Rank-switching, Open-row DRAM Controller for Mixed-Critical Real-Time Systems

by

Yogen Krishnapillai

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, 2015

Note that some contents of this thesis are taken from my previously published paper [1] where I am the first author. In addition, some content are taken from my other published paper [2] where I am one of the co-authors.

Abstract

In this thesis, we present a rank-switching open-row DRAM controller for mixed critical real time systems. This memory controller is optimized for multi-requestor and multi-rank memory systems. The key to improved performance is an innovative rank-switching mechanism which hides the latency of write to read transitions in DRAM devices without requiring unpredictable request reordering. We further employ open-row policy to take advantage of the data caching mechanism (row buffering) in each device. We choose the bank privatization scheme where each requestor is assigned its own private bank or set of banks. This private bank mapping guarantees that each requestor has its own row buffers and cannot be interfered by other requestors. The proposed memory controller design allows maximum of thirty two requestors at a time targeting either two or four ranks. This controller provides complete timing isolation between critical and non-critical applications and allows for compositional timing analysis over number of requestors and memory ranks in the system. We designed both the front end logic for the command generation and back end logic for the DRAM timing constraint check and arbitration utilizing the rank switching techniques. The complete design is implemented and synthesized using Verilog RTL and finally, we evaluated performance using various benchmarks. Our proposed memory controller offers significantly lower worst case latency bounds for critical real-time applications and supports average throughput for non-critical real-time applications compared to existing real time memory controllers in the literature.

Acknowledgements

I would like to thank my supervisor Rodolfo Pellizzoni for his encouragement and excellent guidance for a successful completion of this research. Prof Rodolfo has been really helpful and dedicating his valuable time to accomplish this research project.

I would like to thank my supervisor Manoj Sachdev for giving me the motivation to apply for MASc Program while my attention was diverted by my full time carrier.

I would also like to thank Professor Andrew Morton and Hiren Patel for reviewing this thesis and for their valuable comments.

Dedication

This is dedicated to my parents, my brothers and sisters for their constant love and motivation throughout my life. I would not be here without them. It has been a great challenge studying a thesis based Engineering Master program while focusing on a full time carrier in the same field of interest. Dedicating the time and energy for both studies and carrier at the same time has been a rewarding experience.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The memory clients either have real time or non-real time requirements. The real time requests can be either critical or non-critical. The critical requests demand worst case upper bound latency whereas non-critical requests demand average minimum bandwidth. The critical real time systems such as avionic system, nuclear plant and safety-critical electronic medical devices demand the worst case upper bound latency. In this thesis, memory clients such as CPU or hardware accelerators or IO peripherals are referred to as requestors from now on. The proposed memory controller was implemented by utilizing the rank-switching techniques, open-row policy, private bank mapping and dynamic scheduling. The memory controller implemented in this fashion is to show how the latency of a memory request can be significantly reduced by applying rank switching techniques and thereby hiding the highly time consumed read to write and write to read timing constraints. Further, designing the memory controller in this fashion helps to prove the point that large number of requestors can be handled through effective multi stage arbitration mechanism while satisfying their complex memory timing parameters to serve all requestors to execute their memory request demand in an orderly fashion. In Section 1.1, we present the problem statement and Section 1.2 lists the contributions accomplished for this research. Section 1.3 gives a preview of how this document is organized and finally Section 1.4 provides the content acknowledgement.

1.1 Problem Statement

The challenges arise when large number of memory clients is running in parallel with its own critical and non-critical applications targeting one and only common shared memory controller in a single channel memory environment. The common shared memory controller should be able to handle critical and non-critical applications where heavily inter dependant DRAM timing constraints become very complicated to analyse. Further, the command scheduling of all the memory clients become another challenge when the number of memory clients is growing.

Scheduling DRAM commands for many memory clients is not straight forward, since there are a number of timing constraints that must be satisfied before a memory client can be chosen and its command can be issued. It is a great challenge to design such a memory controller that provides equal chances to all the memory clients through fair arbitration and schedule their commands dynamically while satisfying the inter-dependant timing constraint. The design strategy of this proposed memory controller should also focus on scenarios where both non-critical real time requestors and non-real-time requestors cannot interfere with the tight latency timing deadlines of the critical real time requestors. In other words, the memory controller should be able to handle different type of requestors according to their respective latency and throughput requirements. All these demands should be satisfied by providing the tight bounds on the worst case execution time for the critical requestors and average throughput for the non-critical requestors along with guaranteed bandwidth.

1.2 Contribution

In this thesis, we present a rank-switching, open-row memory controller for mixed critical real time systems. The major contributions are the following.

- The worst-case execution time was analysed for a single memory request from requestor under analysis while remaining other requestors provide worst case memory interference at the same time. This initial phase helped to look into the parallelism nature to reduce the interference among multiple requestors.

- Our rank-switching open row memory controller architecture has both front end and back end logic blocks. The front end design was implemented to achieve address mapping, refresh controller and command generation for a multi requestor environment where our memory controller can accept requests from "n" number of requestors where $0 < n <= 32$.

- The back end design was implemented with three levels of arbitration such as requestor arbitration, rank arbitration and command arbitration. To achieve the highest throughput, back end design was architected in a four stage pipelined fashion.

- The verification platform such as test bench, tests suits and simulation were developed and the design was verified.

- The final design was synthesized. The Static Timing Analysis (STA) was carried out to fix set up and hold time violations.

- The evaluation was carried out on our memory controller through extensive hardware simulations to analyse how the rank-switching techniques effectively improve the performance. The evaluation results were compared with the analytical results.

1.3 Organization

This document is organized as follows. Chapter 2 provides required background on DRAM. Chapter 3 discusses proposed memory controller design that includes the important design decisions and arbitration rules. Next, Chapter 4 is focused on the theoretical analysis of worst case per-request latency. Chapter 5 is dedicated to the implementation detail of our proposed memory controller. Next, Chapter 6 discusses the verification platform used to verify the design and also evaluates the performance of our design. Finally, Chapter 6 provides concluding remarks. At the end, the schematic diagrams and one sample simulation output waveform of the memory controller design are included in Appendix. The Verilog RTL code of the design can be found at [18].

1.4 Acknowledgement

Section 4.0, Section 3.2, Section 2.5 and Section 2.1 were taken from published paper [1]. I would like to thank Professor Rodolfo Pellizzoni for his great assistance in formulating the theoretical analysis and arbitration rules. I would also like to thank Zheng Pei Wu for his support for published paper [1] and providing the necessary benchmark memory traces used in our simulations.

Chapter 2

Background

This background chapter is dedicated to three main areas. First, it describes the basic operation of DRAM memory device. Second, complex timing behaviour of DRAM will be illustrated in details. Finally related work performed by others is discussed and it helps to differentiate between our proposed design and the existing approaches.

2.1 DRAM Basics

Modern memory devices are organized into ranks and each rank is divided into multiple banks, which can be accessed in parallel provided that no collisions occur on either buses. Each bank comprises a row-buffer and an array of storage cells organized as rows and columns. This thesis considers devices with at least two ranks for our analysis on rank switching techniques. A Memory Controller controls the operations of DRAM device by issuing five important memory commands such as Activate, Read, Write, Pre-charge and Refresh.

To access the data in a DRAM row, an Activate (ACT) command must be issued to load the data into the row buffer before it can be read or written. Once the data is in the row buffer, a read CAS or write CAS command can be issued to retrieve or store the data. If a second request needs to access a different row within the same bank, the row buffer must be written back to the data array with a Pre-charge (PRE) command before the second row can be activated. In a rank, when each DRAM device contributes with 8 bits, a rank with 8 devices has the data bus size of 64 bits. The following Figure 2.1 illustrate row, columns, bank and rank configuration and it also shows how the total 64 bit data from the memory controller reach the rank that has 8 DRAM devices.

