A High Performance DDR4 Memory Controller on FPGA

by

Danesh Germchi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2024

Abstract

In this thesis, we introduce a high-performance DDR4 SDRAM memory controller synthesizable design for AMD/Xilinx's FPGA devices. Due to limitations on operating frequency, the design on FPGA presents additional challenges compared to ASIC design: in particular, the controller must be able to issue 4 DRAM commands in a single clock cycle. Utilizing Xilinx's memory controller (MIG) as a foundational framework, our design incorporates features such as the discrimination of received requests based on their origin and the implementation of the FR-FCFS arbitration scheme in the front-end scheduler. Additionally, our memory controller utilizes the Round Robin arbitration scheme in the back-end scheduler to optimize throughput through effective bank parallelism. Our memory controller is able to perform DRAM initialization, refresh, and calibration. Its design is extensible, allowing for further development of other types of DDR4 memory controllers and adaptation for various DDR4 speed grades. The development process involved creating the memory controller's logic blocks in RTL from the ground up, with the integration of specific modules from Xilinx's MIG related to the user interface, calibration, and the physical layer. Standalone verification of each designed module was conducted, followed by the comprehensive validation of the entire integrated project. To evaluate the performance of our memory controller and Xilinx's MIG, we conducted extensive assessments using both EEMBC benchmarks and synthetic benchmarks in simulation. These evaluations provide a comprehensive comparison of their performance across various scenarios, offering valuable insights for further developments in the field.

Acknowledgements

I want to express my heartfelt appreciation to Rodolfo Pellizzoni, my supervisor, for his constant encouragement and exceptional guidance throughout the completion of this research. Prof. Pellizzoni's dedicated support and willingness to invest his time significantly contributed to the successful accomplishment of this research project.

I extend my gratitude to Ali Abbasi for his invaluable assistance during the development of this research, and Dr. Mohamed Hassan for providing continuous guidance throughout the entire research process.

Last but not least, I would like to thank Prof. Nachiket Kapre, and Prof. William D. Bishop for reviewing this thesis and providing valuable comments.

Dedication

To my beloved mother, father, and sister whose enduring love, sacrifices, and unwavering support have been the cornerstone of my academic journey – this thesis is a tribute to your profound impact on my life and aspirations.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Multiprocessor systems facilitate the simultaneous execution of multiple threads on a single chip. Such systems comprise independent processing cores that share the same memory subsystem. They offer advantages compared to single-core systems such as power efficiency, scalability, and improved throughput. However, sharing resources among cores presents a significant challenge when designing such systems. Threads accessing shared memory subsystems may interfere with each other and if this conflict is not effectively handled it can lead to extended waiting times for access to shared resources and result in degraded overall performance. The DRAM memory stands as a vital shared resource in a multi-processor system, and interleaved requests from threads can degrade the performance, and negatively impact locality. Additionally, this interference can lead to more serious problems such as unfairness between some threads at the expense of others, potentially causing important threads to be starved for extended periods. Commercial-off-the-shelf (COTS) memory controllers [15][16][39][28] are designed to improve the overall performance in average-case by reordering requests in multi-core systems. These memory controllers employ intricate optimizations that result in significantly heightened latency in the worst-case. In recent years, researchers in the embedded system field have developed a new line of memory controllers [11][12][14][48] specifically designed to strictly bound the worst-case latency for realtime applications, such as avionics. These memory controllers prioritize minimizing the worst-case latency that a request experiences throughout the memory hierarchy in the system. In addition to bounding worst-case latency, considerations for average-case optimizations, similar to those implemented in COTS memory controllers are also an interesting topic in the realtime community.

1.2 Problem Statement

Various memory controller proposals exist, yet most are only evaluated based on architectural-level simulations, and there is a lack of open-source memory controller implementations for ASICs. On the contrary, FPGAs prove advantageous when aiming to implement configurable memory controllers. A configurable memory controller provides users with the flexibility to intervene in the memory controller and DRAM device interface, enabling manipulation of the designs for enhanced performance. However, this path is not without challenges.

A significant challenge arises from the fact that the maximum frequency of FPGA fabric typically ranges from 300 MHz to 400 MHz, while for example DDR4 DRAM clock frequencies range from 800 MHz to 1600 MHz. For this reason, contrary to an ASIC design, an FPGA controller must be able to schedule and issue multiple DRAM commands per clock cycle. The inherent complexity in the memory controller design requires efficient FPGA implementations to achieve high frequency. Nonetheless, implementing memory controllers offers the advantage of more accurate performance evaluation on real hardware compared to simulation. In particular, architectural simulators typically ignore management operations such as refresh and calibration that are crucial to maintain data correctness in high-performance DRAM devices.

Given the widespread use of FPGAs in cloud and data centers, efficient execution of memory-intensive programs on these platforms is crucial. FPGA vendors like Xilinx and Intel offer users various memory controllers. Our project aims to build upon Xilinx's memory controller, MIG. Despite MIG being designed to achieve high FPGA fabric frequency of 333 MHz, its ability to issue DRAM commands is highly constrained. Depending on the workload, this can result in significantly reduced performance, as we show in our evaluation. Additionally, MIG lacks the capability to track the origin of requests, a feature that can be valuable in real-time memory controllers.

1.3 Objective

Our research group objective is to design and implement a high-performance DDR4 DRAM memory controller on Xilinx FPGAs. Compared to the highly constrained Xilinx MIG, we seek to study trade-offs between achievable implementation frequency and freedom of the controller to dynamically schedule memory requests and DRAM commands. Additionally, our goal is to implement a modularized controller that can be extended in the future for developing and testing different DRAM scheduling and management solutions for both cloud / data centers and embedded systems.

1.4 Solution

Our memory controller design is based on Xilinx's MIG, which includes scheduling logic blocks along with other modules designed for interfacing with DDR4 memory on Xilinx FPGAs. While we utilized these modules from Xilinx, we re-engineered the scheduling part entirely. A significant amount of effort was spent studying the requirements of the Xilinx DDR4 physical interface regarding initialization, refresh and calibration, and integrating the corresponding management blocks in our design. Our memory controller is unique in its ability to distinguish requests from different sources (cores), a feature absent in MIG. Moreover, our memory controller adopts the FR-FCFS arbitration scheme, leading to significant performance improvements. We present two versions of the memory controller, with the initial design excelling in performance but operating at a lower frequency. The second version, while introducing additional constraints on scheduling, is more suitable for FPGA implementation. This version leverages the time-independency between different commands. We designed our memory controller in a parametric way which makes it configurable. Furthermore, we structured our modules to facilitate future reuse, with the intention of developing alternative schedulers.

1.5 Contribution

In this thesis, we present a high-performance memory controller that is implemented from the ground-up in RTL. The major contributions are the following :

- We provide an analysis of the MIG building blocks based on its available code, highlighting its structure and the intercommunication between modules. We highlight the limitations and requirements imposed by the DDR4 physical interface implemented on the Xilinx FPGA as a hard IP, and the role of the management blocks to maintain compliance in the memory controller. We then discuss the design of the scheduling logic and its key limitations.

- We present a new request scheduler and command scheduler design for front-end and back-end arbitration in the memory controller respectively. We show how to incorporate request queues and command queues into the design, and how to integrate our design blocks with the the rest of the Xilinx MIG. We introduce two versions of the command scheduler, trading-off operating frequency vs the freedom to dynamically issue commands.
- We evaluate the performance of our design using RTL simulation and compare it to the existing MIG based on both synthetic memory traces, and traces derived from CPU benchmarks.

1.6 Structure of This Thesis

This thesis is organized as follows. Chapter 2 provides the required background on DRAM and memory controllers. Chapter 3 discusses the structure of Xilinx's MIG followed by Chapter 4 which entails the design of our proposed high-performance memory controller. Chapter 5 provides a comparison between our memory controller and the MIG based on both synthetic and real benchmarks. Lastly, Chapter 6 provides concluding remarks and opportunities for future work.

1.7 Acknowledgment

I would like to thank Ali Abbasi for designing the RTL implementation of the command queue, and the command scheduler version 2.

Chapter 2

DDR4 and Memory Controller Background

This chapter furnishes the essential background information needed for the subsequent chapters. Initially, we delve into the specifics of DDR SDRAM memories in Section 2.1, with a focus on DDR4. We explore the novel features introduced in DDR4 that were absent in DDR3. After establishing a foundational understanding of DDR4, we shift our focus to memory controllers in Section 2.2. In this section, we delve into the critical attributes of memory controllers. Finally, in Section 2.3 we discuss related works.

2.1 DDR4 Background

Due to its favorable cost-per-bit, Double Data Rate Synchronous Dynamic Random-Access Memory (DDR SDRAM), has consistently been preferred for designing main memory subsystems. Notably, the cost-per-bit of DRAM has been diminishing as process technology scales, allowing significantly more DRAM cells to be integrated within the same die area. For this reason, DRAM has been extensively used in many computer systems. In contrast to the ongoing reduction in cost, the latency of DRAM has remained relatively constant. We commence by providing essential background details on DDR. These memories transfer data at both rising and falling edges, which is why they are termed DDR (Double Data Rate). There have been multiple generations of DDR, spanning from the 1st to the 5th. As DDR1 and DDR2 are no longer in common use, we will begin by outlining the features of DDR3. The DDR3 memory system is organized hierarchically, consisting of channels, ranks, and banks and its clock rate ranges from 400 MHz to 1033 MHz. However, the more recent DDR4 generation, introduced in 2013, presents a higher clock rate ranging from 800MHz to 1600MHz while exhibiting lower power consumption. Furthermore, DDR4 introduces an additional address level called bank group, where banks within the same bank group are interdependent. This implies that accessing bank B_0 within bank group BG_0, denoted as access A, incurs larger timing constraints on subsequent accesses to all banks of bank group BG_0 compared to accesses to the banks in other bank groups. The latest generation, DDR5, unveiled in 2020, boasts significantly increased density compared to its predecessor, DDR4. DDR5 features up to twice the number of bank groups as DDR4 and introduces two distinct 32-bit channels compared to DDR4, which provides only one 64-bit channel. Additionally, the fastest DDR5 variant, with a speed grade of 5600 MT/s, exceeds the speed of the fastest DDR4, which operates with a speed grade of 3200 MT/s, by more than 1.5 times. In the rest of the chapter, we focus on DDR4 as our controller designs targets this DDR generation.

2.1.1 DDR4 Structure

In general, a DRAM device is like a three-dimensional array, with its levels named bank, row, and column. A set of DRAM devices is called a rank. The DRAM chip always consists of 16 or 8 banks which can be accessed at the same time while sharing the same command and data bus. Inside each bank, there is a two-dimensional array of memory cells each storing 1 bit. In addition, there is a row buffer within each bank that temporarily stores the content of the most recently accessed row of that bank. Once the data is placed into the row buffer, subsequent read and write commands to that bank can be performed quickly, and this reduces the latency associated with accessing different columns sequentially. Also, utilizing a row buffer reduces energy consumption as activating a row consumes more energy compared to column access.

Additionally, DDR4 leverages parallelism in bank groups to reduce the latency of a request and to significantly increase the overall performance of the system. Each bank group is assigned to one DDR4 memory chip, and within each memory chip, there are four banks. There are three models of DDR4 chips namely x4, x8, and x16. The number represents the output data width of the chip, 4, 8, and 16 bits, respectively. Models x4, and x8 have 4 bank groups and 16 banks in total while model x16 has 2 bank groups and 8 banks in total.

In Figure 2.1, the bank group structure of DDR4 is depicted. Only one bank will be activated during any read and write operations. After the data is amplified through sense

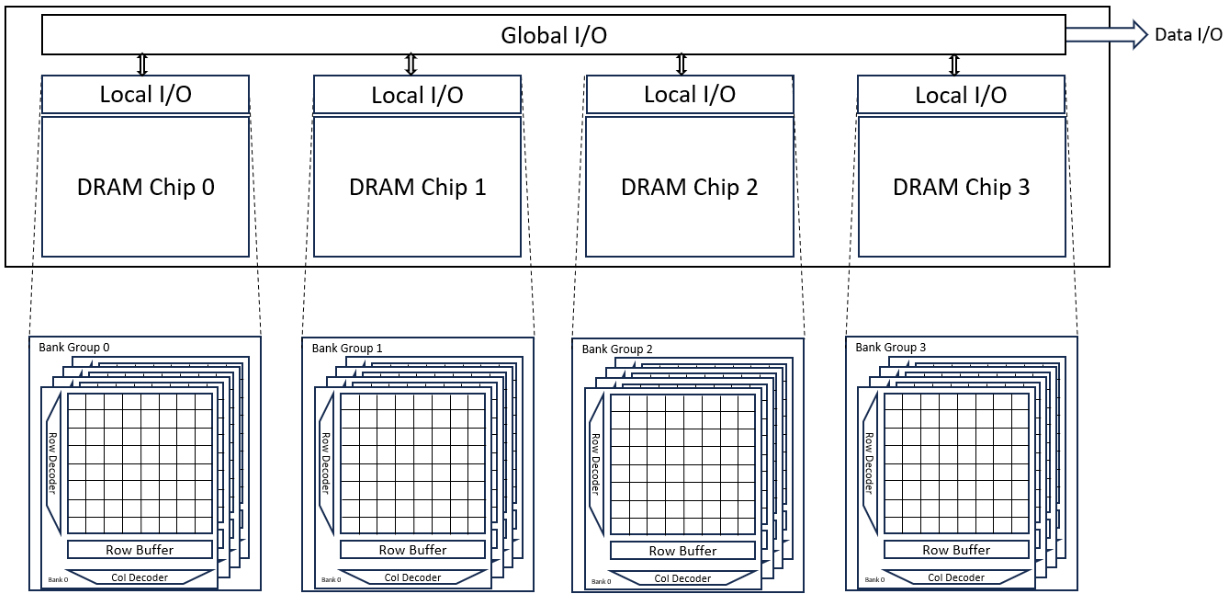


Figure 2.1: Bank group organization in x4/x8 model. The figure only shows the data I/O.

amplifiers, it will be connected to Global IO Gating using Local IO Gating. The width of the Data I/O can be 4, 8, or 16 bits.

2.1.2 DDR4 Access Protocol

There are various pins on the DDR4 memory chip. Pins CAS, RAS, and ACT are used to encode different types of commands [23]. Moreover, pins RAS and CAS are used as address bit 16 and 15 respectively. Additionally, there are separate address pins (A) covering bits 0th to 14th. These pins are used to determine the row and column address. Furthermore, bank groups and banks are assigned through dedicated pins called BG, and BA, respectively. The DRAM operates in two separate clock domains. The main differential clock pin (clk) serves as the reference for command transfer between the memory controller and the memory module. In contrast, the DQS pin acts as a separate clock used as a reference for transmitting data to/from the memory module, with the actual data passing through the DQ pins. on DDR memories data is transferred both on the positive and negative edge of DQS, and that is where the double data rate word comes from. Although both clk and DQS operate at the same frequency, they are not synchronous, and over time, they may drift away from each other.

The flow of accessing DRAM memory consists of three distinct phases, each fulfilling a pivotal role in the retrieval and manipulation of data. Each of these phases is associated with specific commands, and owing to the complexity of DRAM memories, there exist relative timing constraints that govern the transition from one phase to the next.

In the initial phase, known as the precharge phase, when the PRE command is executed, the temporary data stored in the row buffer is sent back to the respective memory cells within the activated row. This phase is essential for preserving data integrity. In addition to PRE, there is a particular command which applies PRE to all banks within the memory, called PREA.

In the subsequent phase, referred to as the activation phase, when the ACT command is given along with the row address, the designated row is activated, and its data is moved to the row buffer. During this phase, the entire row is transferred to the row buffer because subsequent read or write operations might involve different column addresses within the same row. Requests that target the currently activated row within a bank are termed "open requests", while those targeting rows different from the currently activated row are referred to as "close requests."

In the final phase, known as the I/O gate phase, four different commands can be issued. RD and WR commands, accompanied by a column address, are used for read and write operations, respectively. Additionally, RDA and WRA commands are available, which are equivalent to RD and WR but also include an auto-precharge function. After executing these commands, there is no requirement to initiate precharge at the start of the new request.

Additionally, DDR4 supports other types of commands, with one crucial command being the REF (refresh) command. This command is employed to refresh all the stored bits within all cells in the memory banks. Because DRAM essentially stores data as charges in capacitors, and these capacitors tend to lose their charge gradually over time, the refresh command plays a crucial role in preserving data integrity and ensuring the continued reliability of stored information. Furthermore, to account for voltage and temperature variations across the board, the drivers of pins in DRAM need to be calibrated periodically, as elaborated in Section 2.1.4. ZQCS and ZQCL are the designated commands for performing these calibration tasks.

2.1.3 Timing Constraints

As highlighted in Section 2.1.2, it is important to note that there are specific timing constraints between the commands, depending on which banks and bank groups they are directed at. The timing constraints can be categorized into two groups:

1. **Intra-bank timing constraints:** These constraints apply to commands that target the same bank. They define the order and timing of commands within a single bank.
2. **Inter-bank timing constraints:** In contrast, inter-bank timing constraints apply to commands that target different banks. They specify the timing requirements when transitioning between commands directed at separate banks.

In Table 2.1, all important timing constraints are listed. We make the following observations:

- Since open requests target the same row, they do not need any PRE and ACT commands issued. Thus, they proceed faster than close requests which require additional timing constraints such as t_{RC}, and t_{RAS} which are particularly long.
- There is an extra timing constraint once a request type changes from RD to WR or vice versa. These additional timing constraints are denoted by t_{RTW} (Read-to-Write), and t_{WTR} (Write-to-Read). Therefore, switching between different request types can be expensive and affect the overall performance of the memory.
- The burst length in DDR4 is 8 which means it takes 4 clock cycles to transmit data, $t_{BUS} = 4$.
- In addition to these, there are other timing constraints associated with refresh (t_{REFI}, t_{RFC}).

The example shown in Figure 2.2, depicts the relation between a Read CAS directed to Bank a, and column n. The PRE command can be issued after t_{RTP} followed by ACT which can be issued t_{RP} after the PRE command. It is noteworthy that the data burst of length 8 will be placed on the DQ pin after t_{RL}.

Furthermore, in DDR4, the timing constraint between two ACTs targeting two different banks in two distinct bank groups (t_{RRD_S}) is lower than that targeting two different banks in the same bank group (t_{RRD_L}); these constraints are referred to as short and long, respectively. Similarly, there are (t_{CCD_S}), and (t_{CCD_L}) between two CASes directed to two different banks. For instance, in Figure 2.3, the first ACT activates row n in bank group a, and bank c while the second ACT accesses the same row n but in bank group b, and bank c. The imposed timing constraint is t_{RRD_S} since two ACTs are directed to different bank

	Inter-Bank Constraints			Intra-Bank Constraints	
	Description	Cycles		Description	Cycles
t_{RRD}	ACT to ACT	l=6, s=4	t_{RL}	RD to DATA	18
t_{FAW}	4 ACT Window	26	t_{WL}	WR to DATA	12
t_{WTR}	WR DATA to RD	l=9, s=3	t_{WR}	WR DATA to PRE	18
t_{WtoR}	WR to RD	25	t_{RP}	PRE to ACT	18
t_{RTW}	RD to WR	12	t_{RCD}	ACT to CAS	18
t_{BUS}	DATA	4	t_{RTP}	RD to PRE	9
t_{CCD}	CAS to CAS	l=6, s=4	t_{RC}	ACT to ACT	57
			t_{RAS}	ACT to PRE	39

Table 2.1: DDR4-2400U timing constraints [22]. l and s refer to the long (same bank group) and short (different bank groups) timing constraints, respectively.

