEE Design & Test, vol. 38, no. 2, pp. 53-61, April 2021, doi: 10.1109/MDAT.2020.3024144.
- [11] S. Jiang, P. Pan, Y. Ou and C. Batten, "PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification," in IEEE Micro, vol. 40, no. 4, pp. 58-66, 1 July-Aug. 2020, doi: 10.1109/MM.2020.2997638.
- [12] Cieplucha, Marek, and Witold Pleskacz. "New architecture of the object-oriented functional coverage mechanism for digital verification." 2016 1st IEEE International Verification and Security Workshop (IVSW). IEEE, 2016.
- [13] C. Spear, "Systemverilog for verification, second edition: A guide to learning the testbench language features," 3rd ed. 2012 Edition.
- [14] Apoorva H M, Dr. Kiran Bailey, 2020, UVM based Design Verification of FIFO, INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT) Volume 09, Issue 06 (June 2020),

DDR5 SDRAM Memory Controller Design and Verification

[15] T. M. Pavithran and R. Bhakthavatchalu, "UVM based testbench architecture for logic sub-system verification," 2017 International Conference on Technological Advancements in Power and Energy (TAP Energy), 2017, pp. 1-5, doi: 10.1109/TAPENERGY.2017.8397323.

[16] <http://www.fivecomputers.com/language-specification-length.html>

[17] 2020 Wilson Research Group functional verification study. Available: <https://resources.sw.siemens.com/en-US/white-paper-2020-wilson-research-group-functional-verification-study-ic-asic-fucntional-verification-trend-report> accessed at April 2022.

[18] Vutukuri, Venkatesh, et al. "Verification of SDRAM controller using SystemVerilog." 2020 IEEE International Conference on Electronics, Computing and Communication Technologies (CONECCT). IEEE, 2020.

[19] Questa VIP user manual: <https://resources.sw.siemens.com/en-US/fact-sheet-questa-verification-ip-for-ddr5-memory>