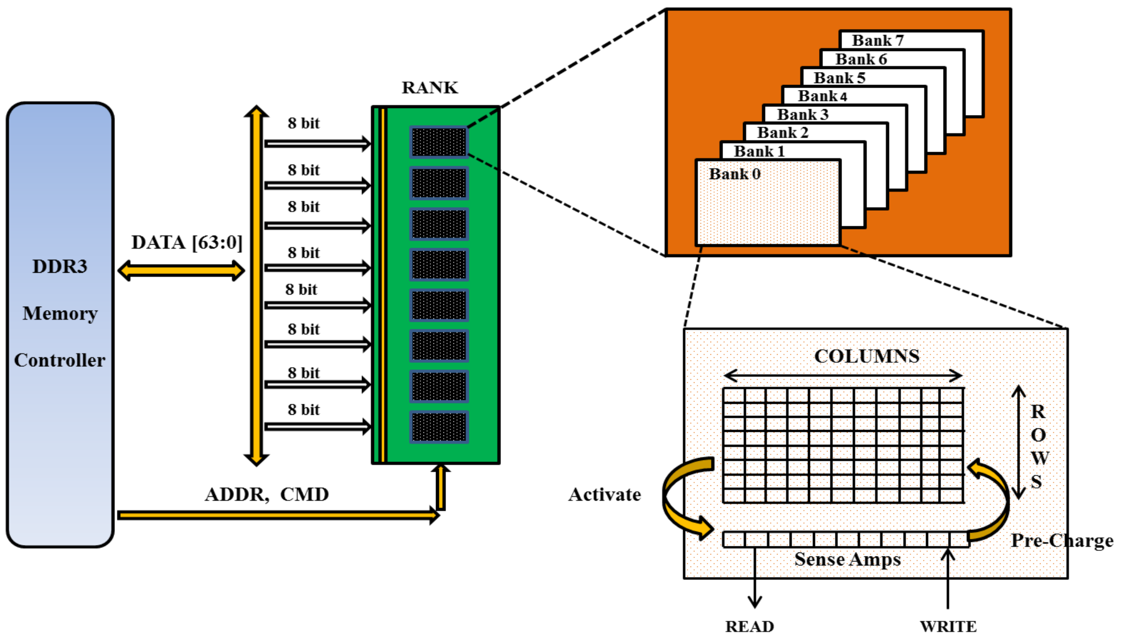


Figure 2.1 DRAM Architecture

Finally, a periodic Refresh (REF) command must be issued to all ranks and banks to ensure data integrity. Note that each command takes one clock cycle on the command bus to be serviced. Each CAS command accesses data in a burst of length BL and the amount of data transferred is BL x WBUS, where WBUS is the width of the data bus. Since DDR memory transfers data on rising and falling edge of clock, the amount of time for one transfer is t_{BUS} = BL/2 memory clock cycles. For example, with BL = 8 and WBUS of 64 bits, it will take 4 cycles to transfer 64 bytes of data. A row that is cached in the row buffer is considered open, otherwise the row is considered closed. For an open request, only a read or write CAS command is generated since the desired row is already cached in row buffer. For close request, if row buffer contains a row that is not the desired row, then a PRE command is generated to close the current row. Then an ACT is generated to load the new row and finally read/write is generated to access data. The memory controller can employ one of two polices to manage the row buffers. Under open row policy, the memory controller leaves the row buffer open for as long as possible. In contrast, close row policy automatically pre-charges the row buffer after every request. Finally, the controller must map the incoming request to the correct rank, bank, row and column. With interleaved bank mapping, each request can access all banks in parallel. However since all requestors share all banks, they can cause mutual interference by closing each other's rows. With private banks mapping, each requestor is assigned its own bank or set of banks. Therefore, the state of row buffers of one requestor cannot be influenced by other requestors.

2.2 DRAM Timing Constraints

Every memory device has timing requirements in order to perform read, write, and refresh operations. Therefore, it is the memory controller which satisfies the timing constraints needed by the memory devices. The operation and timing constraints of memory devices are defined by the JEDEC standard. The Table 2.1 lists the description of all timing parameters for DDR3-1333H device that we used in our design. The table 2.10 also show which timing parameters are involved when the requests target the same bank, same rank or different banks, same rank or different ranks.

JDEC SPECIFICATIONS					
Timing Parameters	Descriptions	DDR3 1333H	Same Bank Same Rank	Diff. Banks Same Ranks	Diff. Ranks
t_{RCD}	ACT to READ/WRITE delay	9	Yes	No	No
t_{RL}	READ to Data Start	9	Yes	No	No
t_{WL}	WRITE to Data Start	7	Yes	No	No
t_{BUS}	Data bus transfer	4	Yes	Yes	Yes
t_{RP}	PRE to ACT Delay	9	Yes	No	No
t_{WR}	End of WRITE to PRE Delay	10	Yes	No	No
t_{RTP}	Read to PRE Delay	5	Yes	No	No
t_{RAS}	ACT to PRE Delay	24	Yes	No	No
t_{RC}	ACT-ACT (same bank)	33	Yes	No	No
t_{RRD}	ACT-ACT (different bank)	4	No	Yes	No
t_{FAW}	Four ACT Window	20	No	Yes	No
t_{RTW}	READ to WRITE Delay	7	Yes	Yes	No
t_{WTR}	WRITE to READ Delay	5	Yes	Yes	No
t_{RTR}	Rank to Rank Switch Delay	2	No	No	Yes
t_{RFC}	Time required to refresh a row	160 ns	Yes	No	No
t_{REFI}	Refresh period	7.8 us	Yes	No	No

Table 2.1 DRAM Timing Constraint

2.2.1 Rank to Rank Timing Basics

Before analysing the timing analysis within the same rank and between different ranks, it is important to analyse the origin of this timing requirement called Rank to Rank (RTR) which was derived from Data Strobe (DQS). DDR SDRAM uses both a clock and a source-synchronous data strobe (DQS) in order to achieve high data rates. The DQS signal is a shared signal used by either bus masters such as memory controller or DRAM memory device. During the read command, the DQS signal is driven by the Memory device to notify the read data timing to the memory controller. During the write command, the DQS signal is driven by the memory controller to notify the write data timing from the memory controller to the Memory device. While DQS is driven by the Memory controller, the same DQS is used by the memory device to sample the incoming write data. Since the DQS is a shared signal by the bus masters, the synchronization time is needed for one bus master to hand off the DQS signal to another bus master. The DQS is the bus turnaround time, inserted to account for skew on the bus and to prevent different bus masters from driving the bus at the same time. To avoid such collisions, a second rank must wait at least t_{DQS} after a first rank has finished before driving the bus. This synchronization time is called Rank to Rank Time, RTR.

2.2.2 Requests targeting the same Rank

The Figure 2.2 shows the scenario where requests target different rows, banks within the same rank. First, the REQ1 targets row 1, bank 1 and rank 1. Since it is a close read request, the memory controller issues the ACT command to row 1, bank 1, rank 1. Then, it waits for RCD delay to issue the CAS Read command and it waits for Read Latency (RL) time unit before expect to receive the first byte of read data from the memory device. The length of the read data is equal to the burst length (BL).

Right after REQ1 arrived, a new write request REQ3 also arrived and targeting different bank in the same rank (row 1, bank 2, rank 1). Since it is a close request, the ACT command needs to be issued for REQ 3. As per the ACT to ACT timing constrain, the row to row delay constrain is applied for requests targeting different banks in the same rank. Therefore, the REQ3 ACT command cannot be issued right after the REQ1 ACT command. Instead, the REQ3 ACT command should be delayed by RRD delay. Now, the REQ 3 has issued ACT command after RRD time.

At this point, that memory controller cannot issue CAS write command after the RCD delay is elapsed as expected. This scenario is explained in the following paragraph.

When read and write requests targeting different banks or same bank within the same rank, the memory controller must satisfy the Read to Write (RTW) timing constrain. Therefore, the memory controller needs to wait for the Read to Write (RTW) time delay from the REQ1 CAS read command before issue REQ 3 CAS write command. After CAS Write command is issued, memory controller needs to wait for Write Latency (WL) in order to send the write data to the memory device. After the write data is written out, memory controller needs to wait for Write Recovery (WR) time before issues the PRE, if row need to be closed. Closing the row depends on open or close row policy.

While REQ1 is in action receiving read data, a new read request, REQ 2, just arrived targeting the same bank and the same rank as REQ 1, but different row (row 2, bank 1, rank 1). Since the REQ 2 target the different row in the same bank, the previous row which was being accessed by the REQ1 has to be closed before opening a new row to REQ2. To issue PRE command in order to close the row, the memory controller has to wait for the maximum of either Read to Pre-charge (RTP) or RAS timing constraint. After issuing the PRE command for REQ2, the controller needs to issue the ACT command after waiting for the maximum of either RP or RC timing constraint. After issuing the ACT command for REQ2, the memory controller has to wait for RCD delay time in order to issue the CAS read command as per the DRAM protocol. But, REQ 2 CAS read command can only be issued after the timing constrain called Write to Read (WTR) is satisfied as shown in Figure 2.2. It is important to point out the difference between RTW and WTR constraints. The RTW constraint is between the read command to the write command of the same or different requestors. But, the WTR constraint is between the completion of write burst data to the read command of the same or different requestors.