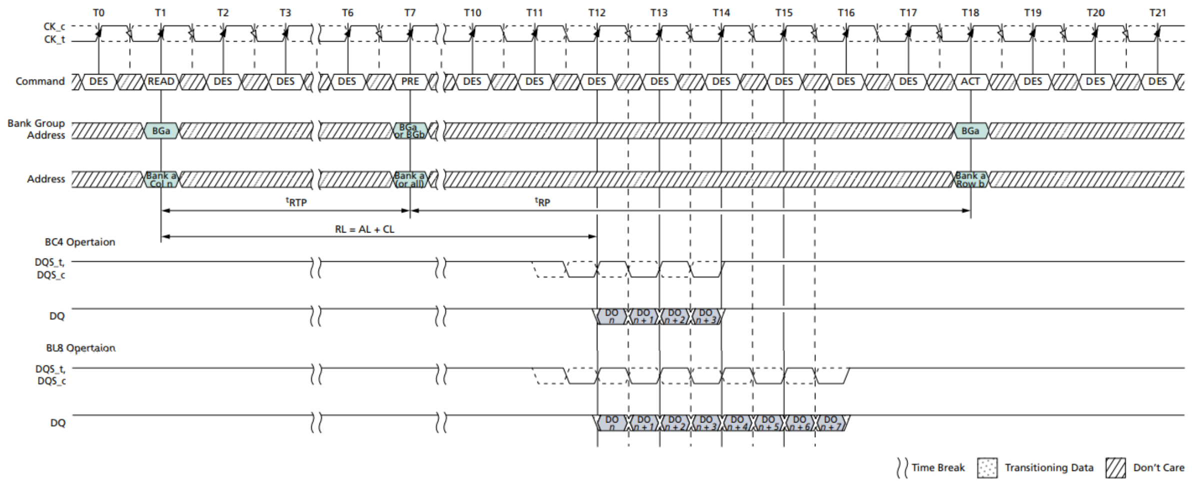


Figure 2.2: The example shows the relative timing constraint between a Read CAS, followed by PRE, and ACT targetting bank a in bank group a. (reproduced from [23])

groups. However, the third ACT activates row n in the same bank group as the second ACT but in bank d, and the timing constraint in this case is t_{RRD_L}.

Another important ACT-related timing constraint is t_{FAW}, specifies a timeframe during which a maximum of four Activate commands can be issued consecutively. During this window, 4 ACTs can be issued back-to-back with the $tRRD_S$, and $tRRD_L$ timing constraint between them. However, once you have executed four ACT commands, you must wait until the t_{FAW} window expires before issuing next ACT command.

When the CAS type changes, transitioning from read to write or from write to read, specific timing constraints are imposed, denoted as t_{RTW} and t_{WTR}, respectively. This introduces additional latency on top of the request and can potentially degrade the system's performance. Therefore, the frequency of switches between read and write requests is a critical factor that influences the overall system performance.

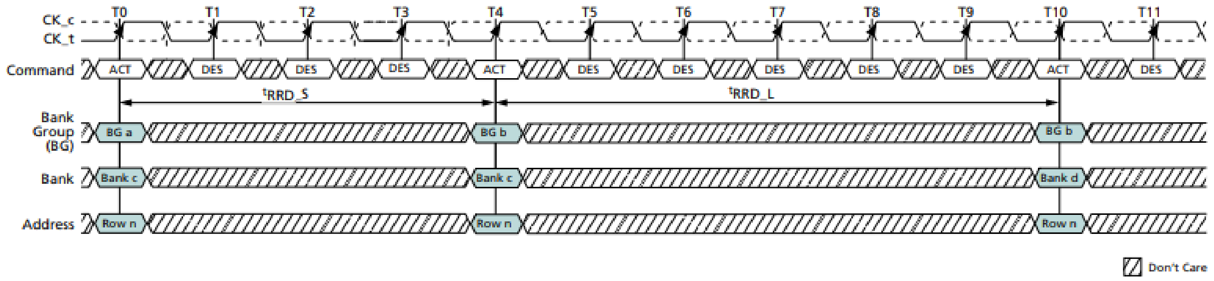


Figure 2.3: t_{RRD_S}, t_{RRD_L} shown for three ACTs accessing the same row in different banks and bank groups.(reproduced from [23])

2.1.4 DDR4 Initialization

DDR4 memory is not ready to operate in normal mode immediately after it is turned on. Depending on the topology of the circuit, trace delays, and other environmental factors, some initialization steps are required. The initialization first begins with a **DQS Gate Calibration** stage during which the read DQS preamble is detected and the gate responsible for data capturing will be calibrated. In the next stage, **Write Leveling** is performed to align DQS with the primary clock in DRAM chip for write requests. During this phase, the DQS is delayed until the 0-to-1 transition on DQ is detected. Similarly, for read requests once again DQS is calibrated to align well with a primary clock in the **Read Leveling** stage. It is worth noting that between these stages, sanity checks are run to ensure that each stage has successfully finished.

In a high-speed I/O interface, the signal integrity (S/I) can quickly degrade due to variations in the impedance of the output driver caused by process, temperature, and voltage (PVT) fluctuations. A ZQ calibration is widely adopted across various generations including DDR3/4 [33][29] and even DDR5 [49]. During ZQ calibration, the pull-down resistor in the DRAM chip is matched to the external reference resistor, and the pull-up impedance is matched to the calibrated pull-down resistor over PVT variation. ZQ calibration is performed using two commands, ZQSC, and ZQSL which stands for ZQ Short Command, and ZQ Long Command, respectively. After all the above stages, initialization ends with ZQ Long calibration after which the memory is ready to operate in normal mode.

In addition to the above stages, at run time the memory device needs to be calibrated. As mentioned before, ZQ Short calibration is used to tune the internal resistors of drivers to increase S/I. Furthermore, additional calibration steps might be required at run time depending on the structure of the DRAM chip. Given that our memory controller is built upon Xilinx's MIG, we elaborate on the calibration mechanisms integrated by Xilinx's MIG specifically designed for Xilinx FPGAs in Section 3.5.

2.1.5 JEDEC Family

JEDEC has introduced three groups of DDR memories that cater to different needs including data rate, power consumption, and application use cases. DDRx series are used for general computing and mostly are employed in computer systems. GDDRx is designed to provide high memory bandwidth for graphical processing units. Lastly, LPDDRx was introduced to provide a high data rate while focusing on power consumption. These series are suitable for mobile computing systems. Our memory controller is specifically designed for DDR4 memories.

2.1.6 PHY

The low-level physical interface to an external DDR memory is referred to as the PHY. The PHY provides the interface discussed in Section 2.1.2 to the memory controller. The PHY is a mixed-signal circuit that receives signals from the memory controller and transmits them to the DRAM. In contrast, the memory controller is a pure digital logic circuit.

2.2 Memory Controller Background

Building upon the foundational knowledge of DDR4 background outlined in Section 2.1, we now delve into the details of memory controller design. Specifically, in the following sections, we introduce a widely employed architectural framework. A DRAM memory controller serves as the bridge between the requestors (CPUs, DMAs, GPUs, etc) and the DRAM memory module. Its primary responsibility is to manage access to the DRAM device by issuing commands in accordance with the timing constraints specified by the DRAM standard. In general, the memory controller consists of an address mapper, a request arbiter, a command generator, and a command scheduler.

2.2.1 Address Mapping

Address mapping involves breaking down the incoming physical address of a memory request into its constituent parts, which typically includes rank, bank group, bank, row, and column bits. The process of address translation determines how each request is assigned to a specific bank group and bank. There are two primary categories of mapping policies:

1. **Interleaved Banks:** In this policy, each core can access any bank or rank within the memory system. The most common way of implementing this is that sequential accesses, each corresponding to a burst, are mapped to different banks in different bank groups. However, it may encounter row interference and lead to higher latency: different cores can potentially interfere with each other by closing each other's row buffers, leading to performance degradation. In the average case, the requests tend to be uniformly distributed among banks and the benefits of accessing various banks are more than the downsides of row interference. However, in the worst-case scenario, all the requests can target the same bank which leads to worse overall performance.

2. **Sequential:** In this scheme, consecutive requests target the same bank. Various resources can access the memory module, each called a requestor. Each bank can be uniquely dedicated to a particular requestor, and that specific bank is not shared among other requestors. This scheme is beneficial in real-time applications as a requestor does not impact the row buffer of a bank, and the MC can take advantage of row locality. However, this policy comes with a few downsides. First, the number of banks in DRAM is limited, and if the number of requestors is more than the number of banks, this policy will not be applicable. Second, since the banks are not shared among requestors, sharing data between requestors is a challenge [40].

2.2.2 Arbiters

Since multiple requestors can have access to DDR4 memory, and they share the same address and data bus, there should be a set of rules that govern the access grant of these requestors. Arbiters can be used on more than one level in memory controllers. Typically, in COTS memory controllers, arbiters are used in two levels. The first one arbitrates requests that arrive at the memory controller, this arbiter is known as the front-end arbiter which is employed in request scheduler design. The second one arbitrates at the command level, which is known as the back-end arbiter, and is used in command scheduler design. In essence, the back-end arbiter in the memory controller monitors all timing constraints and determines when it is safe to issue a particular command. However, the front-end arbiter determines the order of requests to be processed and translated into proper commands. In other words, both arbiters play an important role in affecting the overall memory performance.

2.2.3 Request Scheduler

Once requests arrive at the memory controller, they will be enqueued into buffers called request queues. Depending on the arbitration scheme implemented at the request level, these queues can be per bank group, or bank, or even requestors. COTS memory controllers mostly apply a modified version of First-Ready-First-Come-First-Served (FR-FCFS) policy to improve memory bandwidth. This scheme prioritizes the processing of the open requests over requests that target other rows that are not activated. This technique results in less commands being generated and subsequently, decreases the latency of the request. However, prioritizing open requests over other requests can lead to the starvation of closed requests. This means that, if there is no limit for reordering of requests, the latency bound for closed requests might be unbounded, which is clearly not desirable [41].

2.2.4 Command Generator

When a request is selected by the request scheduler it is passed to the command generator. In some designs, the command generator is part of the request scheduler design. Depending on the currently opened rows in the row buffers and the address of the request itself, the command generator translates the chosen request into a set of commands. These commands are PRE, ACT, and CAS. However, if the required data width from the requestor is larger than the data width of the memory controller, the number of CASes will be more than one as multiple consecutive CASes are required. Generally, the command generator implements one of the following two policies:

1. **Close-Page Policy:** Using this policy every request has a constant latency. The currently activated row in the row buffer will be closed once its corresponding request

finishes. After each CAS, a PRE is automatically issued. By leveraging this scheme, for the next close request that arrives later, there is no need to issue any PRE command, and this results in reducing the latency.

2. **Open-Page Policy:** Command generator can skip generating the PRE command to not close the current row buffer. Thus, subsequent requests still can gain from the current opened row in the row buffer as it reduces the total latency of the request. If further requests target different rows, the command generator would generate PRE as well as other commands, ACT, and CAS. COTS memory controllers preferably implement Open-Page Policy as it optimizes the latency of open requests in average cases.

2.2.5 Command Scheduler

The command scheduler ensures that the commands issued to the DRAM are in proper order and they respect the relative timing constraints. Moreover, some other constraints might be imposed by the DRAM PHY, which should be honored by the command scheduler. In general, there are two categories of command schedulers:

1. **Static:** In this type of command scheduler a request is serviced by issuing a set of commands with pre-defined timing, designed to meet the JEDEC standard[22]. The static scheduler ensures that each set adheres to timing constraints. While fixed scheduling makes implementation easier, it is less efficient as it struggles to capture the parallelism between different types of requests, such as open and close requests with distinct timing constraints.

2. **Dynamic:** These arbiters schedule commands individually without the need for pre-defined timing like static schedulers. Dynamic schedulers require a set of timing counters to monitor all timing constraints. They issue a command when it is deemed safe to do so. Although more complex, dynamic schedulers are generally more efficient and adaptable to various policies and different types of requests.

2.2.6 Other Functionalities

The memory controller is not only limited to request scheduling, it is also responsible for performing maintenance tasks such as calibration, refresh, and even error correction:

1. **Refresh:** In COTS memory controllers, there is always a module responsible for managing the refresh process. In some controllers, the refresh is automated, with the memory controller itself issuing a REF command every t_{REFI}. In other controllers, the cores notify the memory controller when a refresh is required. The refresh process starts by transitioning all banks to an idle state through the issuance of a PRE command. The memory controller issues a PREA command to the PHY, which places all the banks into the idle state. Subsequently, when all banks are in the idle state, the memory controller issues a REF command to the PHY. Once a REF command is issued, all banks undergo the refresh operation.

2. **Calibration:** All initial and run time calibrations discussed in Section 2.1.4 should be performed and configured by the memory controller. Before the normal mode, the memory controller should learn read and write delays using Read and Write Leveling respectively. It is also responsible for issuing ZQ commands to the PHY in initialization and runtime calibration. Meanwhile, it should also respect timing constraints relevant to these tasks.

3. **ECC:** COTS memory controllers often incorporate a specialized feature known as Error-Correcting Code (ECC). ECC is designed to identify and correct 1-bit errors or detect 2-bit errors in the data. This process involves adding extra bits to the data, which are used for error detection and correction purposes.

2.3 Related Works

2.3.1 Memory Controller Design

DDR memory systems stand as a prominent bottleneck in many computing systems, multi-core systems in particular. A huge amount of research effort has been conducted to increase the overall performance of these memories by applying various techniques. Specifically, numerous novel memory controller designs have been proposed in recent years. All memory controller designs mostly center around one of the first scheduling policies, FR-FCFS proposed by Rixner et al. in [37], and it is commonly adopted in current controllers [50, 36, 47]. FR-FCFS prioritizes commands in two orders. First, it prioritizes open requests over closed requests, and if there is not any such request in the buffer, it issues the oldest request in the buffer. However, FR-FCFS is a thread-unfair scheduling policy as faster cores are always serviced before slower threads, and this can cause starving of slower ones. COTS memory controllers often implement a modified version of the FR-FCFS policy in their schedulers.

Significant efforts have been made to enhance the operation of these memory controllers in multi-core systems, with regard to fairness between threads. The Fair Queuing Memory Scheduler (FQM), derived from the fair queuing algorithm in computer networks [3], aims to achieve an equal division of memory bandwidth among threads [30, 35]. It utilizes a counter per requestor and services requests from the thread with the lowest counter value. FQM does not leverage row locality which results in lower throughput compared to other scheduling algorithms [27, 28]. The stall-time fair memory scheduler (STFM) [27] assesses the slowdown of each thread relative to its isolation execution by quantifying the interference between the threads. Once, the value exceeds a threshold, STFM prioritizes the thread that has experienced the most substantial slowdown. The Parallelism-Aware Batch Scheduling (PAR-BS) [28] is a thread-aware scheduler as opposed to FR-FCFS which tries to enhance the overall performance of the system by establishing fairness among threads. This scheduler groups memory requests into batches and prioritizes older batches over younger ones. Nevertheless, writers in [15] showed that PAR-BS implicitly leads to lower system throughput. Moreover, there are also other schedulers proposed to increase DRAM throughput such as ATLAS [15], and Ipek et al. [13] which employs machine learning techniques to optimize memory scheduler policy.

On the other hand, real-time memory controllers strictly bound the worst-case latency of requests while considering the optimizations made on high-performance memory controllers in average-case. In other words, these memory controllers support the derivation of tight latency bounds. The bound is not necessarily for all the cores; they can only provide guarantees to some cores, not all. These types of real-time memory controllers are called mixed-critical. Essential applications running on real-time systems need to have bounded latency. In FR-FCFS, memory latency can be bounded by putting a limit on the number of request reorderings carried out, as shown in [14][8]. Significant research effort has been put into proposing different mechanisms in predictable memory controllers. The Analyzable MC (AMC) [32] is the first design that adopts static command scheduling with a close-page policy. The Programmable MC (PMC) [10] employs a close-page policy but divides larger requests into multiple bundles using an open-page policy. The Private Bank Open Policy MC (ORP) [41] is the first design that uses a dynamic scheduling scheme with private banks. ORP has implemented a complex FIFO to leverage the maximum bank parallelism. ROC [19] is an open-policy memory controller that employs a rank-switching mechanism to hide the latency of the write-to-read transition. ReOrder reorders the read and write requests such that the number of switching between these two is minimized. Moreover, it is shown that the effect of this reordering is bounded [4]. Furthermore, Mirosanlou et al. in [25][15] proposed the DUETTO methodology that includes a pair of a real-time predictable arbiter (RTA) with a high-performance arbiter (HPA). There is a dedicated estimator that monitors the state of the system, and when it is under heavy load and the maximum latency is not violated, HPA would arbitrate accesses. Otherwise, the system will switch to RTA to grant requests in critical situations.

2.3.2 Simulators

Most proposed scheduling policies in the literature are evaluated using a variety of simulators. Some of these simulators are behind the rapid change of technology in DRAMs such as DRAMSim3 [21], and USIMM [8] that only support DDR2, and DDR3 while most systems adopt DDR4 and even DDR5 technology. Commonly used full-system simulators [1, 5] rely on an inaccurate DRAM subsystem due to simulation speed and performance, which is not sufficient for simulating different versions of MC. Ramulator [17] is a modular DRAM simulator that is designed to explicitly simulate a DRAM device's behavior while it offers a relatively basic MC that supports DDR3/4, LPDDR3/4, and GDDR5. Finally, MCSim proposed in [24] allows the user to rapidly implement, verify, and test a user-written MC, but it relies on a Ramulator for DRAM device simulation which does not include the initialization and calibration steps. All the above cycle-accurate simulators are implemented in C++ while our memory controller is implemented in RTL, and supports all the functional details of the DDR4 device calibration steps.

2.3.3 RTL Designs

Given that the PHY is a mixed-signal circuit, there are no existing RTL implementations of PHY on FPGAs. Olgun et al. in [31] introduced an end-to-end framework facilitating the integration of systems with DRAM-based processing-in-unit techniques. The framework encompasses both software and hardware implementations; the hardware side is implemented on FPGAs. However, it is worth noting that the implemented memory controller is based on the DDR3 JEDEC standard, indicating support for DDR3 memory rather than DDR4. Additionally, both Intel and Xilinx, as FPGA vendors, offer users a memory controller IP. Due to the limitations in FPGA fabric frequency, both memory controllers issue 4 commands to the PHY in a single clock cycle. While Xilinx's MIG is designed to support frequencies up to 333 MHz, as we detail in the next chapter, the controller is significantly limited in its ability to reorder requests and exploit parallelism among banks in the same bank group; as we will later show in Chapter 5, this can lead to sub-optimal performance.

Chapter 3

Xilinx Memory Controller

There are two kinds of memory controllers on Xilinx FPGAs depending on its architecture. For instance, on Versal ACAP devices, there are hardened memory controllers accessible through network-on-chip, these memory controllers (called DDRMC) operate at half the DRAM clock rate, and either support DDR4 or DDR5. On the other side, there is a memory controller IP implemented in the Adaptive Engine of ACAP devices which cannot access hardened DRAM PHY through NoC as there are hardened memory controllers in between.

In Ultrascale architecture, there are two DRAM PHYs, one for the Processing Subsystem (PS) side, and one for Programmable Logic (PL). The ARM cores in Ultrascale architecture interface with the DRAM PHY on the PS side, and there is a hardened memory controller for this DRAM device. On the other side, MIG, a soft DDR4 memory controller IP (MIG) interacts with a hardened DRAM PHY on the PL side.

The MIG includes a memory controller, calibration logic, and PHY. Given that we have reused some of the modules within MIG, understanding the structure of MIG is crucial. MIG provides a variety of IPs intended for the management of various operations. These include a memory controller (which will be replaced with our memory controller), calibration logic (used in our design), and other modules that serve specific tasks within the MIG. In the following sections, we will delve into the design and functionality of the various modules, each performing specific tasks inside the MIG.

3.1 Overview

The MIG is mainly composed of four primary modules each specialized for a task, these modules are as follows:

1. **User Interface (UI):** This layer provides queues that are used to buffer write and read data, and supports reordering if configured.
2. **Memory Controller:** The controller processes a burst of requests from the User Interface, generating transactions to and from DDR4 DRAM. It can be configured to implement either an open-page or close-page policy. The controller prioritizes read requests over writes. It also has the capability to rearrange memory requests between bank groups, optimizing the scheduling of workloads with more random address patterns.
3. **Physical Layer (PHY)**: The PHY comprises both hard and soft blocks. The hard blocks on the FPGA handle tasks such as capturing and serializing data or transmitting and de-serializing data. Meanwhile, the soft blocks are involved in initialization and run-time calibration.
4. **Calibration and Initialization:** This block contains a Microblaze soft IP to control and monitor the sequences of initialization as well as calibration steps performed by the PHY.

All four main parts of the MIG are shown in Figure 3.1. Our proposed MC replaces MIG's Memory Controller, while all other modules including Calibration, User Interface, and PHY would remain untouched. There is a mux using which the process switches between the memory controller and the calibration module. The mux and calibration module are wrapped in top module called **cal_top**. First, PHY as a fundamental interface to DDR4 in the MIG is explained in Section 3.2. Second, the User Interface functionality, through which cores interface to MIG, is discussed in Section 3.3, followed by Section 3.4 in which the AXI interface is described. In Section 3.5, the implementation details inside the calibration module are investigated, and then the memory controller design in the MIG is explored in Section 3.6.