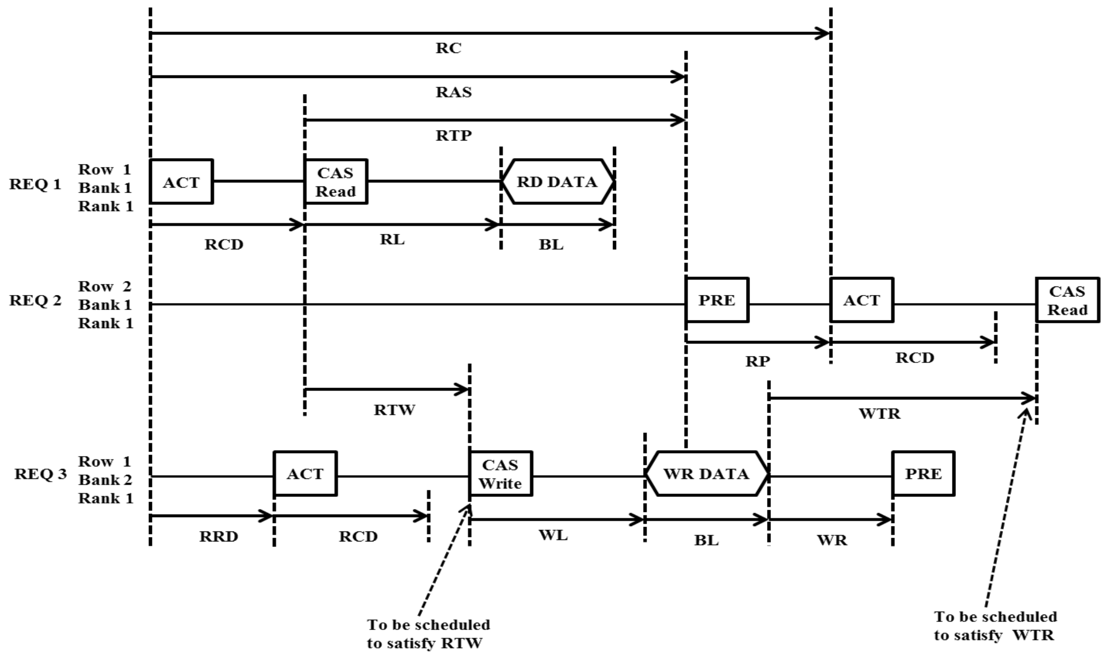


Figure 2.2: Requests targeting different banks in the same rank

2.2.3 Requests targeting different Ranks

The Figure 2.3 shows the scenario where requests target different ranks. The first read request (REQ1) arrives and since it is a close request targeting rank 1, the memory controller issues the ACT command. At the same time, another close read request, REQ 2, also arrives and targeting rank 2. Since it is a close request, the memory controller issues the REQ 2 ACT command. This ACT command can be issued right after the REQ1 ACT command without waiting for the Row to Row delay (RRD). It is due to the fact that both requests are targeting different ranks and there is no constraint between ACT commands of requestors that target different ranks. As it can be seen from Figure 2.3, the REQ1 CAS read command and the corresponding Read Data follows as per regular timing parameters that we saw before. But, for the REQ2, after RCD delay is elapsed from its ACT command, REQ2 CAS write command cannot be issued, instead CAS write command need to be scheduled to satisfy the new timing constraint called Rank to Rank (RTR). The RTR delay is needed to satisfy the constraint between end of read data of REQ1 from rank 1 and beginning of write data of REQ2 from rank 2. In other words, the memory controller can only begin sending the REQ2 write data after RTR timing is elapsed from the end of the REQ1 read data.

While REQ1 is receiving its read data, there is a new request, REQ 3 that arrives and targets the same rank, bank and rows as of REQ1.

The REQ 3 is considered as open request since the row is already opened by the REQ1. Therefore, there is no need to issue the ACT command; instead, the memory controller can issue the CAS read command to access the already opened row. But, this CAS read command cannot be issued right away, instead it need to be scheduled to satisfy the rank to rank (RTR). Important observation is that there is no need for write to read (WTR) or read to write (RTW) constraint in this scenario where requests target different ranks. The WTR and RTW timing constraint are only applicable for the requestors targeting either same bank or different banks in the same rank as we saw in the previous Section 2.2.1.

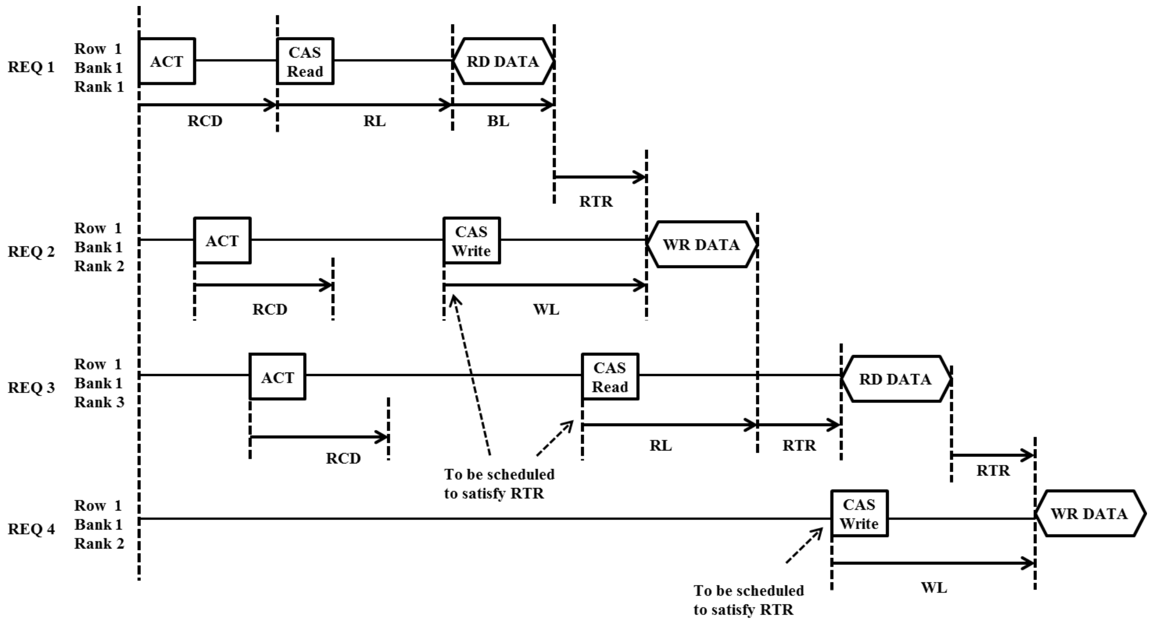


Figure 2.3: Requests targeting different ranks

2.3 DRAM Row Buffer Management Policy

Section 2.1 explains the operation of row buffer in DRAM. The policy that manages the operation of row buffer is called row buffer management policy. There are two types of policies existing and they are open row and close row. The decision on choosing one of them depends on the memory controller designer's choice in terms of performance and power consumption. For the open row policy, the memory controller allows the row buffer to be always open until a request to read a different row. If another memory request arrives to the same row address with different column address, memory access is possible with the minimum latency of CAS Latency (CL) without re-opening the row due to the open row policy. Therefore, this policy saves un-necessary RAS to CAS latency delay by re-opening the row again. When the controller send the memory request to different row in the same bank, the open row needs to be closed by PRE command before opening the new row. On other hand, the close row policy automatically closes the row buffer after a request and consequently, every new request to the same row has to issue ACT command to open the row even if it accesses the same row as before.

2.4 DRAM Mapping

The memory controller receives the memory request in the form of just physical address and the request type. It is the task of memory controller to map the incoming raw physical address into correct rank, bank, row and column addresses to access the memory devices. There are two types of mapping methodologies.

2.4.1 Continuous Memory Mapping

The continuous memory mapping is where the incoming physical memory address is mapped within the single row of a particular bank. The sequential access continues through different columns address in the same row until the end of the row is reached. Only when the current row is finished accessed, the mapping switches to the same row number of next available bank as shown in Figure 2.4. If the next bank is not available, then, the logical address is mapped to the next row in the current bank. Private bank mapping is a sub-set of continuous mapping scheme. When a private bank scheme is used in a multi requestor system, each requestor is assigned to either one bank or set of disjoined banks within the same rank. Continuous memory mapping is very efficient method with no bank conflicts when the memory requests are continuous sequential addresses. But, this method becomes inefficient if the memory requests reach to different rows in the same bank.

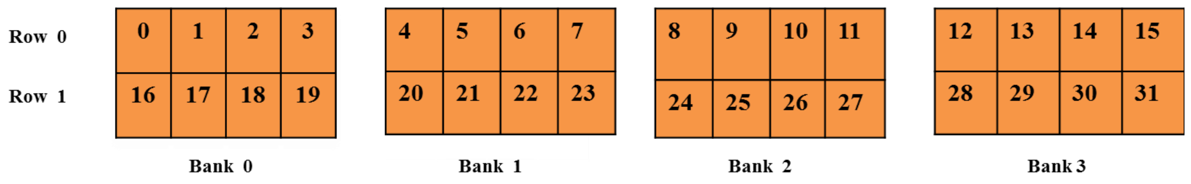


Figure 2.4: Continuous Memory Mapping

2.4.2 Interleaved Memory Mapping

The incoming physical address is mapped to column address locations of a row from all banks available in rank. Once all banks have been accessed, then, the incoming physical address is mapped to the next column address location of the same row from all banks. When the row becomes full, the incoming logical address is mapped the next row from all available banks as shown in Figure 2.5. An interleaved memory with $\boldsymbol{n}$ banks is said to be $\boldsymbol{n}$-way interleaved. If there are $\boldsymbol{n}$ banks, memory location $\boldsymbol{i}$ would reside in bank number $\boldsymbol{i}$ mod $\boldsymbol{n}$. The interleaved method has an advantage of make use of multiple banks and access all banks simultaneously with addresses are spread over banks and hence this mapping provides the efficient bank parallelism and results in higher memory throughput. But, the drawback is that it involves complex design and it is only efficient when you require burst access to all the banks.

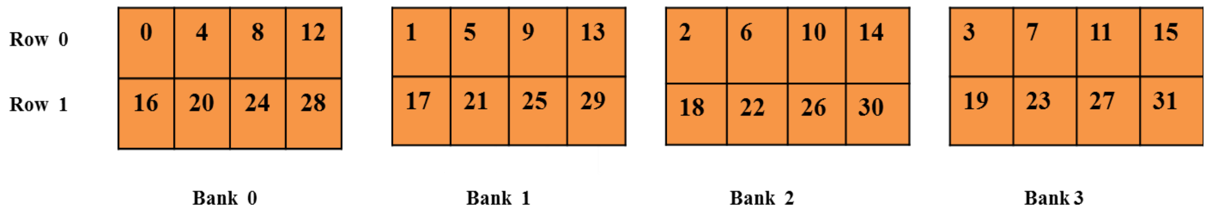


Figure 2.5: Interleaved Memory Mapping

2.5 Related Works

The related works on memory controller design carried out by researchers can be classified under different implementation categories such as close row, open row, critical, mixed critical, rank switching and arbitration policy. Let us analyse the related work under each of these categories.

First, we will be looking into the related work focusing on real time memory controllers with close row policy. The work done by Analyzable Memory Controller (AMC) [7] and Predator [8] employ close row policy designed for critical systems. The interleaved banks are chosen as the bank mapping strategy by [7] and [8]. In interleaved bank mapping, there is no guarantee that rows opened by one requestor will not be closed by another requestor.

Therefore, both [7] and [8] offers predictable timings, but the latency can be significantly higher than controllers using open row policy. The work done by Yonghui et al. [3] presents the architecture of a dynamically scheduled real time memory controller. The paper analyses to minimize the worst case execution through close page policy and bank parallelism with interleaved bank mapping. Further, this paper [3] specifically addresses the issue of having either fixed or variable transaction size for the real time memory controllers. Further, the papers [4], [10] and [11] also utilize the close page policy in their approach for the memory controller design.

From the open page policy category, Goossens et al. [9] have proposed a new type of open page policy called conservative open page policy. The approach in [9] states the following. Do not pre-charge if next request is known to target the open row. Do the Pre-charge if next address is not known in time, or in case of a miss. It also makes sure that not to reduce the guarantees given by the close-page policy. In other words, the approach in [9] wants to leave a row open for a fixed time window to take advantage of row hits. In the worst case, this approach is the same as close row policy if no assumptions can be made about the exact time at which requests arrive at the memory controller. Further, Author [5] has done extensive analysis on rank switching based on open row policy.

Next, we would discuss the related work focussing on experimenting different arbitration policies. The authors [8] employ a credit controlled static-priority (CCSP) is used to share between multiple requestors. Authors [8] uses a hybrid approach between the static DRAM command scheduling, better for timing guaranties, and the dynamic command scheduling, better for average-case memory bandwidth utilization.
Goossens et al. [9] use the work-conserving Time-division multiplexing (TDM) as the arbitration. The TDM arbitration makes the unclaimed slots from one application to be by another application if it has a request available. On the other hand, the Analyzable Memory Controller (AMC) [7] provides upper bound latency for memory requests in a multi-core system by utilizing a round robin arbiter. Reineke et al. [10] propose a memory controller that uses TDMA scheduling. On the other hand, Akesson et al. [11] propose an arbiter called credit-controlled static-priority (CCSP) consisting of a rate regulator and a static-priority scheduler.

Now, let us focus on bank mapping methodology. Most of the research papers in this related work focus on interleaved as the bank mapping methods in their design. Only very few research papers pay attention to the private banking mapping.

Leonardo et al. [4] discuss the private banking through a terminology called virtual devices (VD) where each VD is a group of two banks from the same rank. Further, this paper proposes to share each VD between one critical and a pre-determined number of non-critical applications. The private banking scheme helps to define the clear boundary between critical and noncritical applications for their mixed critical memory controller. Further, Reineke et al. [10] propose a memory controller that uses bank privatization for predictability and temporal isolation.

When it comes to the rank switching techniques, the research paper [4], [5] and [6] use the rank switching methods. The Wang et al. [5] proposed a rank hopping algorithm to maximize DRAM bandwidth by scheduling a read group (or write group) to the same rank to leverage bank parallelism until t_{FAW} constraint is reached. At that point, another group of CAS commands are scheduled for another rank. This way, they amortize the rank to rank switching time across a group of CAS commands. However, this scheduling policy inherently re-orders requests and it is not suitable for critical real time systems that require guaranteed latency bounds. The work in [6] also uses rank scheduling to reduce DRAM power usage by minimizing the number of state transitions from low power to active state. In papers [5] and [6], the rank scheduling and optimizations have only been applied to non-real time systems. The paper [4] introduces the rank switching analysis for mixed critical systems. But, the rank switching analysis is limited to only two ranks.

In contrast, the approach of this thesis takes advantage of rank switching techniques that hides the latency of write to read and read to write transitions and thereby enable the design to achieve the tight bounds on worst case execution time (WCET) to critical core requestors and the lowest possible average execution for non-critical core requestors. While most of the memory controller by other researchers focuses on close page policy, we attempted to implement memory controller based on open row policy and also take advantage of the private bank scheme where the interference from other requestors is eliminated. As a possible downside, using private banks reduces the total memory available to each requestor compared to interleaving methods. But, increasing the DRAM size is not an issue compared to designing a memory controller that can work in a multi requestor real time environment. Further, our memory controller has three stages of arbitration where each stage has its own arbitration mechanism of FCFS, RR and priority.

	Close Row or Open Row	**Arbitration Policy**	**Critical or Mixed Critical or Non Critical**	**Rank Scheduling**	**Bank Mapping**
AMC [7]	Close Row	RR Arbitration	Critical	NA	Interleaved Bank
Predator [8]	Close Row	Credit-Controlled Static Priority (CCSP)	Critical	NA	Interleaved Bank
Reineke [10]	Close Row	TDM Arbitration	Critical	NA	Private Bank
Wang [5]	Open Row	RR Arbitration	Non Critical	Rank Hopping	Interleaved Bank
Yonghui Li [3]	Close Row	FCFS Arbitration	Mixed Critical	NA	Interleaved Bank
Goossens [9]	Conservative Open Row	TDM Arbitration	Mixed Critical	NA	Interleaved Bank
Leonardo [4]	Close Row	Fixed Priority	Mixed Critical	Rank Switching	Private Bank
Akesson [11]	Close Row	Credit-Controlled Static Priority (CCSP)	Mixed Critical	NA	Interleaved Bank

Table 2.2: Summary of the related work

Chapter 3

Memory Controller Design

This chapter discusses details of important design decisions and the arbitration rules that are formulated as a strong foundation to our memory controller implementation. Based on the design decisions and arbitration rules from this chapter, the implementation of the memory controller design will be discussed in the next chapter.

3.1 Design Decisions

The design decisions such as type of row management policy, address mapping scheme and rank-switching mechanism will be discussed next.

3.1.1 Row Management Policy

The Row Management Policies can be either open row or close row. When the same requestors target the same row in the memory, the CAS command can be issued with the minimum CAS Latency (CL) without re-opening the row due to the open row policy. Therefore, the open row policy avoids the un-necessary RAS to CAS latency delay by re-opening the row again and therefore, the open row policy reduces latency time. To take advantage of the latency reduction, the open row policy is chosen for our memory controller design, because we know the number of open and close rows. On the other hand, the downside of the open row policy is that we cannot take advantage of automatic pre-charge operation. Further, the open row policy requires additional commands to be active in the bus which eventually create the bus contention.

3.1.2 Address Mapping Scheme

Address mapping scheme can be either continuous or interleaved. The private bank mapping is a sub-set of continuous address mapping. In the private bank mapping, each requestor is assigned with one bank or set of banks which are disjoined. The incoming logical address is mapped to the first row and when the first row become full, the continuous access is mapped to the next row and so on within the same bank.

It is important to analyse why the interleaved banking method is not suitable for our proposed design. The interleaved banking allows all requestors to access all the banks where banks could be shared by requestors. But, in real time systems, we do not want each requestor to share the banks between them. It is because; there is a high probability where one requestor can close a row in a bank which was already opened by a second requestor. This kind of interference creates unwanted latency delay for the second requestor to re-open the row which was closed accidently by the first requestor. Based on these reasons, the interleaved banking is not suitable for our memory controller design. Therefore, our rank switching memory controller design has chosen the private bank mapping where each requestor is allocated to one bank or set of banks. But, there is a downside of private bank. Since each requestor is allocated to just one private bank, each requestor has limited memory access. But, increasing the memory size is not an issue and we can increase the amount of memory that each private bank is assigned as a solution.

3.1.3 Rank Switching Mechanism

Introducing the Rank switching technique provide strong isolation and composable properties to our proposed memory controller design. The composability provides the way to integrate components at the same time preserving their temporal properties. In an ideal system, we like to achieve a data bus utilization of 100 %. In practice, due to the many timing constraints detailed in Section 2.2, data bus utilization is typically much lower. This is true even if all requests are open, since t_{RTW} and t_{WTR} significantly increase the timing between successive read and write commands or vice-versa. Let us look at an example that illustrates the rank switching mechanism.

Figure 3.1 (A) depicts the worst case situation for four successive open requests of different requestors in a single-rank system, which is an alternation of store and load (write and read CAS commands). Note that it takes 52 clock cycles to complete all four requests, while the data bus is only used for 16 cycles, resulting in a utilization of only 31%. Our key idea is that we can improve the worst-case latency by noticing that t_{RTW} and t_{WTR} do not apply between requests that target banks in different ranks.

Figure 3.1 (B) shows the schedule derived by assigning the four requestors to two different ranks and alternating servicing requests to the two ranks. Since the only constraint between requests to different ranks is the shorter t_{RTR}, the schedule now takes 35 cycles to complete, a 33% improvement.

Similarly, Figure 3.1 (C) shows the effect of assigning each requestor to a different rank. Note that in this case, after data is started at cycle 7, we use the data bus for 4 cycles every 6, resulting in a utilization of 2/3. Finally, notice that alternating ranks also helps reducing the latency of ACT commands of close requests, since the t_{RRD} and t_{FAW} constraints do not apply between different ranks.