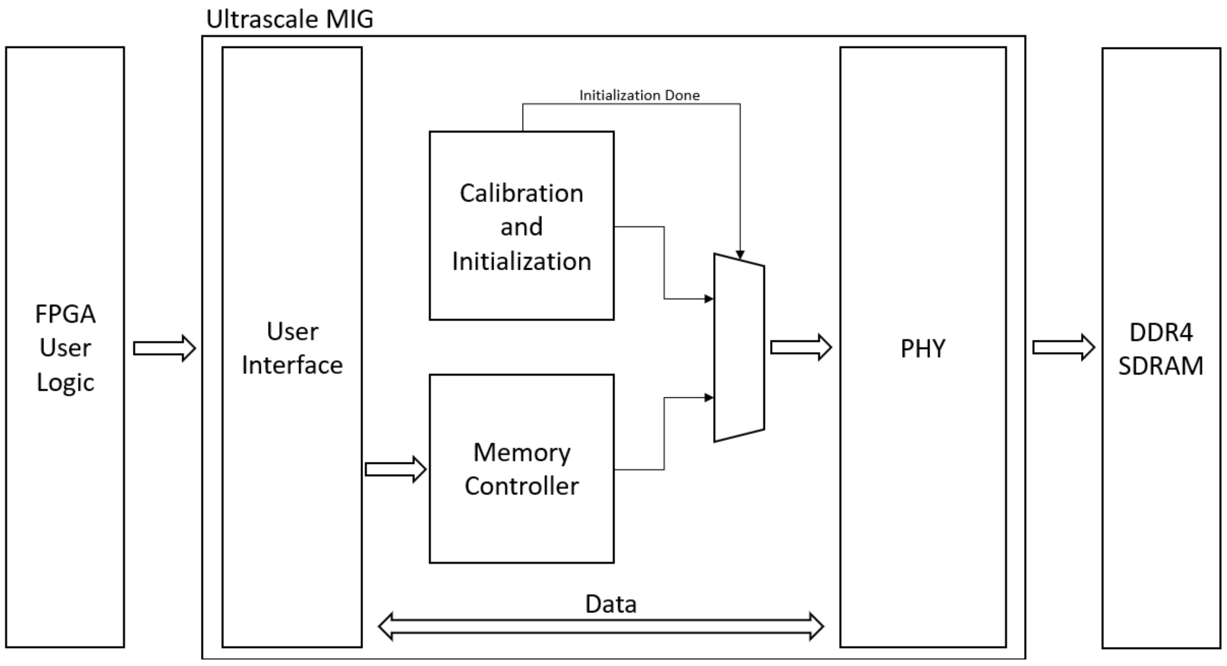


Figure 3.1: UltraScale Architecture-Based FPGAs MIG Core Architecture. [44]

3.2 PHY

The PHY serves as the fundamental physical interface to an external DDR4 DRAM device, incorporating both hard and soft blocks crucial for ensuring the reliable operation of the physical interface. Hard blocks are designed to serialize read data from the DRAM device and de-serialize write data for transmission to the DRAM device. Additionally, the hard block includes a PLL primarily responsible for generating the clock for the DRAM device. On the other hand, the soft core handles the initialization steps necessary to transition the DRAM device into normal mode after being powered on. Furthermore, the soft block provides the calibration logic to set all delays in both hard and soft blocks to interface properly with the DRAM device.

The PHY talks to the other building blocks of the MIG through an interface called the Native Interface. This interface neither supports reordering nor incorporates any buffers. On one side, the PHY receives/sends the data in a burst of 8 from/to the DRAM device in the DRAM's clock domain (serialization, and de-serialization respectively), and on the other side, it transmits the data in a single clock cycle in the memory controller's clock domain. For instance, for a device with data width of 8 bits, PHY receives 64-bit data from upstream (the User Interface) in a single clock at a rate of 300 MHz, while a DDR4 device receives the same data in a burst of 8, in four clock cycles at a rate of 1200 MHz.

Considering that the maximum frequency achievable on implemented logic blocks on FPGAs is approximately 300 MHz, the memory controller logic in MIG operates with a DRAM to a system clock ratio of 4:1. For instance, if the DDR4 memory speed grade is 2400 MT/s, given that DDR4 is a double data rate memory, the command frequency effectively becomes half of the speed grade, which amounts to 1200 MHz. This indicates that commands should ideally be issued at a frequency of 1200 MHz, a frequency that is considerably high for logic implementations on FPGAs. Given the 4:1 ratio, the memory controller would operate at 300 MHz. Consequently, MIG issues a pack of four commands in a single clock cycle to the PHY, resulting in the DRAM device receiving a single command with a frequency of 1200 MHz.

Moreover, there is another limitation imposed by the architecture of the PHY in Xilinx's MIG. CAS commands can only be picked in either slot 0 or slot 2 of the pack issued by the memory controller. However, PRE and ACT can be placed in any of the slots. The PHY also incorporates calibration logic designed to execute timing training calibration tasks, Read Leveling, and Write Leveling which are discussed in Section 2.1.4.

3.3 User Interface

The MIG features a User Interface layer that provides a straightforward FIFO-style interface. This layer functions to buffer data, ensuring that the read data presented aligns with the order of requests. It accepts signals from the Native Interface in the PHY. The Native Interface, lacking buffering, promptly presents returned data to the User Interface as it arrives from the memory, potentially deviating from the original request order. The data buffers employed in the User Interface maintain the order of arrival of requests. Given that the controller is capable of reordering requests, the User Interface takes the responsibility of reordering the returned data from the Native Interface as needed.

The User Interface can only accommodate one request at a time. When a new request arrives at the User Interface, it generates the corresponding bank group, bank, row, and column address of the request. Concurrently, a unique data pointer, indicating the index assigned to the new request in the data buffer, is also transmitted to the MIG. Subsequently, if the request type is read and its CAS is issued, the unique data pointer returns from **cal_top** back to the User Interface. Using this pointer, the read data from the DRAM memory is stored in the index pointed to by the unique returned pointer. Additionally, if the request type is write, its unique data pointer returns one clock cycle before its CAS is issued to the PHY. The User Interface retrieves the corresponding write data settled in the index referred by the data pointer and transmits it to the PHY. In the following situations, the UI would not accept the submitted request from the user:

- The PHY initial calibration process is not finished.
- The signal from the memory controller in MIG indicates that either the request or command queue is full.
- Both read and write data buffers in the UI can store up to 32 64-bit data values. When these buffers get full, no new request will be accepted.

3.4 AXI4 Slave Interface

The Advanced eXtensible Interface (AXI), is a protocol developed by ARM for digital systems, commonly employed in system-on-chip architectures. This protocol streamlines communication and data transfer among diverse components in a digital system, encompassing processors, memory, and peripherals. The AXI establishes a standardized set of rules and protocols, simplifying the integration and interconnection of IP blocks within a system, and contributing to efficient and cohesive system design. The AXI protocol is made up of five channels, two of which are used for read transactions, and the remaining three are used for write transactions. These channels are the following:

1. **Read Address (AR)**: This channel is used for transmitting address information from the initiator to the target during read transactions.
2. **Read Data (R)**: The Read Data channel facilitates the transfer of data from the target to the initiator in response to a read transaction.
3. **Write Address (AW)**: The Write Address channel carries the address information from the initiator to the target during write transactions.
4. **Write Data (W)**: This channel is responsible for transferring data from the initiator to the target during write transactions.
5. **Write Response (B)**: The Write Response channel conveys the response from the target to the initiator after a write transaction, indicating the success or failure of the operation.

Xilinx provides the AXI slave interface block which can directly connect to the User Interface, a layer on top of the User Interface, and facilitates the mapping of AXI4 transactions to the User Interface. The block comprises separate sub-blocks to handle each AXI channel separately, which results in independent write and read transactions. Since the User Interface accepts only one request at a time, if there are simultaneous read and write requests, the block will by default rely on a simple Round Robin arbiter to issue them.

3.5 Calibration

The calibration logic controls the initialization and calibration discussed in Section 2.1.4. The calibration modules provide a thorough process for adjusting all delays within the hard IP (**PHY**) and soft IP (**Memory Controller**, and **Calibration Logic**) to coordinate with the memory interface. Each I/O pin used for data, address, and commands is individually fine-tuned and subsequently combined to ensure the best possible interface performance. After the reset is de-asserted, the calibration module invokes the memory initialization task through which burst length, read, and write CAS latency are configured using configurable registers within the DDR4 device. In addition to initialization, the calibration module implements other crucial types of calibration, including Write Leveling, Read Leveling, and ZQ Calibration [43]. These tasks are mainly employed to center the data along with DQS in order to increase the data-valid window.

The calibration and training processes are carried out by an embedded MicroBlaze (MB) processor which can be configured using a block RAM. This memory stores settings governing the initialization and calibration processes. All other calibration-related modules are wrapped by **cal_top** in RTL files. Module **cal_addr_decode** serves as the interface for the MB to the rest of the system and incorporates helper logic used by the MB.

3.6 Memory Controller

The memory controller is designed with the aim of efficiently handling read, and write requests received from User Interface block. It accomplishes this while ensuring low latency, adhering to all JEDEC protocol and timing requirements, and utilizing minimal FPGA resources. The Memory Controller is composed of the primary logic blocks as the following:

- **Group FSMs:** These state machines are instantiated per bank group and enqueue memory requests that target the same bank group, check DRAM timing constraints and make decisions regarding when to issue PRE, ACT, and CAS commands. More details are discussed in Section 3.6.1. These modules are named **mc_group** in the RTL design files.

- **Safe Logic and Arbitration Units:** These logic units reorder memory requests among Group FSMs based on additional DRAM timing evaluations while ensuring that all issued DRAM commands will be processed and sent to the PHY. In RTL design, these modules are **mc_arb_c**, **mc_cmd_mux_c** for CAS, **mc_arb_a**, **mc_cmd_mux_a** for ACT, and **mc_arb_mux_p** for PRE. Also, another module, called **mc_rd_wr**, manages the switching between reads and writes, as discussed in Section 2.1.3.

- **Final Arbiter:** This module is called **mc_ctrl** and makes the ultimate decision about which commands are dispatched to the PHY and provides feedback to the preceding stages so if a command is sent successfully, it is removed from the queues.

- **ECC:** The MC offers an elective Single Error Correction Double Error Detection (SECDED) Error-Correcting Code (ECC) scheme designed to identify and rectify read data errors involving a 1-bit error per DQ burst, while also detecting all 2-bit errors per burst. However, it does not correct 2-bit errors. Detection of three or more bit errors per burst is uncertain and may or may not occur, but correction is never attempted. The responsible module is called **mc_ecc**.

- **Maintenance:** In addition to command path flow, there are components mainly used to take part in the maintenance path. There is one block, called **mc_ref** dedicated to controlling the refresh and issuing ZQCS commands, and another module named **mc_periodic_read** used to oversee periodic read conditions as discussed earlier in Section 3.5.

In Figure 3.2, all the constituent blocks forming the memory controller of the MIG are shown. The requests are received from the UI and then passed to Group FSMs. Once selected within Group FSMs, the requests are transferred into Final Arbitration, and when it is safe they are conveyed to the PHY. In the meantime, maintenance modules, **mc_ref**, and **mc_periodic_read** take control of the flow when corresponding criteria are fulfilled as discussed in Section 3.6.3.

First, we start by detailing the logic behind the **Group FSMs** in Section 3.6.1, and then we entail the operations of refresh and periodic read in Section 3.6.3.

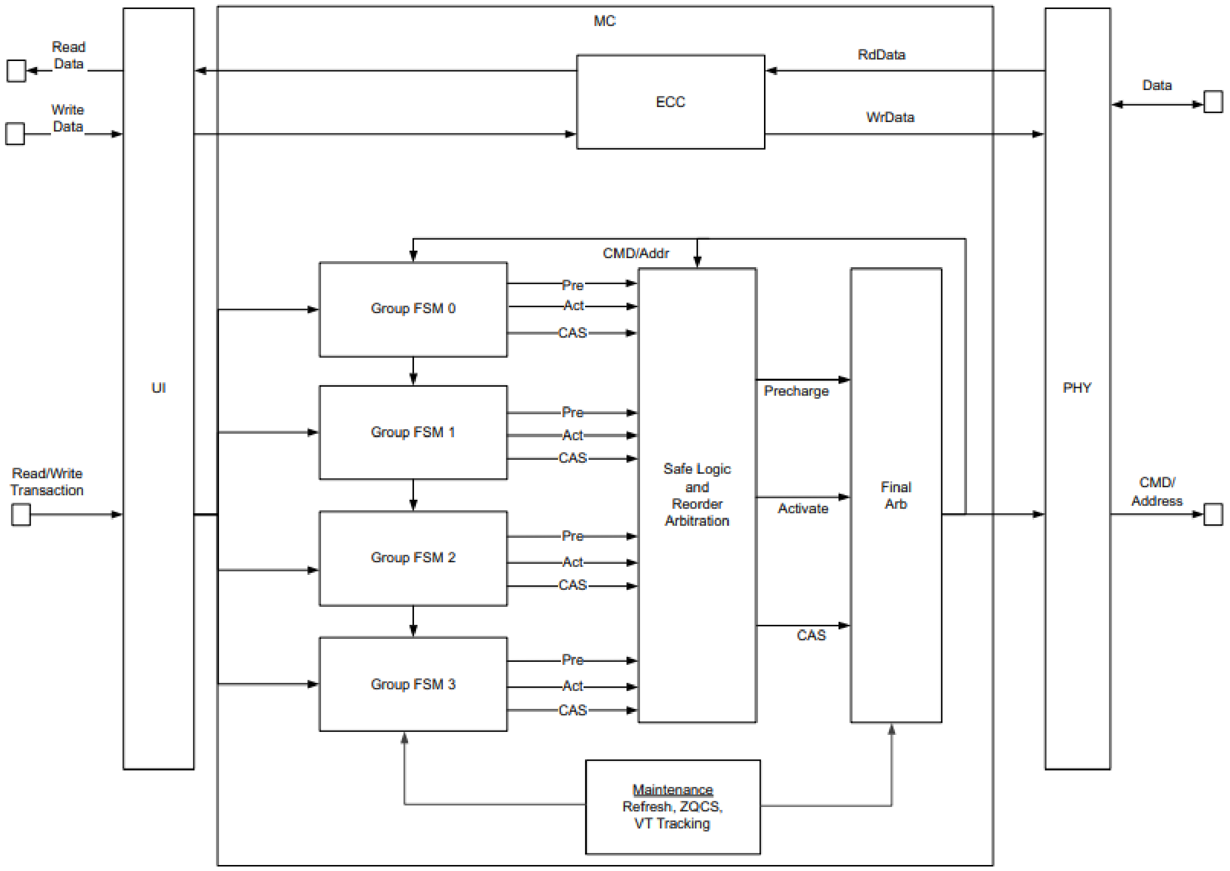


Figure 3.2: MIG Memory Controller Block Diagram. (Reproduced from [44])

3.6.1 Group FSMs

In the MIG there are four Group Machines which are allocated depending on the technology and the data width (x4, x8, x16). In x4, and x8 these groups are allocated 4 banks within one bank group each, and in x16, each group is allocated two banks since the number of total banks in x16 is 8, as discussed in Section 2.1.1.

Figure 3.3 shows the Group FSM block diagram for one instance. The Group FSM block is divided into two primary sections: stage 1 and stage 2, each comprising a FIFO and an FSM.

In stage 1, the block interfaces with the User Interface, where it first enqueues the received request into the stage 1 FIFO. The row of the request at the front of the stage 1 FIFO is examined and shared with the stage 1 FSM. The FSM evaluates whether it

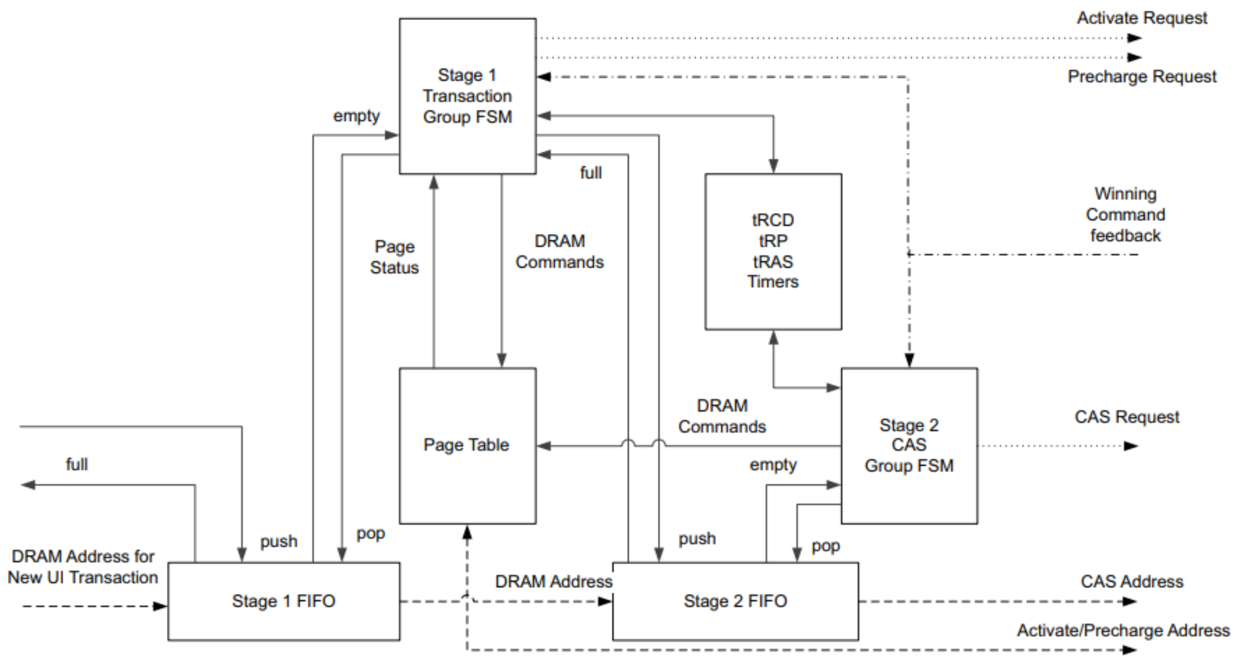


Figure 3.3: MIG Group FSM block diagram. (reproduced from [44])

should send out a PRE or an ACT command and determines the safe timing for issuing these commands. When a request is open and there is no pending Read with Auto-Precharge (RDA) or Write with Auto-Precharge (WRA) commands in the stage 2 FIFO, the transaction is moved from the stage 1 FIFO to the stage 2 FIFO. At this point, the stage 1 FIFO is dequeued, and the stage 1 FSM initiates the processing of the next request.

Simultaneously, the stage 2 FSM handles the CAS command phase of the request at the front of the stage 2 FIFO. The stage 2 FSM sends out a CAS command request when it is safe to do so, taking into account the dedicated timers, t_{RCD}, t_{RP}, t_{RAS}, intra-bank constraints, which are being updated upon issuing PRE, and ACT. Both FSMs can issue their corresponding commands to Safe Logic and Arbitration simultaneously.

In Section 3.2 we discussed the frequency ratio of the PHY and Memory Controller blocks. Since the PHY's clock domain is four times faster than that of the MC, the MC issues a pack of 4 commands to PHY and it is essentially equivalent to issuing a single command in a clock that is 4 times faster. As pointed out in Section 3.2, CAS should be either slot 0 or slot 2 of the packet while ACT and PRE can be put in any slot. However, to achieve higher frequency in the MIG design, in every pack, ACT is only in slot 1, and PRE is only in slot 3. This pattern results in one unused empty slot every clock cycle.

Let us consider an example that illustrates the operation of the Group FSM. Suppose there are no requests in Group FSMs of bank groups 1 to 3, and only close requests are present in stage 1 FIFO of Group FSM 0, with all timing counters set to zero. The request at the front of stage 1 FIFO is read, and its PRE is generated in the Transaction Group FSM, then passed to the Safe Logic and Reorder Arbitration module. As all other Group FSMs are idle, the PRE command is placed in slot 3 of the pack and sent to the PHY. Simultaneously, the timing counter is updated to the value of the intra-bank timing constraint PRE to ACT, t_{RAS}. The Transaction FSM checks in every clock cycle whether the t_{RAS} counter reaches zero. Once it decrements to zero, the ACT of the same request becomes safe to be issued. At the same time, the CAS of the request is pushed into the stage 2 FIFO. Similarly, since other Group FSMs are idle, the ACT will be placed in slot 1 and subsequently sent to the PHY. The corresponding counter, t_{RCD}, is updated with the respective timing constraint. When the t_{RCD} counter reaches zero, the CAS of the request will be issued and placed either in slot 0 or slot 2 of the pack. Meanwhile, it is possible that the PRE and ACT of the next request in the stage 1 FIFO are being generated.

3.6.2 Inter-group Command Arbitration

The controller gives precedence to reads over writes when reordering between Group FSMs is enabled. In situations where both read and write CAS commands are deemed safe for issuance on the SDRAM command bus, the controller exclusively selects read CAS commands for arbitration. This approach reduces the frequency of switching between reads and writes, resulting in fewer incurred switching penalties, specifically t_{RTW} and t_{WTR}. The module that switches arbitration between read and write requests is **mc_rd_wr** as shown in Figure 3.2.

Requests that belong to the same Group FSM are never reordered. The reordering between Group FSM instances is governed by a parameter. When set to "NORM," reordering is enabled, and the arbiter employs a round-robin priority plan, selecting among the Group FSMs in priority order, considering commands safe for issuing to the PHY. The timing of safe command issuance to the PHY can vary based on the target bank, bank group, and their page status, often leading to reordering. In the case of the parameter set to "STRICT", all requests have their CAS commands issued in the order they were accepted at the User Interface.

3.6.3 Refresh and Periodic Read

There is a dedicated module specialized to control refresh and ZQ calibration flow. It supports both user-generated refresh and ZQ provided through the User Interface, and also it incorporates counters that track timing constraints such as t_{RFC}, t_{ZQCS}, and t_{REFI}. Whenever the refresh interval, t_{REFI}, is reached, the module would notify Group FSMs that a refresh is required. In the meantime, it is possible that some requests are on-fly to get issued to the PHY. For this reason, Group FSMs inform the **mc_ref** module whether it is safe to start the refresh process. This module is capable of accumulating postponed refresh tasks and performing all of them when it is safe to do. Sometimes, ZQ calibration is required while a refresh is being performed. In this case, once the refresh finishes, ZQ calibration becomes activated.

As pointed out earlier, the user is able to bypass the **mc_ref** module in the memory controller through a flag, and issue ZQ commands directly through User Interface. The user is responsible for honoring the rate at which these commands should be issued. However, the memory controller adheres to the timing constraints of ZQCS, t_{ZQCS}. The controller may not maintain the precise order of maintenance transactions presented to the User Interface in relation to regular read and write transactions. When initiating a ZQCS, the controller temporarily interrupts system traffic, similar to the default mode, and executes the ZQ commands.

Moreover, limited by Xilinx's PHY design two dynamic and periodic adjustments are required to secure data integrity and a unified working system. The initial requirement involves observing DQS edges to maintain synchronization between the free-running frequency reference clock and the associated read DQS, ensuring phase alignment is locked. The second dynamic adjustment aims to precisely adjust the position of the DQS preamble for the upcoming read. This adjustment specifically focuses on locating the DQS preamble and is essential to accommodate potential drift in the system that may shift the DQS relative to the internal clock. Both of these dynamic adjustments require periodic reads. Consequently, **mc_periodic_read** sends periodic read requests every 1 μs when the bus is idle or performing only writes. This module monitors the command bus and observes whether any read CAS is issued during the past 1 μs, and asserts corresponding signals. Regarding periodic read, when the interval is reached, a signal notifies the first Group FSM, which is for bank group 0, to inject a read request into the queue. For the address of these periodic reads, when there are other requests in the queue of Group FSM and the 1 µs periodic time is due, the periodic read request targets the same address as the preceding read/write request in the queue. When the controller is idle and no reads or writes are in the request queue, the periodic read uses the last address accessed, and it might require an ACT command if it is a close request. These periodic reads are received in **cal_top** module and the relevant tasks are performed to align DQS with the clock.

Chapter 4

Memory Controller Design

We have developed a high-performance DDR4 memory controller that features an FR-FCFS arbitration scheme as its front-end scheduler and implements a Round Robin policy among banks as its back-end scheduler. The command queues are instantiated per bank and the memory controller capitalizes on bank parallelism, diverging from the bank group parallelism characteristic of MIG. Moreover, the memory controller can identify the origin of arriving requests, and enqueue requests per bank and per requestor. We decided to track the core ID of requests for further developments of different memory controllers in the future, real-time memory controllers in particular. We modified the User Interface in the MIG so that when a new request arrives at the UI, it also sends the core ID of a request in addition to its address bits including the bank group, the bank, the row, and the column along with the unique data pointer. Our memory controller is designed to interface with a single-rank DRAM device while the MIG supports multi-rank.

The memory controller issues a pack of four commands akin to the MIG approach. Our memory controller minimizes the switching between read and write requests. Two versions of the command scheduler have been designed. The first version exhibits flexibility in placing ACT and PRE in any slot, enabling the issuance of up to four PRE commands within a single pack. In contrast, the second version restricts the placement of ACT to either slot 1 or slot 3 and allows for only two PRE commands per pack. While the first version outperforms the second at clock parity, it is pertinent to note that it supports a lower frequency compared to the latter.

In this chapter, first, we begin with Section 4.1 to entail the structure of our memory controller followed by Section 4.2, 4.3, 4.4 in which the design of the request scheduler, page table, and command queues are discussed respectively. In Section 4.5, we discuss the timing counters and the implementation of two versions of our memory controller.

4.1 Overview

Our memory controller incorporates three pipeline registers in its design. When a new request is accepted from User Interface (UI), it is enqueued into the request queues, which serve as the first pipeline register in the request scheduler. Once a request is selected by the request scheduler, its corresponding commands are inserted into the subsequent queues, known as command queues. The second pipeline register in our design is the set of command queues, through which the commands progress before traversing the command scheduler logic. After the commands are issued, the relevant PHY signals are generated and registered in the **cal_top** module, constituting the third pipeline register in our architecture.

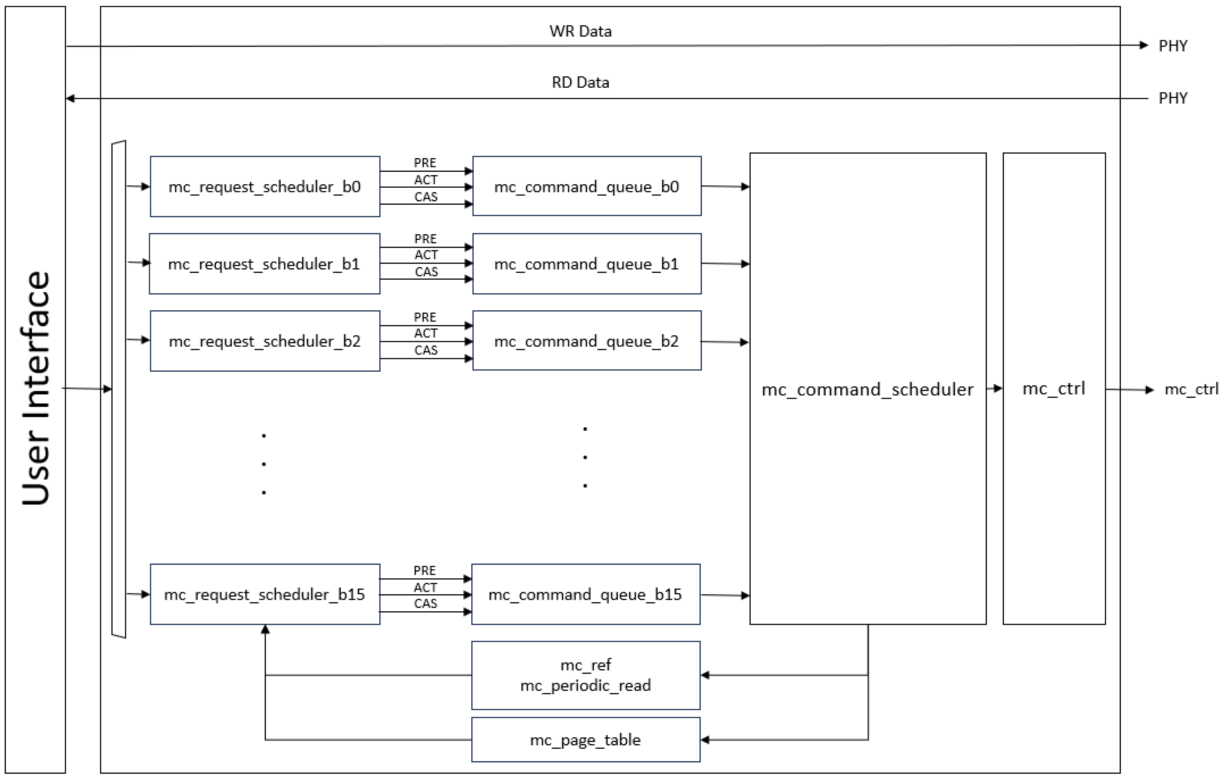


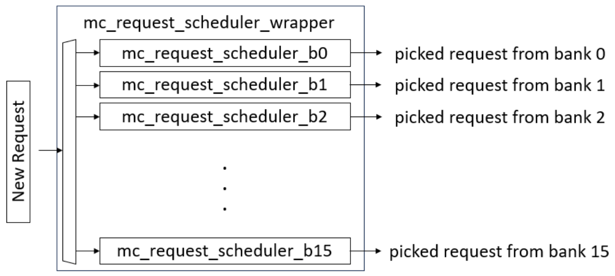
Figure 4.1: Our memory controller block diagram

Moreover, as shown in Figure 4.1, we re-used maintenance modules such as **mc_ref**, and **mc_periodic_read** from Xilinx's MIG design files. Request schedulers, denoted as **mc_request_scheduler_b**, are instantiated per bank and each comes with a command generator issuing PRE, ACT, and CAS commands. Similarly, command queues, denoted as **mc_command_queue_b**, are instantiated per bank. These queues reorder commands within them if needed and each passes the command at its front to the command scheduler, denoted as **mc_command_scheduler**. Finally, module **mc_ctrl** takes the pack of four commands issued by **mc_command_scheduler** and translates each command into the PHY signals as discussed in Section 2.1.2.

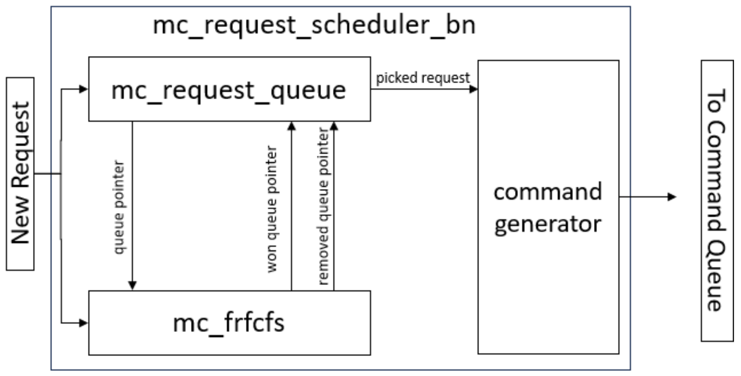
Our memory controller replaces the memory controller of Xilinx's MIG while all other modules such as Calibration Logic, and PHY remain untouched. As pointed out earlier we modified the User Interface so that it receives core ID as well. The reason for employing these modules is to leverage their specific functionalities that are in harmony with DDR4 PHY on Xilinx's FPGAs.

4.2 Request Scheduler

In the memory controller, the request scheduler is composed of **mc_request_scheduler_b** instances with a total number equal to the total number of banks instances are per bank. For example, the structure of the request scheduler with a total of 16 banks is shown in Figure 4.2a. Each **mc_request_scheduler_b** is composed of three primary sub-modules **mc_request_queue**, **mc_frfcfs**, and **mc_command_generator** as shown in Figure 4.2b. Sub-module **mc_request_queue** contains request queues that are per requestor. In addition to request queues, simultaneously, the queue within **mc_frfcfs** enqueues incoming requests in the order of their arrival. Essentially, the queue implements the FR-FCFS scheme as it tracks the relative order of requests once they arrive. At last, **mc_command_generator** generates the commands of the picked request depending on whether the request is open or not, it generates CAS, or PRE, ACT, CAS respectively. The request scheduler is tasked with two primary functions: enqueuing and selection (front-end arbitration). Sometimes, the enqueuing task can be stalled due to the request queues becoming full, or the selection stalls due to a refresh being processed. Now in the following sub-sections, the logic design of two sub-modules, **mc_request_queue**, and **mc_frfcfs** is described. The command generator module simply creates a set of commands from the picked request and passes the set to the command queue module.


mc_request_scheduler_wrapper
New Request
mc_request_scheduler_b0
mc_request_scheduler_b1
mc_request_scheduler_b2
mc_request_scheduler_b15
picked request from bank 0
picked request from bank 1
picked request from bank 2
picked request from bank 15

(a) All **mc_request_scheduler_b** wrapped in **mc_request_scheduler_wrapper**, the total number of banks is 16.



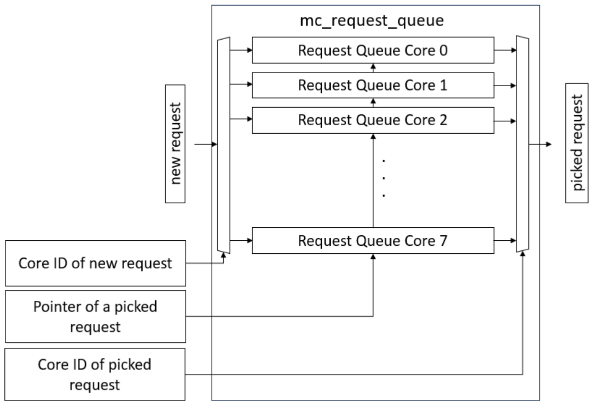
(b) The structure of a single **mc_request_scheduler_b** in bank b_n.

Figure 4.2: The request scheduler architecture.

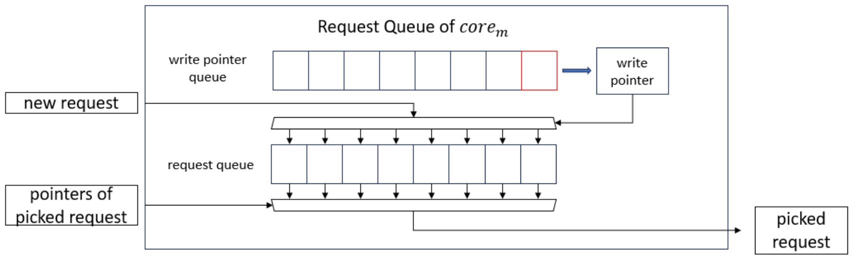
4.2.1 Request Queue

If m denotes the number of cores, sub-module **mc_request_queue** contains m number of pairs of queues, and each pair corresponds to one core. In Figure 4.3a, sub-module **mc_request_queue** for 8 cores is illustrated. Each pair includes two queues, the first queue is used to store all information about requests: the address, type, and data pointer. The second queue stores pointers that point to the empty indices of the first queue. These pointers are used to write a new request to one of the empty slots in the request queue. These two queues are shown in Figure 4.3b.

Upon the arrival of a new request from core C_m directed towards bank b_n via User Interface, the request undergoes the following process. A pointer, Write Pointer in Figure 4.3b is popped out from the Write Pointer Queue and it is used to write the new request into the empty index of the Request Queue corresponding to core C_m. Moreover, when a request is finished, it is safe to be removed from the request queue, and the pointer of the removed slot is returned to the second queue which will be used later for request insertion. Once a request (the picked request in Figure 4.3b) is selected, its address and type are passed to the **mc_command_generator** that corresponds to bank b_n, and subsequently, the resulting commands are issued.



(a) The structure of **mc_request_queue** with 8 number of cores.



(b) The structure of a pair of queues for core M.

Figure 4.3: The structure of **mc_request_queue** in **mc_request_scheduler_b**.

4.2.2 FR-FCFS Arbiter

Module **mc_frfcfs** mainly comprises a FIFO-style queue, called Global FIFO, a comparator, and a one-hot priority encoder. Each entry in the global FIFO includes the row address of a request, the pointer referring to the location of that request in the request queue, and the corresponding unique data pointer of the same request retrieved from UI. The row address of a request is used to determine whether the request is open or close. The pointer of the request queue is used to select the winning request in **mc_request_queue**, and the data pointer retrieved from **cal_top** is used to uniquely identify a request in the Global

FIFO and remove it when it finishes. By leveraging thev Global FIFO, which is common among all the requestors, the scheduler can discriminate the oldest request among all the requests. In Figure 4.4, a Global FIFO with size of 8 is shown. Index 0 (LSB) is the oldest slot, while index 7 (MSB) is the youngest slot. The row addresses of all slots in the Global FIFO are compared against the current opened row of the corresponding page table. After this comparison, a set of flags indicating whether a request is open (ready) or not is generated. Using these flags, the select signal of MUX2 (is_ready in Figure 4.4) is determined. Simultaneously, using a one-hot priority encoder the oldest ready request is picked using MUX1. The priority encoder returns the highest priority index in one hot format. The highest priority is the LSB bit of one-hot encoded output, and the lowest priority is the MSB. For instance, if the ready flags are 8'b0110_1000, the priority encoder would output 8'b0000_1000. If there are any ready requests, the ready request is picked otherwise, the oldest request from index 0 is picked. Once a request is picked, the pointer referring to the location of the won request in the request queue is passed to **mc_request_queue**.

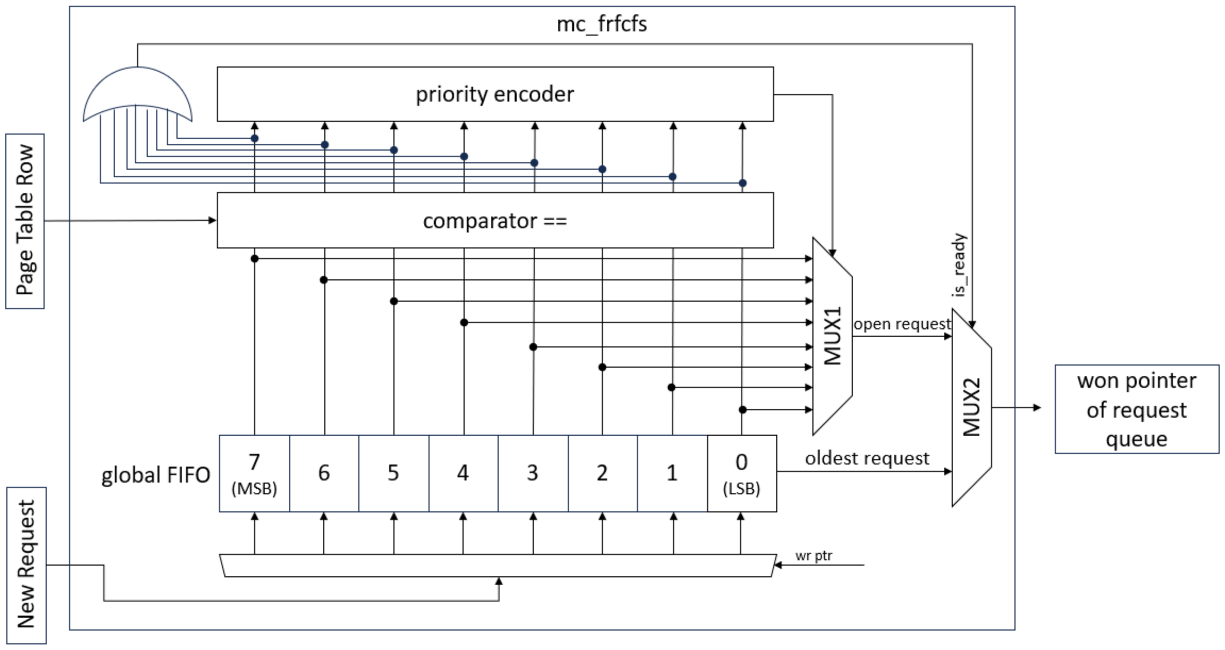


Figure 4.4: **mc_frfcfs** Arbiter Block Diagram

Once a request targeting bank b_n is completed its data pointer returns to **mc_frfcfs** corresponding to bank b_n. Using this data pointer the request is identified in the Global FIFO and subsequently removed from the queue. Additionally, requests located after the removed request are shifted towards LSB, this resembles the shift register behavior of the Global FIFO.