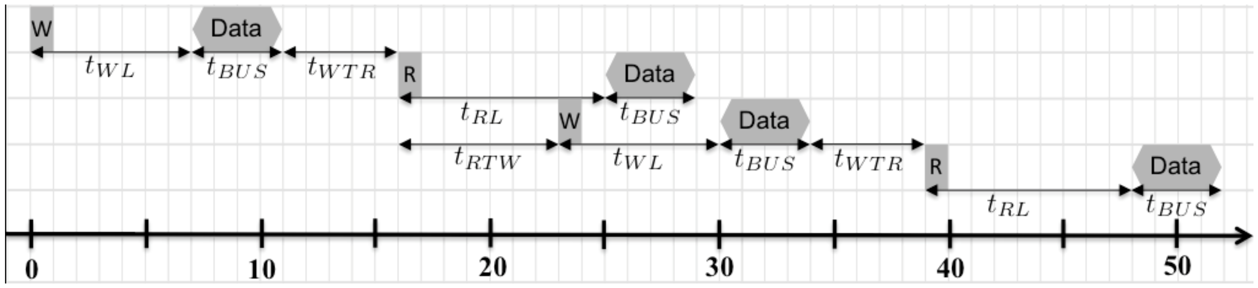


Figure 3.1 (A): Arbitration for 1 Rank

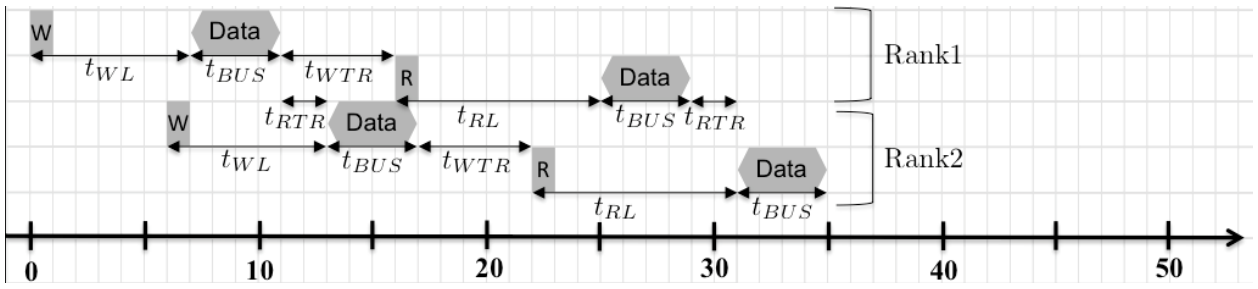


Figure 3.1 (B): Arbitration for 2 Ranks

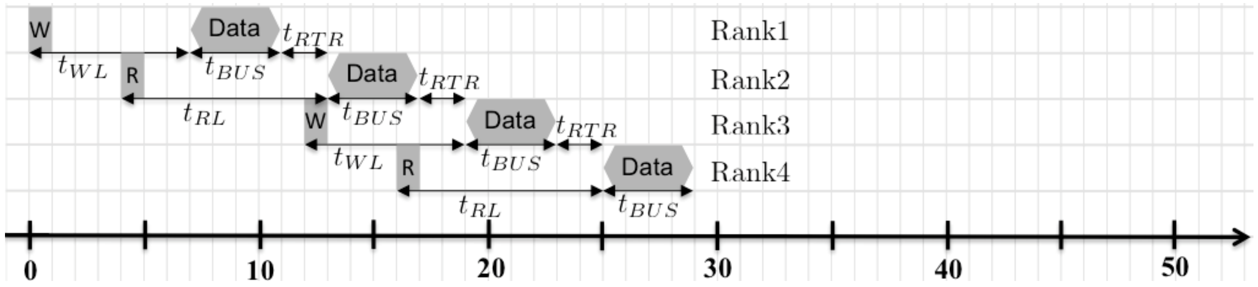


Figure 3.1 (C): Arbitration for 4 Ranks

Our illustrative example shows that a rank-switching mechanism in the back end can both significantly decrease the latency of memory requests and increase bus utilization without requiring us to reorder requests in the front end, which is unsuitable for critical real-time requestors needing guaranteed latency bounds.

The challenge is how to implement such mechanism in a predictable way. In particular, a simple static TDMA schedule is not suitable since requestors can dynamically submit different types of requests at run-time. Instead, a set of dynamic arbitration rules is proposed in Section 3.1. Having seen the advantages of using rank switching technique from the examples; now the challenge is how to implement such rank switching mechanism in the memory controller.

3.1.4 Selection of Arbiter Type

Arbiter scheduling can be chosen from one of the following policies such as Priority, Round Robin, First Come First Served (FCFS) and TDMA. This section describes the logical reasons for the selection of each arbiter type for our proposed design. Figure 3.2 shows an overview of all the arbiters and its scheduling type that were used in our design. We will discuss each arbiter type that was used under requestor arbitration, rank arbitration and command arbitration categories.

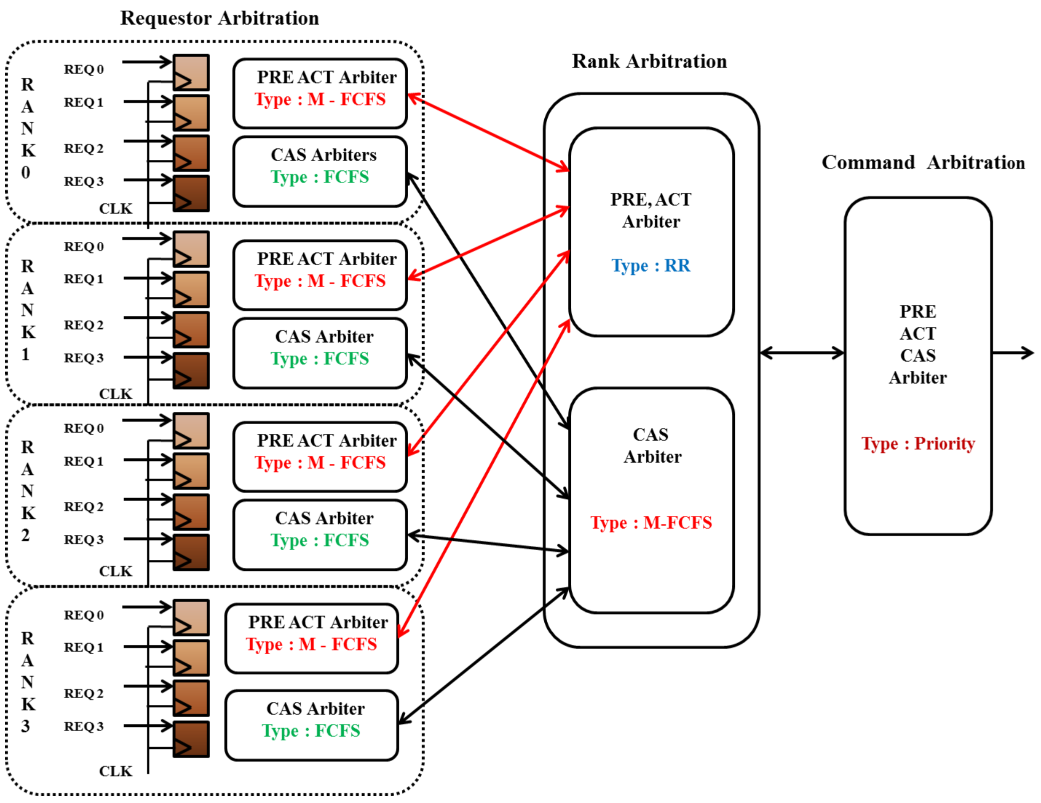


Figure 3.2: Choice of Arbiter Types

The Requestor Arbitration category consists of two arbiters such as CAS Arbiter and PRE ACT Arbiter. The task of these arbiters is to choose a requestor from a set of requestors of the same type. Out of different arbiter scheduling mechanisms, we need to analyse why certain type of arbiter scheduling is suitable and why others are not suitable for our design. If priority style was chosen, it assigns one requestor as the highest priority over others. The lowest priority requestor would be starving while the high priority requestor owns the bus master for a long time. Since all requestors are of the same type, we do not want one requestor to starve for the bus ownership. If the Round Robin type arbiter was used, it rotates the priority level among all the requestors where each requestor has equal time of being the highest priority. On the other hand, a simple static TDMA schedule is not suitable since requestors can dynamically submit different types of requests at run-time. Our requirement is that while achieving the equal fairness, we do not want to waste the clock cycles by the arbiter visiting the requestor that has no requests. Therefore, CAS arbiter was designed as a First Come First Served (FCFS) style. On other hand, PRE ACT Arbiter has to do additional task of giving PRE command higher priority compared to the ACT when the ACT is waiting to satisfy the timing. The design decision is made to grant PRE command higher priority than ACT command and this make the PRE ACT arbiter to be a modified First Come First Served arbitration style (M-FCFS).

The Rank Arbitration category consists of Level 2 PRE ACT arbiter and level 2 CAS arbiter. In order to maintain the fairness and equal priority in choosing the ranks, level 2 PRE ACT arbiter is designed as Round Robin style. On the other hand, the level 2 CAS Arbiter has to choose its level 3 queues based on the burst to burst (BTB) values of the requests arriving from level 3. Further, this level 2 CAS Arbiter has to differentiate clients that arrive early versus other clients who wait for a write to read (WTR) or read to write (RTW) timing constraints to be elapsed. By considering all these requirements, the level 2 CAS Arbiter was designed to be Modified First Come First Served (M-FCFS).