4.2.3 Data Flow

After the arrival of a new request from core c_m targeting bank b_n at UI, the core ID of c_m along with other information of the request will be passed to the corresponding request scheduler, **mc_request_scheduler_**b_n, and the request will be enqueued in the request queue core c_m of **mc_request_queue**. At the same time, the request queue passes the index, in which the new request is placed, to **mc_frfcfs**. Subsequently, **mc_frfcfs** enqueues the new request along with the request queue pointer received from **mc_request_queue**. When a request is picked in **mc_frfcfs**, the corresponding request queue pointer is returned to **mc_request_queue** and then it outputs the address, type, and data pointer to the command generator of bank b_n. At last, the command generator generates the commands.

When a request is finished, it will be identified using its unique data pointer in global FIFO in **mc_frfcfs**, and this module passes the request queue pointer of the finished request to **mc_request_queue**. Subsequently, the request will be removed from the request queue in **mc_request_queue**.

4.3 Page Table

The page table module, identified as **mc_page_table**, tracks the currently activated row for each bank. Each entry in this table corresponds to a specific bank where the opened row of that bank is located. Moreover, each entry has a valid bit indicating whether the corresponding bank of that entry is idle or not. Note that after the PRE command, a bank will be in idle mode. Upon the issuance of an ACT of close request to a particular bank, the row within the corresponding entry of the page table is updated to reflect the row associated with that request. To illustrate the functionality of the page table, consider the example of the FR-FCFS arbiter, **mc_frfcfs**, within the request scheduler of bank b_n denoted as **mc_request_scheduler_**b_n. In this scenario, the arbiter compares the rows of all requests in the Global FIFO against the current row stored in the b_n entry of the page table. Let us assume the current opened row is A. If the request scheduler selects a close request, r, targeting row B, and there are no open requests in the Global FIFO, the issuance of ACT of request r leads to the replacement of the row A in the b_n entry of the page table with row B. Furthermore, once the refresh operation finishes, after which all banks are idle with no activated rows, all valid bits in entries of the page table will be zero, and the rows within banks are no longer valid.

The page table serves as a register holding the current open rows in all banks. When an ACT command is issued from the command scheduler, the table gets updated with a new row in the subsequent clock cycle. On the other hand, the request scheduler queries the activated rows from the page table to select a request in the current clock cycle. Due to the one-clock cycle latency in the page table being updated, there is a possibility that a previously open request, which is, in reality, a close request, might be detected as an open request and passed downstream. To address this issue, we have imposed certain constraints in the command queue design, which will be discussed in the Section 4.4.

4.4 Command Queue

The Command Queue module, denoted as **mc_command_queue** and instantiated per bank, plays a pivotal role in storing commands generated by the corresponding command generator. The command queue is capable of reordering commands in the case of open requests. Let us delve into the logic behind the command queue. The command queue can enqueue commands of three distinct requests at most. Since the command queue is one of the pipeline stages in the memory controller design, and we prefer not to stall the flow, the command queue should store commands of more than one request. Let us say we have two requests r_1, and r_2. The PRE and ACT of the former are picked by the command scheduler while its CAS is not serviced yet, and the latter is in the request queue, and not yet pushed into the command queue. Regardless of request r_2 being open or close, its commands will be pushed into the command queue due to the same reason that we do not want to stall the logic. In other words, when there is a single open request in the command queue, meaning that a CAS is in the command queue, to prevent a waste of clock cycles, the command queue would accept the next request. However, following the discussion in Section 4.3, if a third request r_3, which targets the same row as request r_1 arrives when r_2 is at the front of the command queue, it will not be pushed into the command queue since we do not know when commands of r_2 will be selected by the command scheduler. Although, if request r_1 is still in the command queue, r_3 will be accepted, and the command queue will be re-ordered in such a way that the CAS of r_3 will be placed after the CAS of r_1, and ahead of the commands of r_2. Furthermore, if there is a close request in the command queue, meaning that there is an ACT or there are a PRE, and an ACT in the command queue, no other close request will be accepted since the memory controller still needs to issue at least two commands of the previous close request. The following are the conditions where an open request and a close request would not be accepted by the command queue:

Conditions not to accept an open request

- command queue being full
- number of CAS commands in the command queue being more than 2
- there is no CAS with the same address as the open request, and one PRE in the command queue

Conditions not to accept a close request

- command queue being full
- number of CAS commands in the command queue being more than 2
- number of ACT commands in the command queue being more than 1

When **mc_command_queue** accepts the commands it checks whether it is an open request or close request. For the former case, there is a combinational block logic called CAS Insertion as shown in Figure 4.5 that inserts the open CAS into the command queue. The insertion logic is illustrated in Table 4.1.

Command Queue	Action
‖ - ‖ Any ‖ Any ‖ Any ‖ CAS ‖	After CAS if row matches, otherwise push back
‖ - ‖ Any ‖ Any ‖ CAS ‖ ACT ‖	After CAS if row matches, otherwise push back
‖ - ‖ Any ‖ Any ‖ CAS ‖ CAS ‖	After the second CAS if row matches, otherwise push back
‖ - ‖ Any ‖ CAS ‖ CAS ‖ ACT ‖	After the second CAS if row matches, otherwise push back
‖ - ‖ CAS ‖ ACT ‖ PRE ‖ CAS ‖	After either the first or second CAS if the row matches, otherwise push back

Table 4.1: The CAS insertion logic in the command queue.

However, for close requests, it pushes PRE, ACT, and CAS to the back of the queue. The module always outputs the front of the command queue to the command scheduler. The queue itself is a cyclic queue and it supports simultaneous read and write.

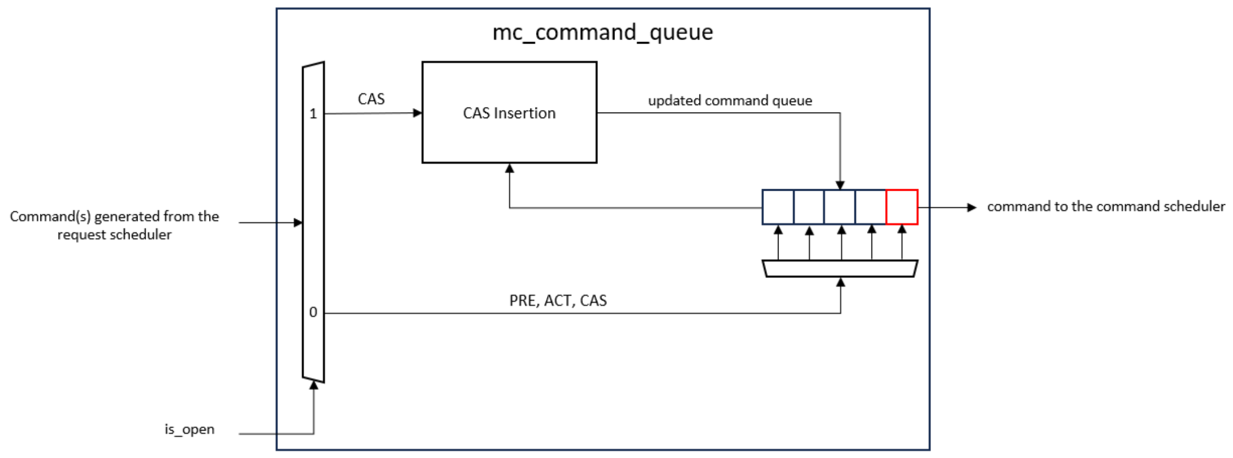

mc_command_queue
CAS
CAS Insertion
updated command queue
1
0
Command(s) generated from the request scheduler
command to the command scheduler
PRE, ACT, CAS
is_open

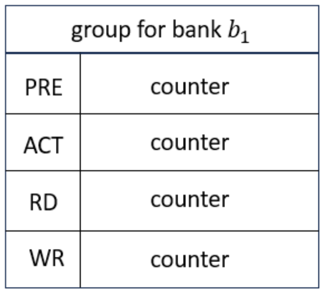
Figure 4.5: The block diagram of **mc_command_queue** module.

4.5 Command Scheduler

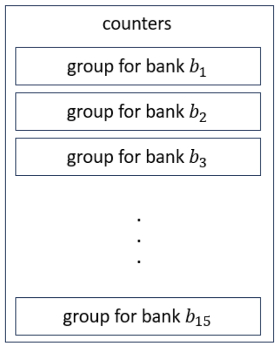
The command scheduler, denoted as **mc_command_scheduler**, receives a single command from each instance of the **mc_command_queue** and selects four commands within a single clock cycle, a limitation imposed by the frequency constraint outlined in Section 3.2. Employing a Round Robin policy, the command scheduler arbitrates among command queues organized per bank, thereby capitalizing on bank parallelism within our memory controller. The primary functions of the command scheduler include command arbitration and the monitoring of timing constraints to ascertain the safety of issuing specific command types. We have implemented two designs of the command scheduler, each representing a different trade-off between the constraints on issuing commands and the level of parallelism in the command scheduler. In Version 1, ACT can be placed in any of the slots, and it can issue up to four PRE commands. On the other hand, Version 2 can only put ACT in either slot 1 or slot 3 and can issue up to two PRE commands. In Section 4.5.2, the building blocks of the 1st version are discussed followed by Section 4.5.3, in which the 2nd version is discussed.

4.5.1 Counters

Counters play a crucial role in ensuring the accurate operation of the command scheduler by monitoring timing constraints and determining the safety of issuing specific commands. The command scheduler relies on counters to keep track of timing constraints, ensuring compliance with JEDEC standards. These counters count down, and when the counter value reaches zero, it indicates that the corresponding command is safe and ready to be issued. These counters are organized into groups, with each group instantiated per bank. Within each group, four counters are designated, each dedicated to a specific command type associated with that bank. Given that there are four distinct command types—PRE, ACT, RD, and WR—that can be directed to a bank, these counters provide detailed tracking for precise command issuance timing. For instance, the PRE counter within a group corresponding to bank b_n determines when PRE command is safe to get issued to bank b_n. The structure of each group is shown in Figure 4.6a, and in Figure 4.6b, the structural arrangement of all groups is illustrated.



(a) A group of counters for bank b_1.



(b) All groups of counters for all banks. The total number of banks is 16.

Figure 4.6: The structure of counters used in the command scheduler.

4.5.2 Version 1

Concerning the frequency limit on the FPGA fabric, discussed in Section 3.2, the command scheduler version 1 should issue a pack of four commands in one clock cycle, each command within one slot. We implemented combinational logic that picks a command for a given slot, given the current counter value, and the Round Robin pointer, and then computes the new value for counters that are affected by the picked command. Also the Round Robin pointer will get updated. Since, we need to pack 4 commands in 4 slots, we chained this combinational logic one after another to pick all 4 commands sequentially. After the last slot (slot 3), the final counters value and the final Round Robin pointer are registered to be used for the next clock cycle. Now let us delve into the details of the combinational logic for one slot.

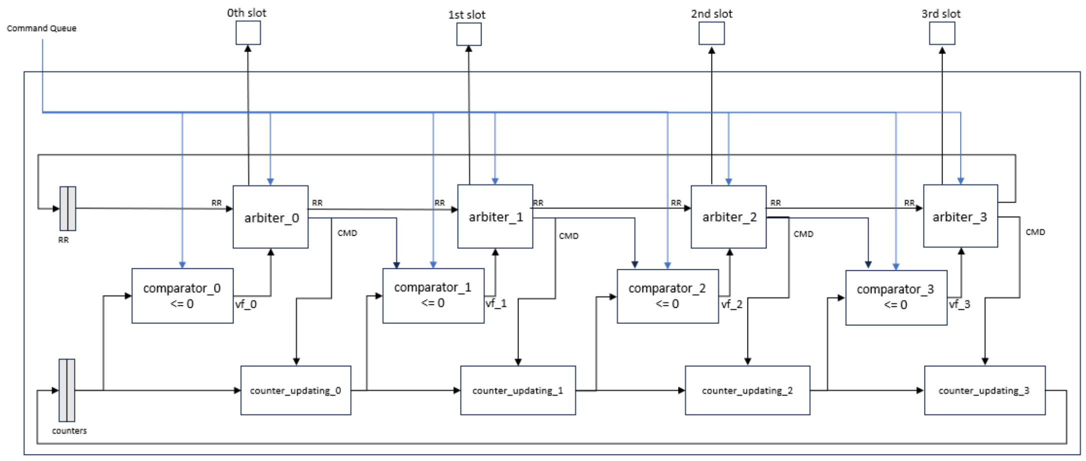


Figure 4.7: The basic structure of command scheduler version 1.

In each combinational logic of a slot, there are three blocks, a comparator, a Round Robin arbiter, and a counter-updating module. For instance, for slot 0, these blocks are **comparator_0**, **arbiter_0**, and **updating_counter_0**. In each clock cycle, the flow begins with filling slot 0. First, **comparator_0** takes the front command of all command queues, and checks whether their corresponding counters are 0 or not, and subsequently outputs a set of flags called **vf_0**. These flags are used by **arbiter_0** to select the commands that are safe to issue concerning timing constraints. Once the arbiter picks a command, it will be placed into slot 0. Lastly, the picked command is passed to **counter_updating_0** block through which the counters will be combinationally updated according to the timing constraints, as discussed in Section 2.1.3.

Round Robin Arbiter

The arbiter implements a Round-Robin policy over command queues which are per bank. The comparator in each slot logic generates a set of flags by which the arbiter would select a command. The arbiter in the command scheduler version 1 employs a set of one-hot priority encoders. The reason that we chose one-hot priority encoders instead of index priority encoders is that the set of flags is nothing but a sequence of 0s, and 1s, therefore, one-hot priority encoders are a good match for this kind of logic. Each priority encoder prioritizes a bit index in the flag. The structure of the arbiter is shown in Figure 4.8. For instance, in priority encoder 0, and priority encoder 15 the highest priority index is the 0-th, and 15-th bit respectively. After the priority encoder, only one of the outputs, selected by the round-robin pointer, is valid, using which the command will be extracted.

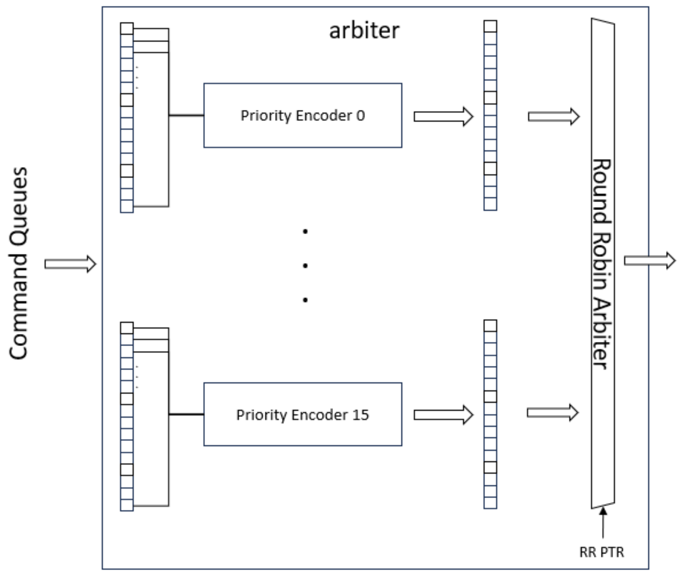


Figure 4.8: The design of arbiter in the command scheduler.

It is important to note that the arbiter always selects among commands that are ready to be issued, as dictated by the state of the counters. As noticed in Section 2.1.3, the penalties for switching between read and write requests are long, so if the memory controller sends a read, other reads will be ready before the writes, so the arbiter prefers not to switch. The same is true for bank groups, since t_{CCD_S}, and t_{RRD_S} are shorter so commands of different bank groups are ready first, and the arbiter prefers to switch between bank groups.

Counter Updating

This module, employing information from the bank group, the bank, and the type of the selected command, identifies the counters to be impacted. Initially, it updates the counters if the new constraint exceeds the current value of a counter. After the completion of this update process for all counters, they are uniformly decremented by one. Subsequently, the updated counters undergo transmission to the subsequent slot logic blocks. All these operations are executed through combinational logic, and no registers are employed in the process. For example, if in slot 0, a PRE command hitting bank b_2 is picked, the counters within the group for bank b_2 are updated if the following timing constraints. (refer to Section 2.1.3) are larger than the current counter value:

1. PRE: t_{RC}
2. ACT: t_{RP}
3. RD: $t_{RP} + t_{RCD}$
4. WR: $t_{RP} + t_{RCD}$

In particular, the ACT commands are affected by the t_{FAW} constraint, as discussed in Section 2.1.3. To keep track of the t_{FAW} constraint, which is common among all banks, we have employed a FIFO-style queue with a size of four; each entry holds a valid bit and a counter that keeps track of t_{FAW}. Whenever a new ACT is issued, a fresh t_{FAW} constraint is pushed into the FIFO along with its valid bit being asserted. When the FIFO becomes full, it indicates that four ACTs have been issued so far (t_{FAW} is triggered), and the t_{FAW} of the oldest ACT in the FIFO is now valid as a timing constraint. Once the t_{FAW} counter of the oldest ACT reaches zero, the next ACT, the 5th one, can be issued. Meanwhile, the oldest ACT will be popped out from the FIFO, and instead the 5th one will be inserted into the FIFO.

Optimization

According to the JEDEC standard, no more than one command of the same bank can be selected before 4 clock cycles, all intra-bank timing constraints are more than 4 clock cycles. We confirmed this for all speed grades of DDR3, and DDR4. Likewise, no more than one same command type, except PRE, to different banks can be selected before 4 clock cycles. This is mainly due to the fact that the smalest inter-bank constraints, t_{RRD_S}, and t_{CCD_S} for ACT, and CAS respectively, are 4 clock cycles. By observing these properties derived from JEDEC timing constraints, we can further optimize the design of the command scheduler version 1.

The most basic structure of Version 1 is depicted in Figure 4.7. Assume that components **arbiter**, **comparator**, and **updating_counter** each have t_{arb}, t_{cmp}, and t_{tm} delay. The critical path in Figure 4.7 starts from **comparator_0** and traverses through all blocks of all slots, and ends in **updating_counter_3**. Therefore, the longest path delay is $4t_{cmp} + 4t_{arb} + 4t_{tm}$. Nevertheless, exploiting the constraint that, in each clock cycle, only one command from a specific bank is selected, and only one ACT and one CAS can be selected, allows these arbiters and updating counter modules to operate concurrently rather than being sequentially interconnected, as shown in Figure 4.9. The main difference between the implementation in Figure 4.9, and that of Figure 4.7 is that comparators in slots 1, 2, and 3 are comparing against 1 instead of 0, which is the case in the basic design. This flow is shown in Figure 4.7. These comparators use the counters from the previous slot, before getting updated, to check whether a command is safe while in the basic design, the comparators use the updated counter. For this reason, in design of Figure 4.9, the picked command is passed to the comparator of the next slot logic, this is to ensure not to pick the same command twice. To elaborate more, the objective is to prevent issuing a PRE to the same bank or issuing two ACTs to two different banks in subsequent slots. The novel idea in the design of Figure 4.9 results the following: the process of arbiter in slot n, **arbiter_n** is parallel with the process of **counter_updating_n-1** (each red parallelogram in Figure 4.9) and this results in a reduction in the longest path delay, either it is $4t_{arb} + t_{tm} + 4t_{cmp}$ or $4t_{tm} + t_{arb} + t_{cmp}$ depending on the values of t_{arb} and t_{tm}.

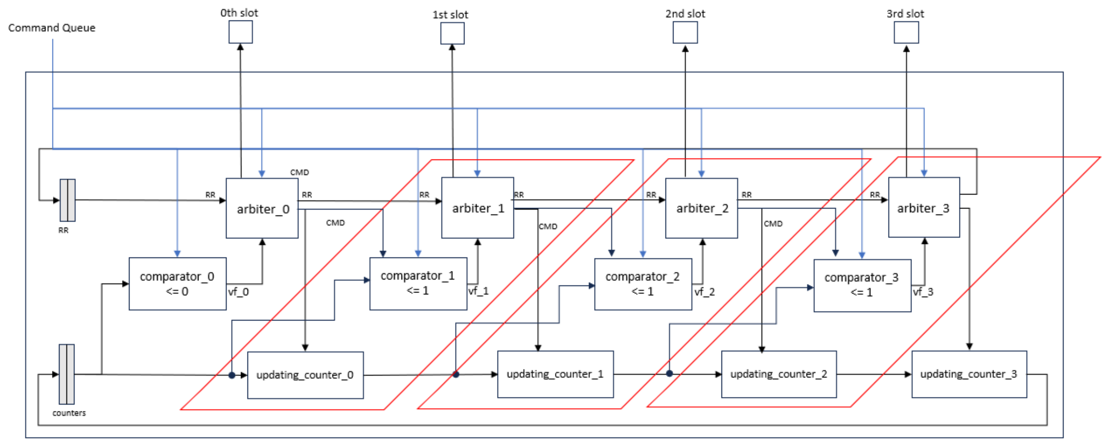


Figure 4.9: The improved design of the command scheduler version 1 .

Due to the timing constraints, outlined earlier, if any type of command from the front of the command queue corresponding to bank b_n is picked for slot 3, definitely none will be picked again from the same queue in slot 0 of the next clock cycle. Moreover, if any ACT is picked in slot 3, no ACT from other command queues can be picked in slot 0 of the following cycle since t_{RRD_S} is 4. This implies that the last **updating_counter** block in Figure 4.9, **updating_counter_3** can operate in parallel with the first arbiter, **arbiter_0**. This results in the implementation shown in Figure 4.10. All comparators now compare against 1 while in the design of Figure 4.9, **comparator_0** compares against 0. By this

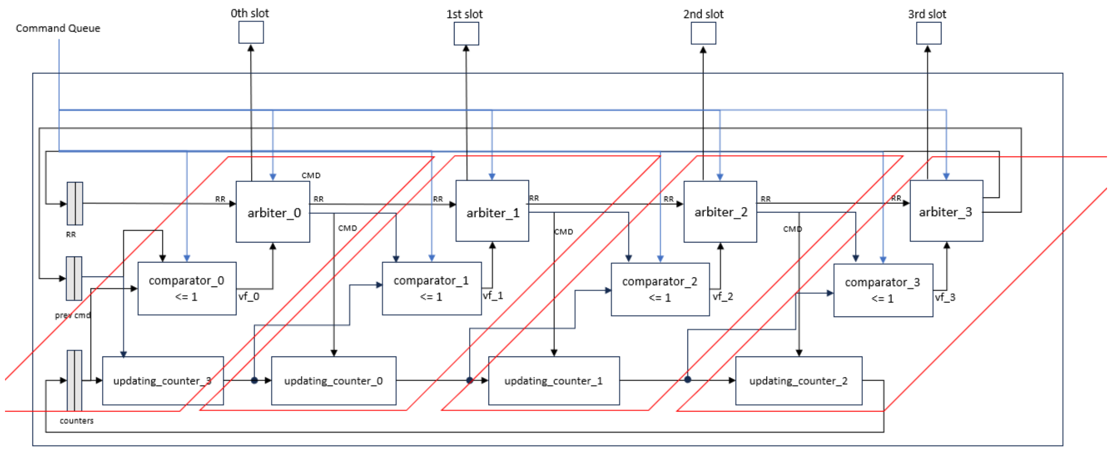


Figure 4.10: The best design of the command scheduler version 1.

approach all arbiters and time modules are operating in parallel (red parallelograms in Figure 4.10), and this results in the longest path delay of either $4t_{tm}$ or $4t_{arb} + 4t_{cmp}$ depending on the values of t_{tm}, t_{arb}, t_{cmp}.

4.5.3 Version 2

In Command Scheduler Version 2, the scheduling task is further parallelized by considering the type of commands. This decision is primarily influenced by the nature of timing constraints. Specifically, the issuance of an ACT does not impose any intra-bank constraints related to CAS on other banks, and vice versa. This implies that the arbitration of ACT and CAS can be independently parallelized. In Version 1, there is a dedicated arbiter for each slot, capable of selecting any type of command. In Version 2, however, there are four arbiters—two dedicated to selecting ACT commands and the other two dedicated to selecting CAS commands. Given the constraint imposed by Xilinx's PHY, which limits the placement of CAS commands to either slot 0 or slot 2 (as outlined in Section 3.2), for the sake of design simplicity, it is decided to place ACT commands either in slot 1 or slot 3. Given that, at most, one ACT and one CAS are selected in every clock cycle, two vacant slots remain for the placement of PRE commands. Priority encoders are employed for the selection of PRE commands, and these encoders differ in their highest priority index. Specifically, one of them assigns bank 0 the highest priority (**priority_lsb**), while the other designates the same bank as the lowest priority (**priority_msb**). These two enocders would select two different PRE commands if avaiable.

The structure of the second command scheduler is depicted in Figure 4.11. Initially, a set of comparators, akin to those in Version 1, is employed to ascertain the types of commands that are ready in their predefined slots. For example, **comparator_0** generates **CAS_vf_0** flags, indicating the readiness of CAS commands in slot 0 in all command queues. Additionally, **comparator_0** particularly identifies the presence of ready PRE commands at the front of command queues through **PRE_vf_0** and **PRE_vf_1** flags. Subsequently, these flags are utilized in priority encoders to facilitate the selection of PRE commands. This means that the readiness of PRE commands in the command queue is only determined through **comparator_0**. Each Round Robin arbiter has its own Round Robin pointer and by using the received flags from the corresponding comparator a command is selected. All these arbiters and priority encoders concurrently make their command selections. Ultimately, **resolve_logic** determines the final set of commands to be picked. Once the final commands are determined, they are transmitted to the **update_counter** block, where the counters are updated based on timing constraints for utilization in the subsequent clock cycle.

The comparators and arbiters are similar to those used in Version 1. However, the implementation of the timing module, **updating_counter** is changed. We next delve into the major changes in the timing module followed by a discussion of how **resolve_logic** works.

updating_counter

This module simultaneously receives a packet of four commands and, similar to the updating counter module in Version 1, imposes new timing constraints. However, in Version 1, the updating counter module would decrement all counters after each slot logic, whereas in the 2nd version, this is not the case since, at the end of the cycle, all commands are selected. Instead, Version 2 decrements unaffected counters by 4. The decrement value for those affected depends on the slot of the command influencing those counters. For instance, if an ACT is placed in slot 1, the counters affected by ACT-related timing constraints (intra-bank constraints such as t_{RCD}, and inter-bank constraints such as t_{RRD_L} or t_{RRD_S}), should be decremented by 3 instead of 4. If the ACT is selected in slot 3, they should be decremented by 1.

However, as discussed in the Counter Updating section of Section 4.5.2, the ACT and the CAS each can be impacted by one intra-bank and one inter-bank constraint in a single clock cycle. Implementing the **updating_counter** in this way requires more complicated logic since, when it comes to updating the value of counters associated with ACTs and

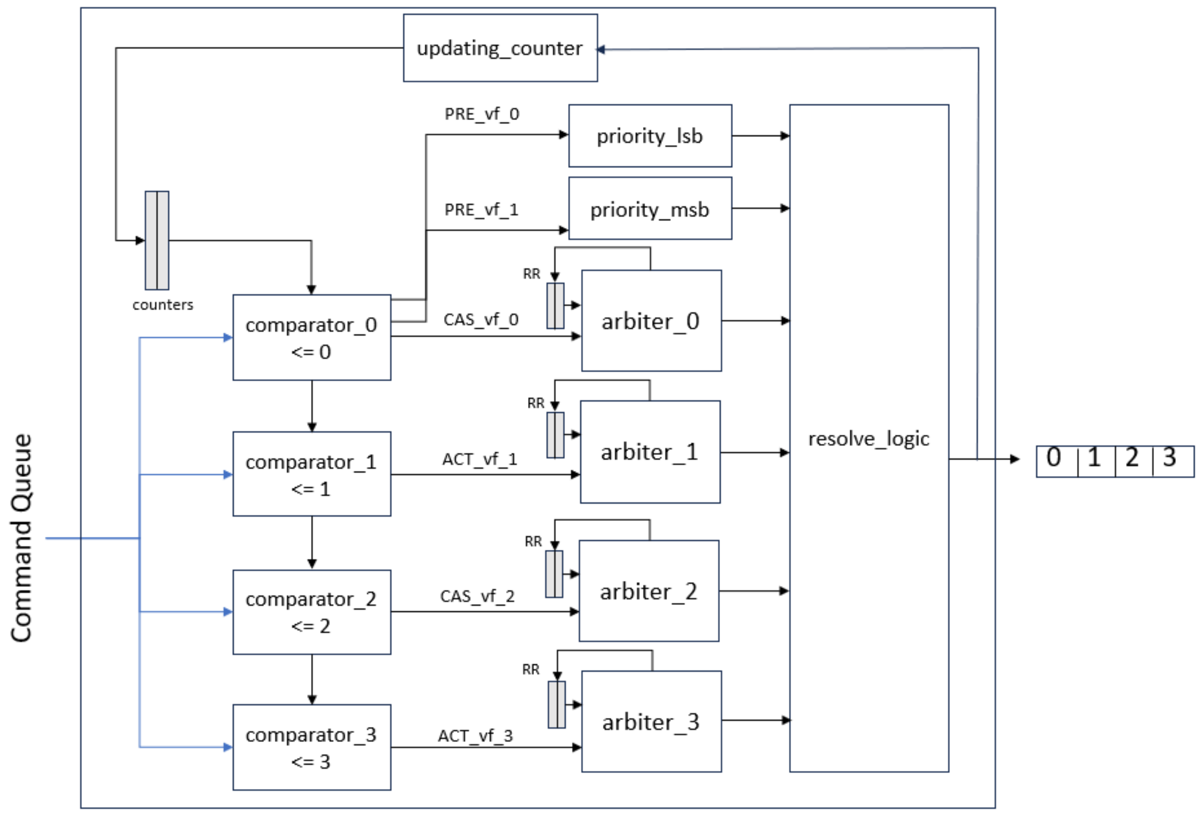


Figure 4.11: The command scheduler version 2 block diagram

CASs, three values should be compared: the current value of the counter, the intra-bank constraint, and the inter-bank constraint.

resolve_logic

resolve_logic comprises two distinct blocks—one for ACT and another for CAS. Essentially, it checks for the presence of any valid ready CAS command for slot 0. In the absence of such a command, it introduces the first PRE command to slot 0. If no PRE command is available, a NOP (null command) is inserted into the slot. This logic is mirrored for slot 2. Similarly, for ACTs, if a valid ready ACT command exists for slot 1, it is selected; otherwise, the 2nd PRE command, if available, is inserted into slot 1. The same logic applies to slot 3.

4.6 Maintenance Modules

As discussed in Section 4.1, we re-used modules **mc_periodic_read** and **mc_ref** from Xilinx's MIG to handle the periodic read required by MIG's PHY and execute the refresh process. When a refresh is due, the **mc_ref** module signals the request scheduler to halt its selection flow. The module waits until all the commands currently within the command queues are issued, and once the command queues are empty, they notify the **mc_ref** module that refresh can safely start. After the refresh, the page table also invalidates all its entries. For the periodic read, one of the command generators generates only CAS or PRE, ACT, and CAS and passes them to the command queue. The decision to generate only CAS depends on whether there is any valid open row in the page table. If there are multiple valid opened rows, the command generator with the least corresponding bank number will generate the CAS. If there is no valid opened row in the page table at all, PRE, ACT, and CAS with an address from the previously issued request will be generated by the command generator of bank 0. When the CAS of the periodic read is selected by the command scheduler, a specific signal will assert in **cal_top**, indicating that the issued CAS is intended for the periodic read. Once the periodic read finishes in PHY, **cal_top** notifies **mc_periodic_read** to resume the regular flow.

Chapter 5

Evaluation

We next present a comprehensive experimental evaluation of our memory controller with two versions of the command scheduler, MC Ver 1, and MC Ver 2 as well as Xilinx's MIG. In Section 5.1, we first discuss the various validation methods that we applied to verify our memory controller design. We then explain our experimental framework in Section 5.2. Then, in Section 5.3, using synthetic traces we explore the behavior of the Xilinx MIG. Subsequently, we compare the performance of both versions of MC and MIG under various stress scenarios using synthetic traces. These traces encompass scenarios featuring only open requests (representing best-case performance), only close requests, and a combination of open and close requests. Next in Section 5.4, we compare MC Ver 1, and MIG based on traces derived from an actual benchmark suite. Lastly, in Section 5.5 we report the the maximum frequency of first, and second command scheduler designs along with their LUT utilization.

5.1 Verification

The design of the DDR4 memory controller is a complex project that encompasses several modules. Throughout the implementation phase, standalone testbenches were developed to verify the correct functionality of individual modules, including the request scheduler, the command queue, and the command scheduler. However, the verification process did not conclude there. After integrating all these modules, further verification was performed on the system. Xilinx provides a Verification IP that is based on AXI protocol, called AXI VIP. AXI VIP is generally used to generate various patterns of AXI read, and write requests. To verify the integrated project, we employed the AXI Interface in the MIG (refer to Section 3.4) along with AXI VIP as a request generator. We performed various experiments through AXI VIP. Initial tests involved issuing a single read and a write request to validate that the DDR4 memory is functioning correctly, and the memory controller sends commands to the memory model. Subsequently, various patterns were implemented to perform more extensive design verification. In these tests, AXI VIP first issues a portion of write requests followed by read requests with the same addresses as write requests. This would confirm whether the integrated design functions properly and all data are written successfully. We conducted experiments with up to 10,000 read and write requests. Furthermore, these tests were conducted with different numbers of read-to-write switching. All test patterns were successfully passed.

5.2 Simulation Framework

To more easily test the memory controller's performance with a variety of traces representing both stress scenarios and real benchmarks, we opted to establish a suitable simulation framework that can issue requests in a controlled manner. This framework is designed to support multiple concurrent cores, with each core's requests described by a trace. Each line in the trace file contains information about a single request, including: 1) the address of that request, 2) the request type, which is either read or write, and 3) the clock cycle at which the request is made assuming zero memory time.

To simulate cores issuing multiple concurrent requests, each core can be configured separately to issue a variable number of out-of-order requests. Once a core issues all the concurrent requests it can, it will be blocked until at least one of the requests completes. When a request completes, its latency is added to the next request's start cycle, and this new start cycle becomes the time at which the next request will be issued.

The evaluation platform has been implemented using an Object-Oriented approach in System Verilog. As depicted in Figure 5.1, each instance of the Requestor class reads a trace file. Whenever the corresponding cycle of the request is equal to or greater than the simulation cycle counter, the requestor passes the request to the Round Robin arbiter. The arbiter selects one of the ready requests according to the Round Robin pointer, and subsequently, the request is translated into the User Interface address standard. As discussed earlier in Section 3.3, the User Interface accepts one request at a time. For this reason, the Round Robin arbiter is used to select only one request among cores. Furthermore, there is a class called Scoreboard, which logs the origin, address, command type, issuance, and finish time of each request.

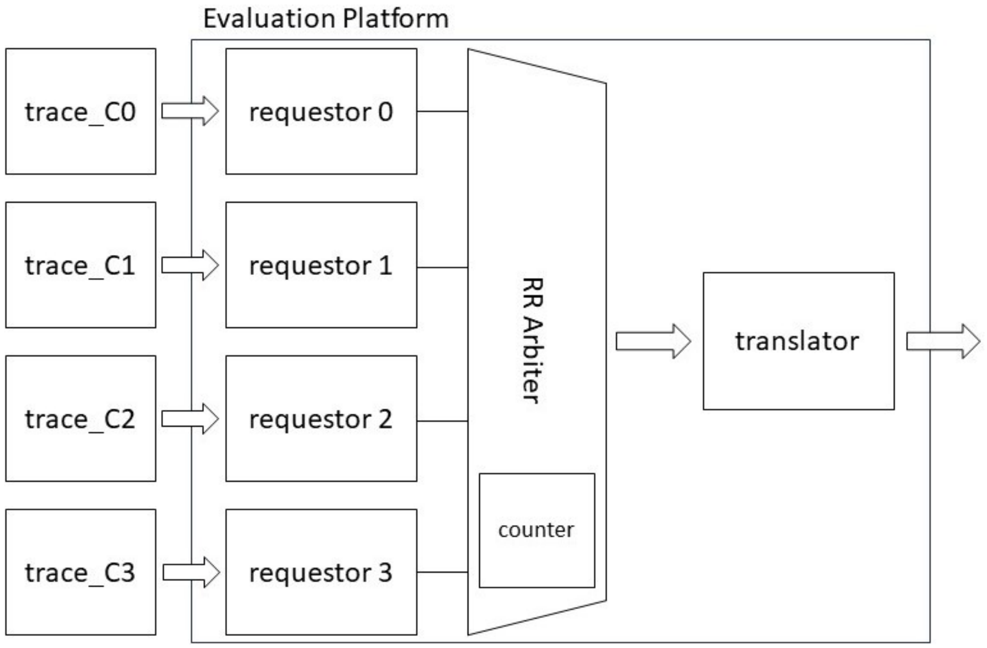


Figure 5.1: The evaluation platform block diagram.

The memory module is modeled using MIG's implementation. Unless otherwise specified, DDR4 2666 x8 is employed for all the experiment runs, and the total number of cores is set to 8, where core 0 is the core under analysis denoted as $core_{ua}$. This approach ensures uniformity across all designs, as they operate with identical memory devices, utilize the same type of traces, and share a common User Interface.

Now let us compute the theoretical maximum throughput of DDR4 2666 model. In this model, the data rate is 2666 MT/s, which means the main clock frequency is 1333 MHz (750 ps). Per the configuration, the device width is 8 bits, and since we have a burst of 8, the total data width would be 64 bits, and as discussed in Section 2.1.2, the 8 bursts are transmitted in 4 clock cycles. In ideal conditions, the throughput will be $\frac{8\ bytes}{(750\ ps)\times 4} = 2666$ MB/s if the memory controller fully utilizes the data bandwidth.

5.3 Synthetic Benchmarks

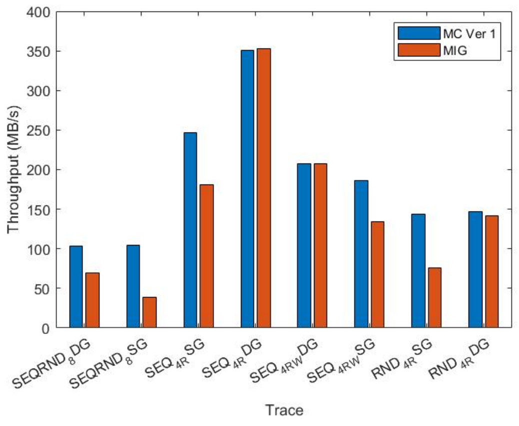
A set of synthetic experiments are conducted to achieve the following objectives: 1) Explore the behavior of MIG, 2) Validate our memory controller design, and 3) Compare the performance of our memory controller with that of MIG. We generated two categories of traces. The first category involves sequential accesses, resembling open requests hitting the same row. The second category comprises random requests targeting different rows within the same bank and bank group. All cores are able to issue 4 concurrent requests to stress the system.

5.3.1 Latency

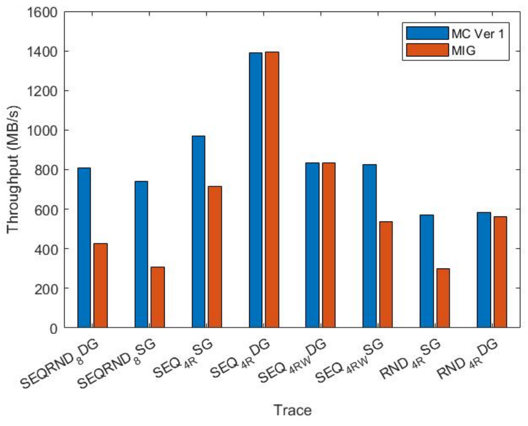
To evaluate our memory controller design in terms of request latency, we conducted experiments with a single core issuing isolated requests based on a synthetic trace file. We are interested in computing the latency for open/close read requests as well as open/close write requests, a total of 4 experiments. In this setup, we can compute the end-to-end latency of a request traversing through the memory controller without being interfered with any other interfering requests from the same core. The end-to-end latency is defined as the time difference between when a request is issued by a core and when the corresponding data is taken to/from the UI. First, we begin with open read-only and write-only requests; the latency is 18, and 7 clock cycles respectively. This shows, that write requests finish sooner than read requests. We believe that this significant difference in latency is due to the extra clock cycles needed for the data to traverse the PHY modules. Subsequently, we conducted experiments for close read-only and close write-only requests which resulted in a latency of 27 and 18 clock cycles. All these values are the minimum latency that a single request suffers.