In Command Arbitration category, there exists a command arbiter who handles all commands such as PRE, ACT and CAS. In order to give CAS command to be higher priority than other commands, the priority based arbiter is used. All three arbiters in this design were not intended to perform reordering of the incoming requests and thereby, it really help to avoid the unnecessary complexity in timing analysis. Now that we have seen the arbitration types, let us analyse the detail of the arbitration rules.

3.2 Arbitration Rules

The back end memory controller logic is built into three levels of arbiters as shown in Figure 3.3 below. Each level has different type of arbiters and it is important to analyse them through arbitration rules. We consider a device with $R \geq 2$ ranks. The memory controller can support both critical and non-critical real-time requestors. Our design goal is to minimize the latency bound of critical requests, while simultaneously attempting to maintain high data bus utilization and thus provided memory bandwidth to all requestors. To this end, each rank is assigned either to critical or to non-critical requestors and each requestor uses only one rank; let Mr $1 \leq r \leq R$, be the number of requestors that use rank ***r***. The banks in critical rank ***r*** are statically partitioned among the Mr requestors in rank ***r***, according to the private bank principle.

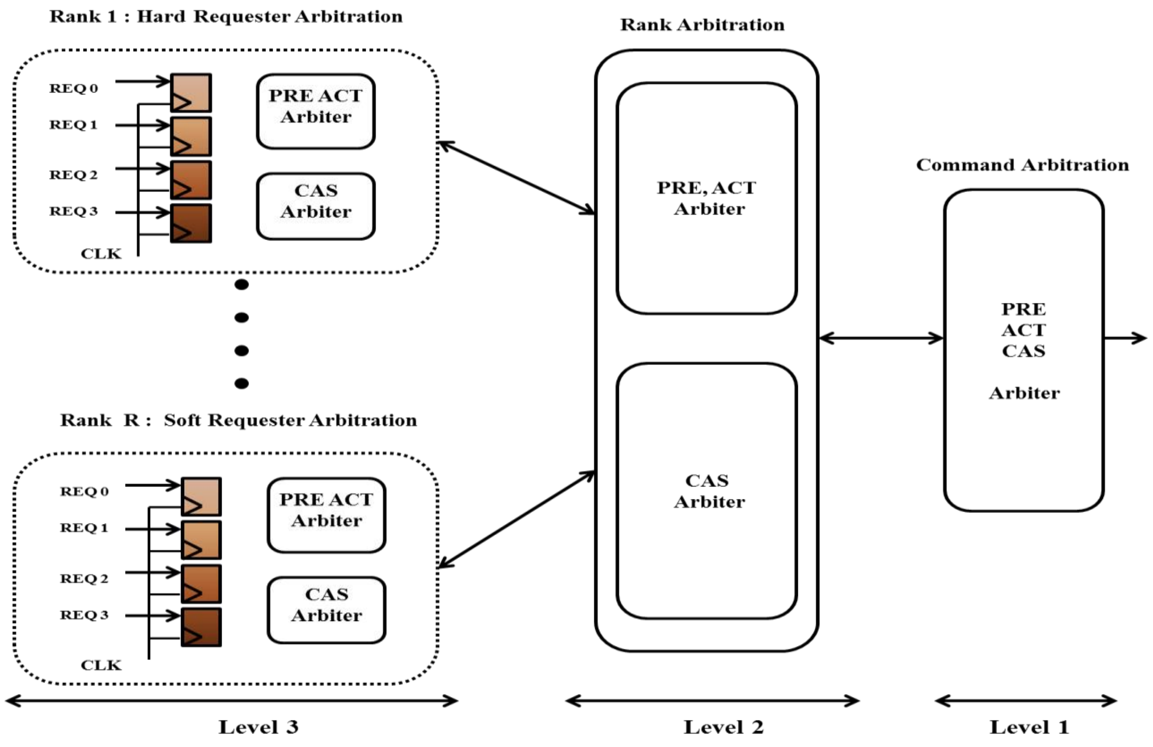


Figure 3.3: Three Levels of Arbitration

Figure 3.3 shows an example block diagram of the three levels of arbitration logic in the back end, where Rank 1 is a critical rank, Rank ***R*** is a non-critical rank. $M_1 = 4$ indicates that rank 1 has four requestors. Arbitration is performed in three levels.

For critical ranks, commands generated by the front end are en-queued in the per-requestor command queues. Level 3 (L3), or Requestor Arbitration, arbitrates among requestors within the same rank. The command at the front of the selected requestor queue is propagated to Level 2 (L2), or Rank Arbitration, which arbitrates among the R ranks. Note that Level 3 and Level 2 arbitrations are split between a PRE ACT Arbiter that handles PRE and ACT commands only, which are needed only for close requests, and a CAS Arbiter that handles CAS commands, which are needed by all requests. Finally, Level 1 (L1), or Command Arbitration, simply assigns higher priority to CAS than PRE or ACT command; i.e., if during the current clock cycle the L2 CAS Arbiter propagates a CAS command to Level 1, the Command Arbiter will issue it to the device, otherwise, if the L2 PRE ACT Arbiter propagates a PRE or ACT the L1 Arbiter will issue it. This is done to ensure that the critical timings of CAS commands in the rank-switching mechanism are not disrupted by command bus contention with PRE/ACT commands. The following rules capture the behavior of the Level 2 arbiters and of the Level 3 arbiters for a critical rank ***r***.

(1A) A command at the head of each per-requestor queue is said to be active if all timing constraints that are caused by previous commands of the same requestor are satisfied;

(1B) A CAS command does not become active until the data of the previous CAS command of the same requestor has been transmitted. In other words, an active command can be issued immediately if there are no other requestors in the system.

(2A) The L3 PRE ACT Arbiter uses a modified First-Come-First-Serve (FCFS) arbitration; The requestor is en-queued at the back of a modified FIFO Queue as soon as it has an active PRE or ACT command, and it is removed from the queue once the command is finally issued by L1.

(2B) Every clock cycle, the arbiter scans the modified FIFO Queue and propagates to Level 2 the first command that can be issued (without violating timing constraints), if any.

(2C) An active PRE command can always be issued; an active ACT command could instead by blocked by t_{RRD} or t_{FAW} constraints caused by other requestors in the same rank.

(3) The L3 CAS Arbiter uses standard FCFS arbitration, with a requestor being en-queued once it has an active CAS command and removed once the CAS command is issued by L1. The L3 CAS Arbiter propagates to L2 the CAS command of the first requestor in FCFS order (if any) together with the earliest time t_{SDr} at which the data transmission associated with the CAS command could be started. The t_{SDr} is calculated based on previous CAS commands already issued either from the same or a different rank. Note that contrary to L3 PRE ACT Arbitration, it is allowed to propagate a CAS command that cannot yet be issued; this is required to properly alternate among ranks.

(4) The L2 PRE ACT Arbiter can use either FCFS or Round-Robin (RR) arbitration; we adopt RR in our prototype since it is easier to implement in hardware than FCFS.

(5) The L2 CAS Arbiter uses a different, modified FCFS arbitration; a rank is en-queued at the back of a FIFO queue once a new CAS command is propagated from L3, and it is removed from the FIFO once the command is issued by L1. Let t_{ED} be the time at which the data transmission of the last issued CAS command will end, or has ended. Then at every clock cycle, if for any queued rank it holds $t_{SDr} \leq t_{ED} + t_{RTR}$, the first such rank in FCFS order is selected. Otherwise, the first rank in FCFS order with the smallest value of t_{SDr} is selected. In either case, the corresponding CAS command is propagated to L1 only if it can be issued in the current clock cycle (without violating timing constraints).

(6) The L1 arbiter receives all the commands such as PRE, ACT, REF and CAS. This arbiter is designed as a priority arbiter where CAS command is given higher priority than PRE, ACT, REF commands. The acknowledgement (ACK) is generated at this level when the commands are sent out from this arbiter. This ACK signal is used by level 3 to schedule the DRAM timing of the commands in an orderly manner.

For the Critical requestors, each requestor puts its requests into the L3 CMD queue only after its previous request has been successfully completed with either read or writes data. Otherwise, the request within the same requestor is blocked until the read or write data of the previous request of the same requestor is completed. Note that since each requestor has at most one active command and each L3 PRE ACT or CAS Arbiter only propagates one command at a time, it follows that only one instance of each requestor or rank can be present in a given FCFS queue; after a command of that requestor/rank is issued by L1, the requestor or rank can be re-en-queued at the back of the queue.

Hence, while the system is backlogged the scheme approximates a fair arbitration where each rank is allowed to transmit once every R times, and thus each requestor within that rank transmits once every R · Mr Times.

Exceptions are made in Rules 2 and 5. The modified FCFS arbitration of Rule 2 ensures that PRE commands do not have to suffer from t_{RRD} or t_{FAW} constraints; if the first requestor has an active ACT command that cannot be issued right away, we still allow the rank to propagate a PRE command of a later requestor, since issuing the PRE command cannot delay the ACT command of the first requestor in any case. The modified FCFS arbitration of Rule 5 implements the rank-switching mechanism for CAS commands as long as the "burst to burst gap" between successive data transmission is at most t_{RTR}, ranks are scheduled in FCFS order. However, if scheduling the first rank would result in a longer gap (in particular, because of a t_{WTR} constraint), then we reorder ranks to avoid stalling the data bus.