5.3.2 MIG vs. MC Ver 1

In the initial experiment, we set up a configuration where 4 cores each issues purely sequential accesses, and each core is assigned to a unique bank. Given that the requests are open, only CAS commands will be issued, and only the timing constraints of t_{CCD_L} and t_{CCD_S} are imposed. The traces $SEQ_{4R}DG$ and $SEQ_{4R}SG$ correspond to this setup, (SEQ refers to sequential, 4 shows the number of cores, R is the request type, read). In the former, the four banks of the four cores are in different bank groups (shown by DG), while in the latter, the four banks of the four cores are in the same bank group (shown by SG). For this experiment, we present both the throughput of $core_{ua}$, and the total throughput of the system. As illustrated in Figure 5.2a, the throughput of $core_{ua}$ in both MC Ver 1 and MIG is equal in experiment $SEQ_{4R}DG$. This equality stems from the fact that MIG applies parallelism over bank groups, and since the banks are in different bank groups, MIG fully leverages bank group parallelism. Similarly, our memory controller arbitrates over banks, resulting in leveraging bank parallelism. However, the throughput of MC Ver 1 in setup $SEQ_{4R}SG$ drops by 30% compared to setup $SEQ_{4R}DG$, which is the result of long inter-bank constraints as four banks are in the same bank group, and the issuance of CAS to any of these banks imposes t_{CCD_L}. Moreover, there is a 50% reduction in the throughput of MIG. The reason for this is that all requests go into a single Group FSM, and the queues inside this Group FSM frequently become full which results in a significant drop in throughput.



(a) The throughput of $core_{ua}$ comparison amongst eight experiments for both MC Ver 1, and MIG.



(b) The total throughput of MC Ver 1, and MIG comparison amongst eight synthetic benchmarks.

Figure 5.2: MIG performance VS. MC Ver 1 performance

As discussed in Section 2.1.3, there is a penalty when switching from read to write or vice versa. This effect is demonstrated in the trace $SEQ_{4RW}DG$, where four cores sequentially access unique banks in different bank groups with random request types. If we compare the throughput of $core_{ua}$ in this trace with that of $SEQ_{4R}DG$, a significant drop is observed, primarily attributed to the read-to-write and write-to-read switching timing constraint. Similarly, the reported throughput for trace $SEQ_{4RW}SG$ is less than that of $SEQ_{4RW}DG$ due to the same reasons as before.

Next, we continue the experiment with 4 cores, each issuing close requests with random rows, with each core assigned to a unique bank. As illustrated in Figure 5.2a, the throughput of $core_{ua}$ for memory controllers in traces $RND_{4R}DG$ and $RND_{4R}SG$ is notably lower than that of traces $SEQ_{4R}DG$ and $SEQ_{4R}SG$. This discrepancy is mainly attributed to the difference between open request latency and close request latency. However, when comparing the bars of these two random traces, the throughput of $core_{ua}$ in MC Ver 1 remains almost the same between DG and SG whereas it changes in MIG since there is no exploitable parallelism among requests to different banks in the same bank group.

The last experiment combines sequential and random requests. In this scenario, there are 8 cores, forming four pairs, with each pair targeting a unique bank. In each pair, the first core sequentially accesses a unique bank, and the second core issues random close requests to the same bank. In other words, the requests of the second core interfere with those of the first core. This setup, denoted as $SEQRND_8SG$, is conducted for all pairs targeting different banks in the same bank group. For different banks in different bank groups, this setup is denoted as $SEQRND_8DG$. As shown in Figure 5.2a, in trace $SEQRND_8DG$, the throughput of $core_{ua}$ in MIG is less than that of MC Ver 1. This suggests that MIG does not prioritize open requests of $core_{ua}$ over the close requests of its pair, and the result suggests that MIG does not implement FR-FCFS arbitration. Similar to previous experiments, when the cores switch to different banks in the same bank group, the throughput in MIG decreases slightly due to the same reasons mentioned previously. The total throughput of both memory controllers is shown in Figure 5.2b, and the same patterns as in Figure 5.2a are observed.

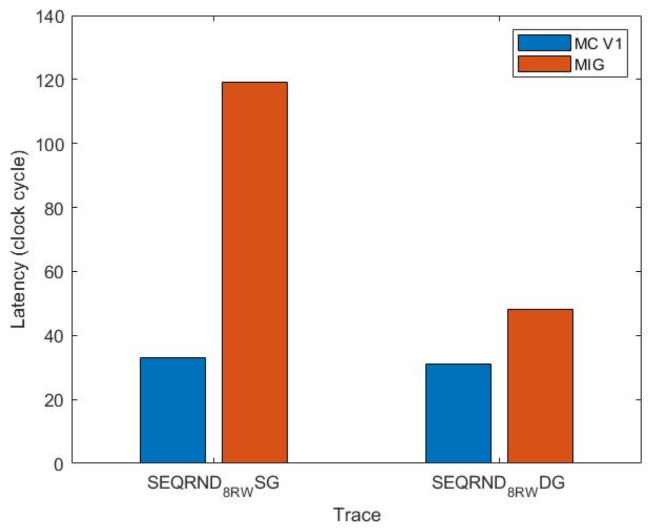
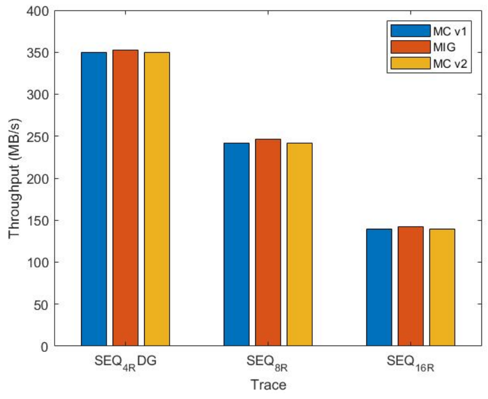


Figure 5.3: The latency comparison of two traces $SEQRND_8SG$, and $SEQRND_8DG$ between MC Ver 1, and MIG.

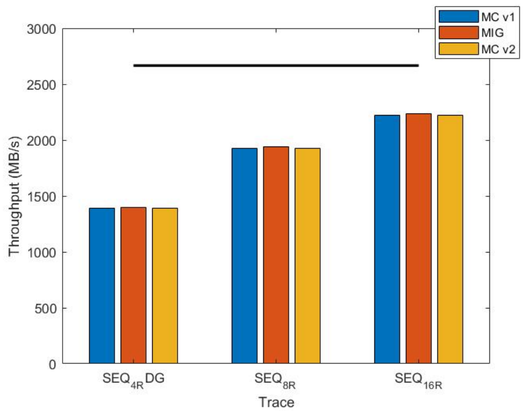
Previously, we compared the throughput of our memory controller (MC V1) with MIG for traces $SEQRND_8SG$ and $SEQRND_8DG$. Now, let us delve into the latency of requests for these traces in both MC V1 and MIG. As depicted in Figure 5.3, the average latency of requests for both traces in MC V1 is nearly equal. This is because our memory controller employs bank arbitration, effectively managing request access to the banks. Conversely, the average latency of requests for trace $SEQRND_8SG$ is significantly higher in MIG compared to MC V1. This is primarily attributed to MIG's parallelization over bank groups. When all requests target the same bank group, interference among requests increases, resulting in elevated latencies. However, when cores target different bank groups, latencies decrease, as expected.

5.3.3 Scalability

In this section, we will evaluate the performance of MC Ver 1, MC Ver 2, and MIG as the number of cores increases from 4 cores to 16 cores. In Section 5.2, we showed that the maximum theoretical throughput for DDR4-2666 is 2666 MB/s. To achieve the maximum data bandwidth in our experiments, we used sequential traces as follows 1) Traces SEQ_{8R}, and SEQ_{16R} contain 8 cores, and 16 cores respectively, each issuing sequential read requests to a unique bank. 2) Traces SEQ_{8W}, and SEQ_{16W} include 8 cores, and 16 cores respectively, each issuing sequential write requests to a specific bank.



(a) The throughput of $core_{ua}$ comparison for sequential read requests, the number of cores changes from 4 to 16.



(b) The total throughput of MC Ver 1, MC Ver 2, and MIG comparison for sequential read requests, the number of cores changes from 4 to 16.

Figure 5.4: The impact of the number of cores on throughput for sequential read accesses.

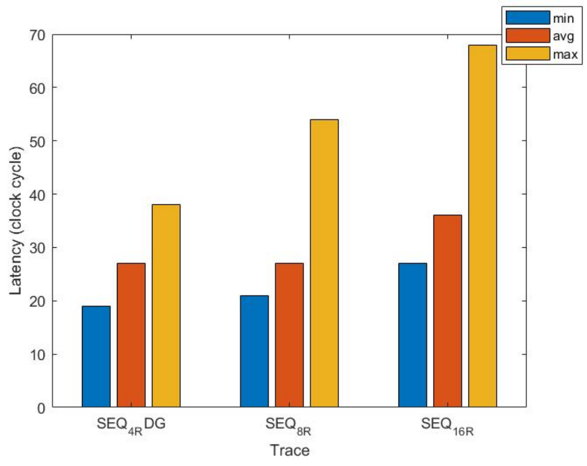


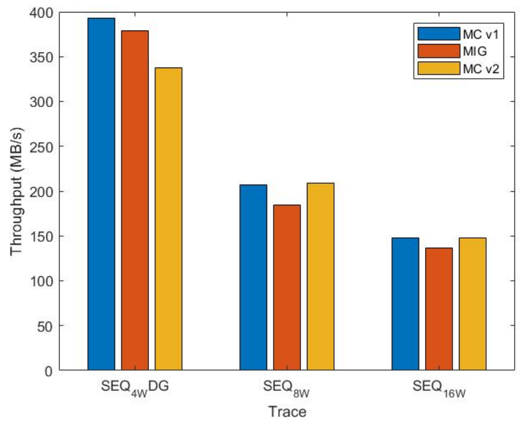
Figure 5.5: The minimum, average, and maximum latency of sequential read accesses for MC Ver 1.

The throughput of $core_{ua}$ for traces $SEQ_{4R}DG$, SEQ_{8R}, and SEQ_{16R}, are shown in Figure 5.4a. As the number of cores increases, the interference among banks also increases, leading to a drop in the reported throughput. Conversely, when the number of cores increases, the total throughput of all memory controllers increases, primarily because all banks are utilized. However, the design cannot reach the ideal theoretical bandwidth of 2666 MB/s due to the arbiter used in the test platform, as discussed in Section 5.2, which increases the time it takes for a new request from a core to arrive at the memory controller. This is shown in Figure 5.4b, the black line is the theoretical throughput.

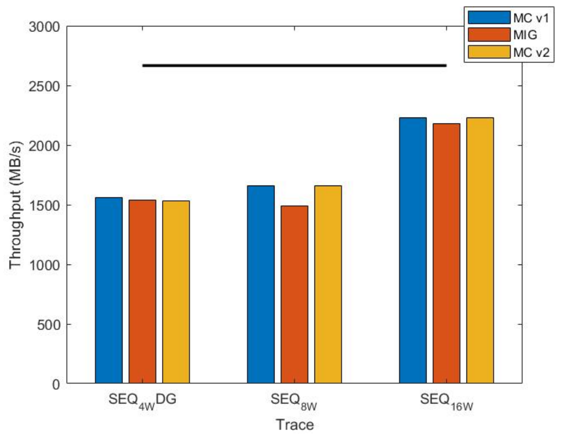
The throughput of $core_{ua}$ for sequential write accesses is illustrated in Figure 5.6a. The reported throughput for sequential write requests, $SEQ_{4W}DG$, SEQ_{8W}, and SEQ_{16W} are higher than that of sequential read requests, mainly because write requests are finished sooner than read requests as discussed in Section 3.3. We computed the average latency for $SEQ_{4W}DG$, which is 16 cycles, and the average latency for $SEQ_{4R}DG$, which is 27 cycles. These values confirm the fact that write requests finish sooner than read requests. Also, these values are higher than what is determined only by the timing constraints, the extra latency is added by PHY, and memory controller logic.

Moreover, we computed the latency of sequential read accesses of the same traces, $SEQ_{4R}DG$, SEQ_{8R}, and SEQ_{16R}. The minimum, the average, and the maximum latency are reported in Figure 5.5. As the number of cores increases, there is more interference among cores, and consequently, the latency is increased.

Now, we evaluate the performance for the most interference traces. In this configura-



(a) The throughput of $core_{ua}$ comparison for sequential write requests, the number of cores changes from 4 to 16.



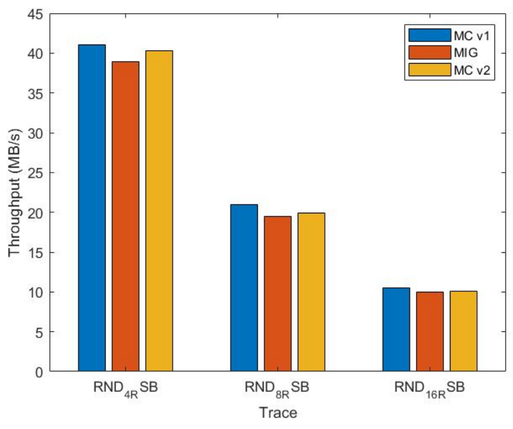
(b) The total throughput of MC Ver 1, MC Ver 2, and MIG comparison for sequential write requests, the number of cores changes from 4 to 16.

Figure 5.6: The effect of the number of cores for sequential write accesses.

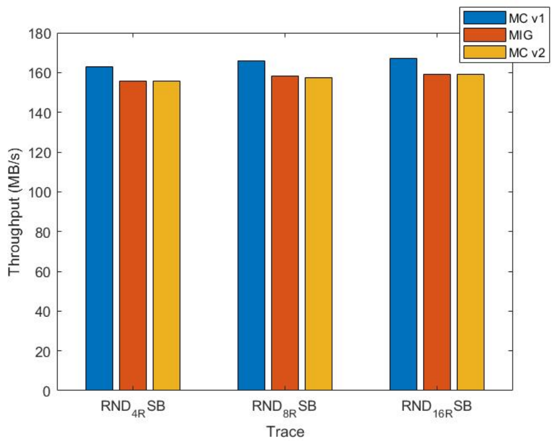
tion, all cores issue close requests to the same bank as $core_{ua}$, and thus, all interfere with $core_{us}$. First, we used traces $RND_{4R}SB$, $RND_{8R}SB$, and $RND_{16R}SB$, with 4, 8, and 16 cores respectively each issuing close read requests to the same bank (SB). As the number of cores increases, interference also rises, decreasing the throughput of $core_{ua}$, Figure 5.7a. However, the total throughput for different numbers of cores remains the same since all requests are closed, and the simulation time linearly increases with the number of cores, leading to a constant total throughput. This is shown in Figure 5.7b. Likewise, an experiment with traces $RND_{4W}SB$, $RND_{8W}SB$, and $RND_{16W}SB$ are conducted with 4, 8, and 16 cores respectively, each issuing close random write requests to the same bank. The reported throughput of $core_{ua}$, and total throughput are shown in Figure 5.8a, and Figure 5.8b respectively.

5.3.4 MC Ver 1 vs. MC Ver 2

In this section, we demonstrate that MC Ver 1 outperforms MC Ver 2 using the same synthetic benchmarks used in the previous section. In the experiment with sequential requests both MC Ver 1 and MC Ver 2 illustrated in Figure 5.6, and Figure 5.4, both MCs have the same throughput since the requests are open and only CAS commands are issued.

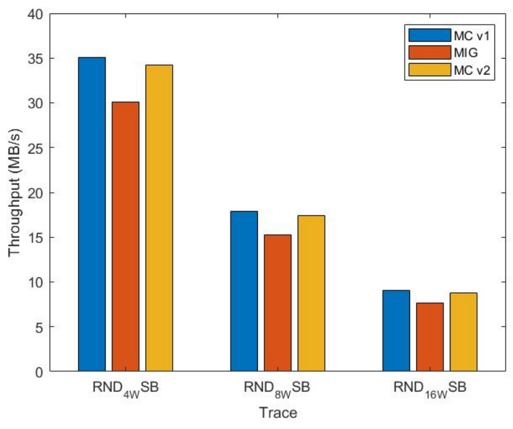


(a) The throughput of $core_{ua}$ comparison for most interfering case, close read requests to the same bank, the number of cores change from 4 to 16.

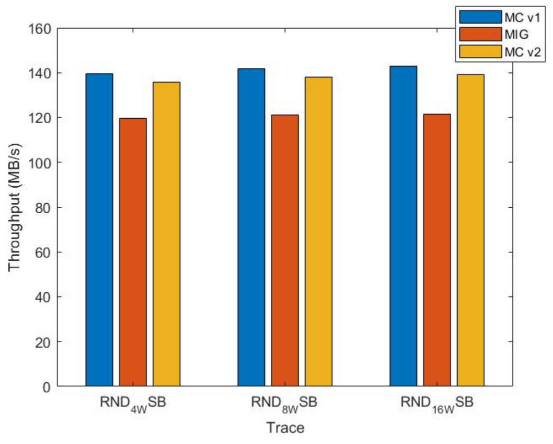


(b) The total throughput of MC Ver 1, MC Ver 2, and MIG comparison for most interfering case, close read requests to the same bank, the number of cores change from 4 to 16.

Figure 5.7: The effect of the number of cores for close read requests.



(a) The throughput of $core_{ua}$ comparison for most interfering cases, close write requests to the same bank, the number of cores change from 4 to 16.



(b) The total throughput of $core_{ua}$ comparison for most interfering cases, close write requests to the same bank, the number of cores change from 4 to 16.

Figure 5.8: The effect of the number of cores for close write requests.

However, in the experiment with close requests, Figure 5.8, and Figure 5.7, the throughput of $core_{ua}$ in MC Ver 1 is only slightly higher than the throughput of the same core in MC Ver 2. This marginal difference is attributed to the variations in the scheduling policies of the command schedulers in these memory controllers. As explained in Section 4.5.3, the command scheduler of MC Ver 2 imposes more constraints in selecting ACT and PRE commands. It only puts ACT on slot 1 and slot 3, and only issues 2 PREs.

5.4 EEMBC Benchmarks

In this section, we aim to assess the performance of our memory controller and MIG in the presence of requests generated by a processor. To achieve this, we utilize the EEMBC benchmark suite [34]. The benchmarks are executed on MacSim [24] with the following configuration: Superscalar x86 cores clocked at 1GHz with bypassed cache to maximize the stress on DRAM memory. The execution of these benchmarks on MacSim results in trace files containing requests issued from the cores to DRAM. We leverage these traces in our experiments to evaluate the performance of our memory controller. In all experiments, we execute one of the EEMBC benchmark traces in $core_{ua}$, while other interfering cores run either EEMBC traces or synthetic benchmarks.

In the first experiment, the single-core system only contains $core_{ua}$ solely running EEMBC traces without any interfering cores. In this configuration, $core_{ua}$ issues only one request at a time, waiting for a request to be completed before sending the next one. Since $core_{ua}$ is the sole core in this experiment and there are no interfering requests, the performance of MC Ver 1 and MIG are nearly identical, shown in Figure 5.9. This experiment serves to validate the integration of our memory controller design, with MIG acting as a reference model. It is noteworthy that the throughput of $core_{ua}$ in all traces ranges from 80 to 100 MB/s, with one exception, **puwmod** trace. This exception suggests that the number of closed requests in this trace is higher than in the other traces, impacting the subsequent experiment.