We make no assumption on arbitration for non-critical ranks, outside of the fact that the Level 3 arbiter will propagate at most one issuable PRE/ACT command and one CAS command with associated time t_{SDr} to Level 2 every clock cycle; rank-level arbitration ensures that the worst-case latency for a request of a critical requestor depends only on the total number of ranks R and the number of requestors Mr within the same rank. L3 arbitration for non-critical requestors can be optimized for average case latency and throughput. In particular, we can use techniques employed by high-performance commercial controllers such as per-bank queues rather than private banks, and request reordering to favor load over store and open over close requests.

Finally, due to space limitations we only briefly discuss the issue of data sharing; more details on our approach are discussed in [15]. If critical cores are sharing data, we allocate a separate shared bank partition and use an additional "virtual" critical requestor to manage accesses to the shared partition; contention between data-sharing cores is then handled in the front end. For I/O communication, DMA is treated as a separate requestor. A communicating core can then access the DMA bank partition while the DMA is not transmitting.

This chapter has shown the backbone structure for our proposed memory controller design. The next section, we will look into the theoretical analysis done to analyse the design structure and the timing constraint from the theoretical point of view.

Chapter 4

Theoretical Analysis

Based on the arbitration rules detailed in Section 3.2, we will now show how to derive a safe upper bound on the latency of each memory request of a critical requestor assigned to rank ***r***. In particular, we consider the back end worst case latency t^{Req} measured from the time when a request arrives at the front of the per-requestor command queue until its data is transmitted. As shown in [2], such latency can then be used to derive the overall delay suffered by a task due to main memory contention; for example, we can use the static analysis method described in [10] to obtain the worst-case numbers of open/close and load/store requests, which let us derive a worst-case request pattern for the task. Since the same strategy in [2] can be used to account for refresh operations, we do not cover them here. We adopt the DRAM latency analysis framework introduced in [2].

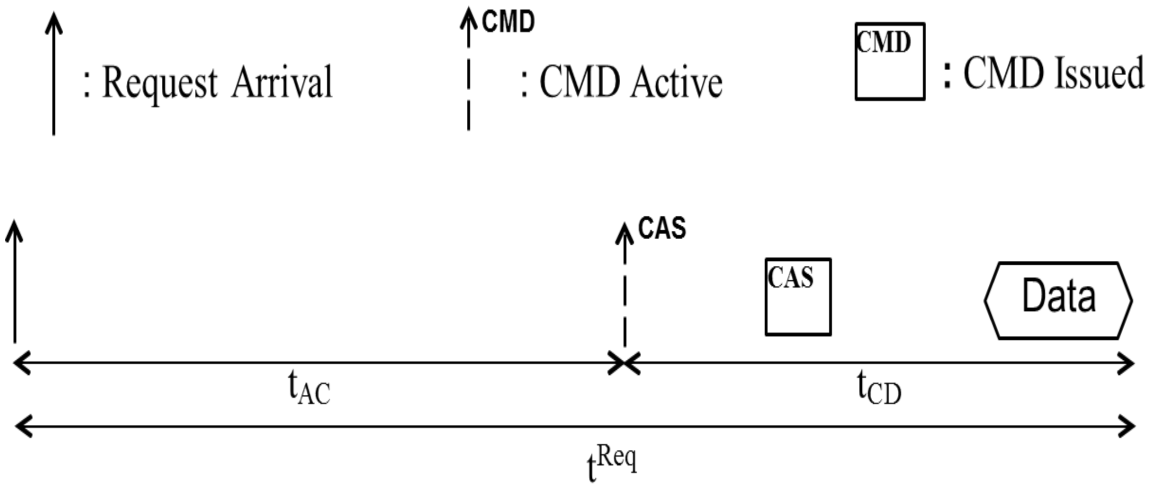


Figure 4.1: Worst Case Latency Decomposition

4.1 Worst Case Per-Request Latency

The worst case latency t^{Req} is decomposed into two parts, t_{AC} and t_{CD} as shown in Figure 4.1. Time t_{AC} (Arrival-to-CAS) is the worst case interval between the arrival of a request at the front of the per-requestor command queue and when the corresponding CAS command becomes active. The t_{CD} (CAS-to-Data) is the worst case interval between the CAS becoming active and the end of data transfer. In all figures in this section, we use a solid arrow to indicate when a request arrives at the front of the per-requestor command queue; we use a dashed arrow to indicate the time instant at which a command becomes active; solid square boxes denote when commands are issued on command bus; dashed square boxes denote commands that are ready to be issued but cannot be issued right away due to contention with other requestors.

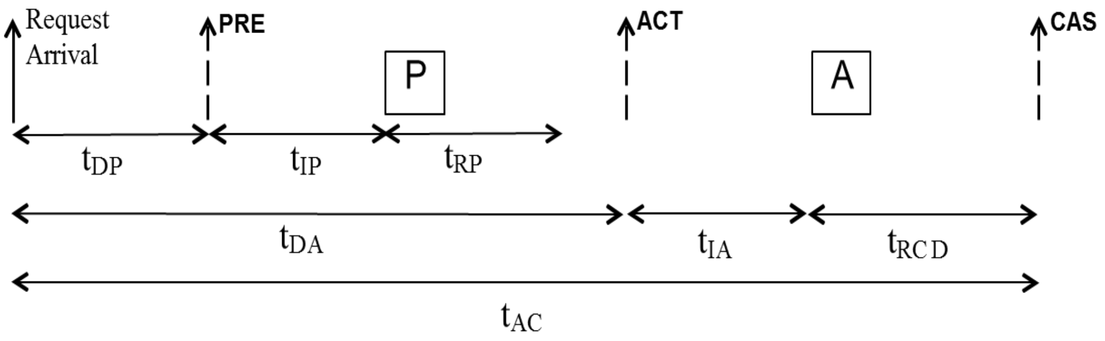


Figure 4.2: Arrival-to-CAS Decomposition for Close Request

For a close request, t_{AC} includes the latency required to process a PRE and ACT command; we thus further decompose t_{AC} into smaller parts as shown in Figure 4.2. Each part is either a JEDEC timing constraint shown in Table I or a parameter that we compute, as shown in Table 4.1. Both t_{DP} and t_{DA} determine the time at which a PRE and ACT command becomes active, respectively. t_{IP} and t_{IA} represent the worst case delay between a command becoming active and when that command is issued, and thus capture interference caused by other requestors. Times t_{DP} , t_{DA} as well as t_{AC} for an open request are computed based only on timing constraints caused by the previous request of the requestor under analysis, and are independent of the specific arbitration used by the memory controller; hence, we can reuse the expressions provided in [2]. Instead, in the following Sections, we will detail how to compute t_{IP} , t_{IA} and t_{CD}.

Timing Parameter Definitions	
t_{DP}	End of previous DATA to PRE Active
t_{IP}	Interference Delay for PRE
t_{DA}	End of previous DATA to ACT Active
t_{IA}	Interference Delay for ACT

Table 4.1: Timing Parameter Definition

Once all timing components have been computed, the value of $\mathbf{t_{AC}}$ for a close request is obtained as:

$$\mathbf{t_{AC} = max\ (t_{DA}, t_{DP} + t_{IP} + t_{RP}) + t_{IA} + t_{RCD}} \qquad \mathbf{(1)}$$

and for both open and close requests we simply compute the overall latency as $t^{Req} = t_{AC} + t_{CD}$.

4.1.1 Interference Delay for PRE and ACT Commands

We begin by computing the worst-case interference delay for PRE commands. We limit ourselves to devices for which the relation $\mathbf{t_{RTR}} \geq \mathbf{t_{RL}} - \mathbf{t_{WL}}$ holds, which includes all devices except the one with the largest timing constraints, i.e., the least performance ones in each speed category, which are rarely used. The relation ensures that no more than one CAS command can be issued every $\mathbf{t_{BUS}}$ cycles, despite the fact that $\mathbf{t_{RL}}$ is generally larger than $\mathbf{t_{WL}}$; this helps bounding the maximum delay suffered by PRE and ACT commands due to Level 1 arbitration. It has the benefit of simplifying the proofs. We begin by determining the maximum delay suffered by PRE and ACT commands due to L1 arbitration. Note that due to space limitations, some proofs are provided in appendix.