Next, we stress the memory controller by running EEMBC traces on $core_{ua}$ along with interfering cores that run synthetic benchmarks. These synthetic benchmarks are crafted to target the banks that are mostly hit by EEMBC traces. The objective of these experiments is to assess the performance of MC Ver 1 and MIG, comparing their throughput. In this configuration, $core_{ua}$ executes EEMBC traces and issues a new request once the previously issued request completes. Meanwhile, the interfering cores, with a total number of 7, run synthetic benchmarks, with each interfering core sequentially issuing read requests to a unique bank. $core_{ua}$ in the first version of our memory controller outperforms the

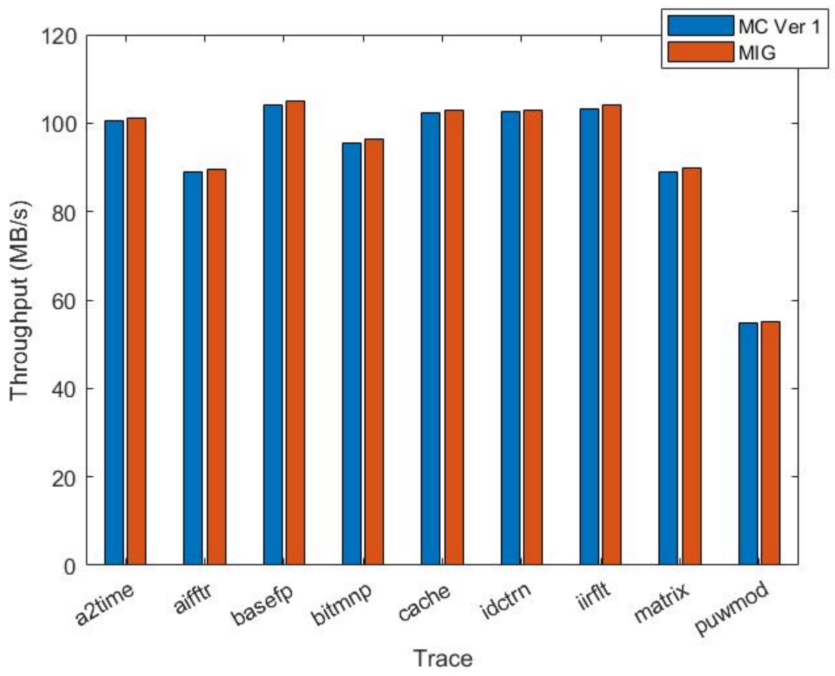
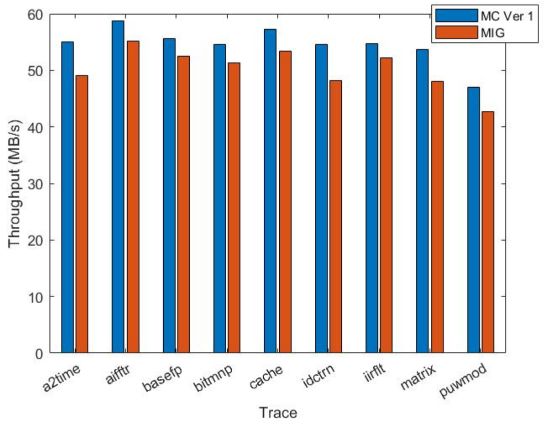


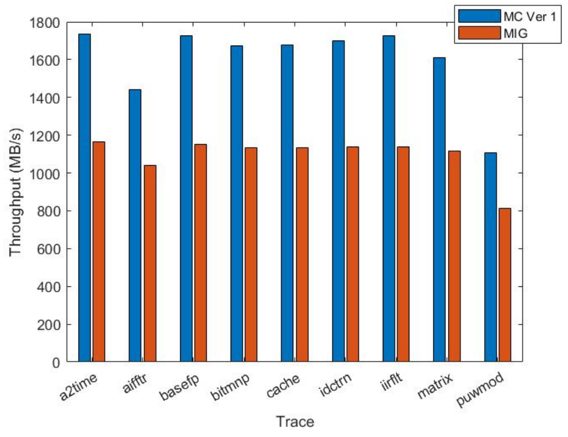
Figure 5.9: The throughput of $core_{ua}$ comparison amongst all EEMBC traces. Single-core system.

performance of the same core in MIG, as shown in Figure 5.10a. This is primarily attributed to the utilization of parallelism over banks in our memory controller, whereas MIG employs parallelism over bank groups. Furthermore, our memory controller exhibits roughly a 50% increase in the total system throughput compared to MIG, Figure 5.10b. As mentioned earlier, the total throughput of trace **puwmod** is significantly lower compared to the total throughput of the other traces. Upon closer examination of the log output from this trace, it was observed that trace **puwmod** took much longer to complete. This observation supports the conclusion that trace **puwmod** includes more close requests compared to the other EEMBC traces.

In the final experiment, eight cores are each running an EEMBC trace. Since our benchmark suit consists of 9 traces but we have 8 cores, we neglected the last trace **puwmod** in this setup. Each core issues a single request once its previously issued request completes, and the simulation ends when any of the cores finishes. The throughput of $core_{ua}$ is illustrated in Figure 5.11a, with each trace indicating the specific trace that $core_{ua}$ is running. The first version of our memory controller outperforms MIG in all traces, showcasing



(a) The throughput of $core_{ua}$ comparison amongst nine experiments for both MC Ver 1, and MIG. In each experiment, $core_{ua}$ runs the trace in the X-axis.



(b) The total throughput of both memory controllers amongst nine experiments. In each experiment, $core_{ua}$ runs the trace in the X-axis.

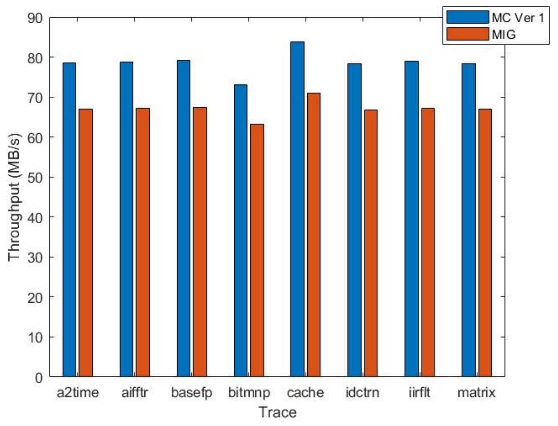
Figure 5.10: EEMBC benchmark chart with the following configuration: 1) $core_{ua}$: EEMBC trace 2) interfering cores: synthetic benchmark.

around a 20% improvement. The total throughput is depicted in Figure 5.11b, where once again, our memory controller exhibits a 20% increase in total throughput compared to MIG.

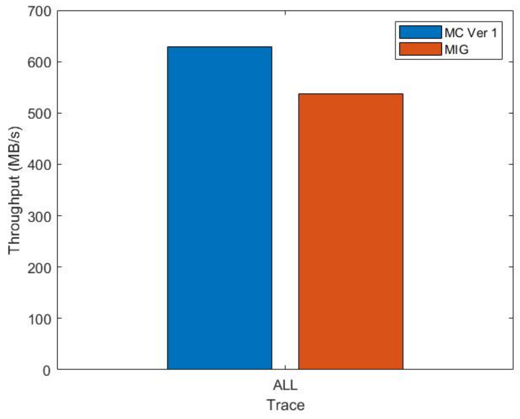
5.5 Frequency and Hardware Utilization

The implementation details of both command schedulers were discussed in Sections 4.5.2 and 4.5.3. The primary motivation for designing the second command scheduler was the limitation of the first version in achieving a frequency comparable to MIG's capability. MIG can offer a design with a maximum frequency of 333 MHz. FPGA board latency depends on various factors, one of which is Logic Level. It indicates how many LUTs are connected sequentially in the implementation of a logic block. A higher number of logic levels leads to increased logical path delay and a reduced maximum frequency.

We designed our memory controller to be implemented on Xilinx Zynq Ultrascale+ ZCU102. Table 5.1 presents the maximum frequency, number of logic levels, and utilization of resources, including LUTs and registers, for both designs—command scheduler version



(a) The throughput of $core_{ua}$ comparison amongst eight experiments for both MC Ver 1, and MIG. In each experiment, $core_{ua}$ runs the trace in the X-axis.



(b) The total throughput of both memory controllers amongst eight experiments. Since the same traces are permuted among cores, the total throughput of all eight experiments is identical.

Figure 5.11: EEMBC benchmark chart with the following configuration: 1) $core_{ua}$: EEMBC trace 2) interfering cores: EEMBC trace.

1 and command scheduler version 2. We did not utilize any BRAMs or DSPs in the design of both command schedulers. The first version of command scheduler demonstrates nearly a 60% improvement in maximum frequency compared the second version. Additionally, the second command scheduler occupies fewer Logic Levels, approximately two-thirds of that of version 1. In terms of resource utilization, MC Ver 1 employs more resources than MC Ver 2.

Now, let us discuss the maximum frequency. The bottleneck of our memory controller design is the command scheduler, although we can still implement the design on FPGAs for lower frequencies than the maximum frequency for the following reason. The frequency at which the memory controller should operate depends on the speed grade of DDR4 memory due to the 4:1 clock ratio discussed in Section 3.2. For instance, if the DDR4 memory module on the FPGA board is DDR4 1600, the memory controller should issue a pack of 4 commands at 200 MHz rather than 300 MHz. The speed grades of DDR4 are: 1600, 1866, 2133, 2400, 2666, 2933, and 3200 MT/s, resulting in the frequency of the memory controller ranging from 200 to 400 MHz. MIG supports up to 333 MHz, while our current command scheduler design supports up to 180 MHz, as shown before in Table 5.1. However if we can not reach the nominal frequency of DDR4, based on the observation we had, we

	Command Scheduler Ver 1		Command Scheduler Ver 2	
	Banks = 8	Banks = 16	Banks = 8	Banks = 16
Max Freq (MHz)	115	105	181	170
Logic Level	28	30	17	19
LUT	3756	7862	3747	7242
Register	252	463	212	408

Table 5.1: The utilization, and timing report for both command schedulers with the total number of 8 banks, and 16 banks.

could reduce the clock frequency of PHY on the FPGA board. We dropped the frequency of DDR4 2400 on ZCU102, which has the nominal frequency of 1200 MHz, to 600 MHz, and it was still functional. However, this technique comes with a downside: since the clock frequency is reduced by 50%, the throughput would decrease by almost 50%. For these reasons, we aimed to reach at least the frequency of 200 MHz in our memory controller design.

Chapter 6

Conclusion

As highlighted in Chapter 1, DDR4 memories serve as a bottleneck in the performance of many computer systems, and the efficiency of memory controllers significantly influences this performance. Given the lack of open-source ASIC implementations of memory controllers, FPGAs offer a flexible platform for implementing various memory controller designs due to their programmable logic. Major FPGA vendors such as Intel and Xilinx provide memory controller designs for their users. In our case, we chose to implement our design on Xilinx FPGAs. Xilinx's MIG, while available, may not provide optimal performance, and there is a limited number of high-performance DDR4 memory controllers implemented on FPGAs. Our goal was to design a high-performance DDR4 memory controller on Xilinx's FPGAs that could surpass the performance of Xilinx's MIG. Our memory controller implements a FR-FCFS arbitration scheme as a front-end scheduler and employs Round Robin arbitration over banks for the back-end scheduler, implementing an open-page policy. It is important to note that we utilized Xilinx's MIG as a foundation for our design, reusing certain modules such as calibration logic and PHY logic. In Section 5.4, we demonstrated that our memory controller outperforms Xilinx's MIG in EEMBC benchmarks assuming operating at same frequency.

Our memory controller is available in two versions, each tailored to address specific performance considerations. Due to the frequency constraints on FPGA boards, both our memory controllers and MIG issue up to 4 commands in a single cycle. While MIG supports a maximum frequency of 333 MHz, the initial version of our memory controller is limited to a maximum frequency of 100 MHz. This limitation primarily stems from the design of the command scheduler in the first version.

In response to this constraint, the second version of our memory controller was developed, featuring a redesigned command scheduler that adheres to a Round-Robin policy over banks. This design enhancement led to a faster implementation, achieving 59% higher maximum fre quency than that of the first version. Notably, the key distinction between the two versions lies in their command scheduling capabilities. The first version can issue up to 4 PRE commands and place ACT commands in any slot within a single cycle. In contrast, the second version is more constrained, only allowing ACT commands in the 1st and 3rd slots and permitting the issuance of two PRE commands in a single clock cycle. While these additional constraints slightly impact the performance of the second version, they contribute to a faster design with a higher frequency.

Our entire implementation is configurable and parameterized, providing a versatile testbench platform with a realistic setting that incorporates refresh, initialization, and calibration. Leveraging MIG's design as a foundation, our memory controller models various DRAM speed grades with different data widths, enhancing its adaptability and applicability.

Looking forward, we anticipate that there are additional optimization opportunities for the command scheduler design, which could result in a higher maximum frequency for the overall design. Currently, our memory controller has been validated and tested solely in simulation, despite being synthesizable. In the future, we plan to implement the design on various FPGA boards equipped with different DDR4 devices to verify the functionality of our memory controller and assess its real-world performance.

From the inception of the project, we have designed our memory controller with extensibility in mind, enabling future modifications and potential developments of real-time memory controllers. Our ultimate objective is to create a real-time memory controller, building upon the foundation laid by our high-performance memory controller. Additionally, we aspire to design a pair-like memory controller, akin to DUETTO [25], leveraging the lessons learned and innovations made during the course of this project.

References

[1] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, aug 2011.

[2] Andrew Mason Cheong KUN, Shaolei QUan. A power-optimized 64-bit priority encoder utilizing parallel priority look-ahead. In *ISCAS*, 2004.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. *SIGCOMM Comput. Commun. Rev.*, 19(4):1–12, aug 1989.

[4] Leonardo Ecco and Rolf Ernst. Improved dram timing bounds for real-time dram controllers with read/write bundling. In *2015 IEEE Real-Time Systems Symposium*, pages 53–64, 2015.

[5] H. Kim et al. Macsim: Simulator for heterogeneous architecture, 2012.

[6] H. Lee et al. Design of non-contact 2 gb/s i/o test methods for high bandwidth memory (hbm). In *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, 2016.

[7] J. Koo et al. Small-area high-accuracy odt/ocd by calibration of global on-chip for 512 m gddr5 application. In *Proc. IEEE Custom Intergr. Circuits Conf. (CICC)*, 2009.

[8] N. Chatterjee et al. Usimm: the utah simulated memory module, 2012.

[9] Michel Goossens, Frank Mittelbach, and Alexander Samarin. *The LaTeX Companion*. Addison-Wesley, Reading, Massachusetts, 1994.

[10] Mohamed Hassan, Hiren Patel, and Rodolfo Pellizzoni. A framework for scheduling dram memory accesses for multi-core mixed-time critical systems. In *21st IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 307–316, 2015.

[11] Mohamed Hassan and Rodolfo Pellizzoni. Bounding dram interference in cots heterogeneous mpsocs for mixed criticality systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2323–2336, 2018.

[12] Mohamed Hassan and Rodolfo Pellizzoni. Analysis of Memory-Contention in Heterogeneous COTS MPSoCs. In Marcus Völp, editor, *32nd Euromicro Conference on Real-Time Systems (ECRTS 2020)*, volume 165 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:24, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[13] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *2008 International Symposium on Computer Architecture*, pages 39–50, 2008.

[14] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154, 2014.

[15] Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA - 16 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[16] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 65–76, 2010.

[17] Yoongu Kim, Weikun Yang, and Onur Mutlu. Ramulator: A fast and extensible dram simulator. *IEEE Computer Architecture Letters*, 15(1):45–49, 2016.

[18] Donald Knuth. *The TEXbook*. Addison-Wesley, Reading, Massachusetts, 1986.

[19] Yogen Krishnapillai, Zheng Pei Wu, and Rodolfo Pellizzoni. A rank-switching, open-row dram controller for time-predictable systems. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 27–38, 2014.

[20] Leslie Lamport. *LaTeX — A Document Preparation System*. Addison-Wesley, Reading, Massachusetts, second edition, 1994.

[21] Shang Li, Zhiyuan Yang, Dhiraj Reddy, Ankur Srivastava, and Bruce Jacob. Dramsim3: A cycle-accurate, thermal-capable dram simulator. *IEEE Computer Architecture Letters*, 19(2):106–109, 2020.

[22] Micron. Jedec standard, 2012.

[23] Micron. Ddr4 sdram 8gb datasheet, 2015.

[24] Reza Mirosanlou, Danlu Guo, Mohamed Hassan, and Rodolfo Pellizzoni. Mcsim: An extensible dram memory controller simulator. *IEEE Computer Architecture Letters*, 19(2):105–109, 2020.

[25] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duetto: Latency guarantees at minimal performance cost. In *IEEE Design, Automation and Test in Europe Conference*, pages 1136–1141. IEEE, IEEE, 2021.

[26] Reza Mirosanlou, Mohamed Hassan, and Rodolfo Pellizzoni. Duomc: Tight dram latency bounds with shared banks and near-cots performance. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '21, New York, NY, USA, 2022. Association for Computing Machinery.

[27] Onur Mutlu and Thomas Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 146–160, 2007.

[28] Onur Mutlu and Thomas Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems. In *2008 International Symposium on Computer Architecture*, pages 63–74, 2008.

[29] Taesik Na, Yong Shim, Indal Song, Jeong-Kyoum Kim, Seokhun Hyun, Jun-Bae Kim, Jung-Hwan Choi, Chi-Wook Kim, Jung-Bae Lee, and Joo Sun Choi. A heterogeneous dual dll and quantization error minimized zq calibration for 30nm 1.2v 4gb 3.2gb/s/pin ddr4 sdram. In *2013 Symposium on VLSI Circuits*, pages C242–C243, 2013.

[30] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222, 2006.

[31] Ataberk Olgun, Juan Gómez Luna, Konstantinos Kanellopoulos, Behzad Salami, Hasan Hassan, Oguz Ergin, and Onur Mutlu. Pidram: A holistic end-to-end fpga-based framework for processing-in-dram. *ACM Trans. Archit. Code Optim.*, 20(1), nov 2022.

[32] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, and Mateo Valero. An analyzable memory controller for hard real-time cmps. *IEEE Embedded Systems Letters*, 1(4):86–90, 2009.

[33] Churoo Park, HoeJu Chung, Yun-Sang Lee, Jaekwan Kim, JaeJun Lee, Moo-Sung Chae, Dae-Hee Jung, Sung-Ho Choi, Seung young Seo, Taek-Seon Park, Jun-Ho Shin, Jin-Hyung Cho, Seunghoon Lee, Ki-Whan Song, Kyu-Hyoun Kim, Jung-Bae Lee, Changhyun Kim, and Soo-In Cho. A 512-mb ddr3 sdram prototype with c/sub io/ minimization and self-calibration techniques. *IEEE Journal of Solid-State Circuits*, 41(4):831–838, 2006.

[34] Jason Poovey. Characterization of the eembc benchmark suite., 2007.

[35] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Effective management of dram bandwidth in multicore processors. In *16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007)*, pages 245–258, 2007.

[36] S. Rixner. Memory controller optimizations for web servers. In *37th International Symposium on Microarchitecture (MICRO-37'04)*, pages 355–366, 2004.

[37] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens. Memory access scheduling. In *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)*, pages 128–138, 2000.

[38] Paul Rosenfeld, Elliott Cooper-Balis, and Bruce Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, 2011.

[39] Lavanya Subramanian, Donghyuk Lee, Vivek Seshadri, Harsha Rastogi, and Onur Mutlu. Bliss: Balancing performance, fairness and complexity in memory access scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):3071–3087, 2016.

[40] Zheng Pei Wu. Worst case analysis of dram latency in hard real time systems, master's thesis. university of waterloo, 2013.

[41] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. Worst case analysis of dram latency in multi-requestor systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 372–383, 2013.

[42] AMD Xilinx. Ultrascale architecture configurable logic block user guide, 2017.

[43] AMD Xilinx. Zynq-7000 soc and 7 series devices memory interface solutions v4.2, 2019.

[44] AMD Xilinx. Ultrascale architecture-based fpgas memory ip v1.4, 2022.

[45] AMD Xilinx. Ultrascale architecture libraries guide, 2022.

[46] Cong-Kha Pham Xuan-Thuan Nguyen, Hong-Thu Nguyen. A scalable high-performance priority encoder using 1d-array to 2d-array conversion. *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS*, 64(9), 2017.

[47] George L. Yuan, Ali Bakhoda, and Tor M. Aamodt. Complexity effective memory access scheduling for many-core accelerator architectures. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 34–44, 2009.

[48] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64, 2013.

[49] Hongguang Zhang. A 4 In *2023 8th International Conference on Integrated Circuits and Microsystems (ICICM)*, pages 246–253, 2023.

[50] W. K. Zuravleff and T. Robinson. Controller for a synchronous dram that maximizes throughput by allowing memory requests and commands to be issued out of order., 1997.