Theorem 1: The worst case value for t_{IP} is:

$$\mathbf{t_{IP} = \alpha_{PA}(R \cdot M_r) - 1} \qquad \textbf{(2)}$$

$$\text{where } \alpha_{PA}(K) = K + \left\lceil \frac{K}{(t_{BUS} - 1)} \right\rceil$$

Proof: Note that there are no interfering constraints between the PRE under analysis and commands by other requestors, since they must target different banks. Since furthermore arbitration Rule 2 ensures that commands blocked by timing constraints are not considered for arbitration, it follows that the PRE under analysis can only be delayed due to contention on the command bus, i.e., the command bus must be continuously in use between the en-queuing of the requestor under analysis and when its PRE command is issued. In the worst case, when the requestor under analysis is en-queued into the L3 PA Arbiter FCFS queue, there can be a maximum of Mr - 1 preceding requestors in the queue. Note that requestors en-queued after the requestor under analysis cannot delay it; and after a PRE/ACT command is issued, the corresponding requestor can only be re-en-queued at the end of the queue. Hence, each other requestor in rank ***r*** can only issue one PRE/ACT command before the requestor under analysis, leading to a total of Mr PRE/ACT commands from rank ***r***, including the PRE under analysis. Furthermore, since the L2 PA Arbiter uses either FCFS or round robin arbitration, in the worst case R - 1 PRE/ACT commands of other ranks must be issued before any command of rank ***r***. Hence, the worst case number of issued PRE/ACT commands is (R - 1) Mr + Mr = (R • Mr), and the L2 PA Arbiter is backlogged while issuing them. Based on Lemma 1, the worst case time required to issue all R•Mr commands is then α_{PA}(R • Mr). To conclude the proof, it suffices to notice that t_{IP} does not include the extra clock cycle required to transmit the PRE under analysis; hence, $t_{IP} = \alpha_{PA}$ (R · Mr) - 1. Note that t_{IP} depends on the number of requestors Mr in rank 'r' but it is independent from the number of requestors assigned to other ranks; this is because L2 arbitration isolates rank ***r*** from requestors in other ranks. We will show that the same is true for the derived t_{IA} and t_{CD}, hence making our analysis compositional.

We next analyze t_{IA}. We prove that the ACT command under analysis suffers maximal delay in the scenario shown in Figure 4.3, where R = 2 and the rank under analysis is r = 1 with M_r = 5. The worst case is produced when all M_r - 1 other requestors of rank ***r*** en-queue an ACT command at the same time t_0 as the core under analysis, which is placed last in the L3 PA Arbiter FCFS order; each other requestor triggers a t_{RRD} timing constraint. Furthermore, four ACT commands have been completed as late as possible before t_0; this forces the first ACT after t_0 to wait for t_{FAW}- 4· t_{RRD} before being propagated to Level 2. Once an ACT has been propagated to L2, in the worst case it will have to wait for R - 1 PRE/ACT commands of other ranks and for interfering CAS commands, similarly to the case of PRE commands in Theorem 1; we call this delay Δ_{IA}. Finally, we need to consider the effect of t_{FAW} on successive ACT commands after t_0.

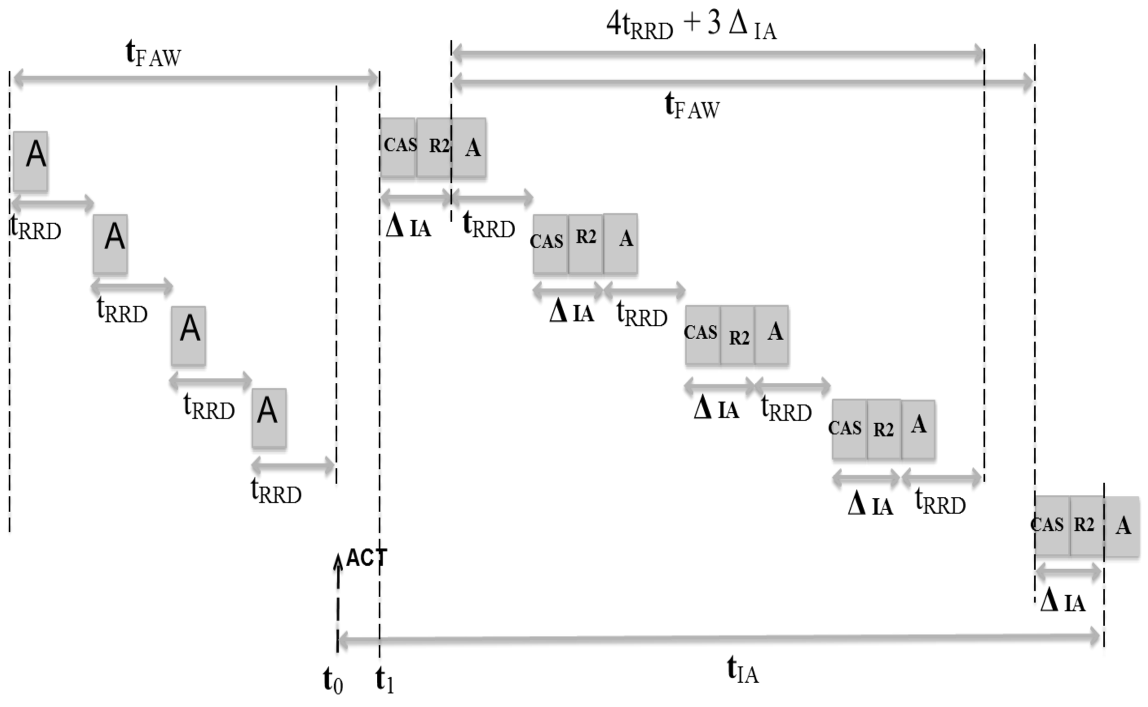


Figure 4.3: Interference Delay for ACT command, *R* = 2, *r* = 1 and Mr = 5

As shown in Figure 4.3, since the t_{FAW} applies from the time when an ACT is issued to the time when the fourth following ACT can be propagated to L2, we have to take the maximum of either t_{FAW} or $4.t_{RRD} + 3.\Delta_{IA}$ for every 4 ACT of rank ***r*** issued before the one under analysis. ■

Theorem 2: The worst case value for t_{IA} is:

$$\mathbf{t_{IA} = t_{FAW} - 4\, t_{RRD}} + \max\left(\mathbf{(M_r} - 1\mathbf{)t_{RRD} + M_r \cdot \Delta_{IA}, K \cdot t_{FAW} + (M_r - 1 - 4K)\, t_{RRD} + (M_r - 3K)\, \Delta_{IA}} \right) \qquad \mathbf{(3)}$$

Where $\Delta_{IA} = \alpha_{PA}(R) - 1$ and $K = \left\lfloor (M_r - 1)/4 \right\rfloor$

Proof: Let t0 be the time at which the requestor with the ACT under analysis is en-queued in the L3 PA Arbiter FCFS queue. We show that the worst case latency for the ACT under analysis is produced when at time t_0 there are (Mr – 1) other requestors en-queued before the requestor under analysis, all with ACT commands.

First note that requestors en-queued after the ACT under analysis cannot delay it: if the ACT under analysis is blocked by the tRRD or tFAW timing constraint, then any subsequent requestor with an ACT command in the L3 PA Arbiter FCFS queue would also be blocked by the same constraint. Requestors with PRE commands en-queued after the requestor under analysis can be issued before it according to arbitration Rule 2 if the ACT under analysis is blocked, but they cannot delay it because those requestors access different banks, and there are no timing constraints between ACT and PRE of a different bank. Furthermore, after a PRE/ACT command is issued, the corresponding requestor can only be re-en-queued at the end of the queue. Hence, each of the other Mr - 1 requestors on rank ***r*** can only delay the requestor under analysis by one command, either ACT or PRE. A PRE command can only interfere with the ACT under analysis due to command bus contention, i.e., one bus cycle. On the other hand, each ACT of another requestor en-queued before the requestor under analysis can contribute to its latency for at least a factor tRRD, which is larger than one clock cycle on all devices. This shows that the worst case is produced when all other requestors on rank ***r*** have ACT commands.

Second, we show that all requestors of rank ***r*** en-queuing their ACT command at the same time t_0 are the worst case pattern. Requestor en-queuing an ACT after t_0 does not cause interference as already shown. If an requestor en-queues an ACT at time $t_0 - \Delta$ with $\Delta < t_{RRD}$, the overall latency is reduced by Δ since the requestor cannot en-queue another ACT before t_0 due to arbitration Rule 1 (the next ACT would not be active due to t_{RRD}).

Third, we consider the latency of ACT commands issued after t_0 due to t_{RRD} and L2/L1 arbitration; similarly to the proof of Theorem 1, each ACT command of rank ***r*** can suffer command bus contention delay of $\Delta_{IA} = \Delta_{PA}(R) - 1$ (as an example, $\Delta_{IA} = 2$ in Figure 4.3). Furthermore, once an ACT command of rank ***r*** is issued, notice that the next ACT command of the same rank ***r*** cannot be propagated from L3 to L2 until after the t_{RRD} constraint has elapsed; hence, each ACT command can take $\Delta_{IA} + t_{RRD}$ before being issued.

Finally, we consider the effect of the t_{FAW} timing constraint. Note that a requestor could issue an ACT at or before t_0- t_{RRD} and then en-queue another ACT at t_0 before the ACT under analysis. Due to the t_{FAW} constraint, ACT commands after t0 could then suffer additional delay. Since the t_{FAW} constraint is activated by four consecutive ACT commands, the worst case is produced when four ACT commands are issued as late as possible before t_0, as shown in Figure 4.3. The first ACT after t0 is then blocked until time $t_1 = t_0 + t_{FAW} - 4 \cdot t_{RRD}$. Note that similarly, the second ACT after t_0 cannot be propagated from L3 to L2 before $t_0 + t_{FAW} - 3t_{RRD} = t_1 + t_{RRD}$ due to the same constraint; however, this constraint does not affect the worst case pattern since the second ACT after t_0 is blocked until $t_1 + \Delta_{IA} + t_{RRD}$ anyway due to the t_{RRD} constraint generated by the first ACT and L2/L1 arbitration. It remains to consider the case when t_{FAW} is activated by ACT commands of rank ***r*** issued after t0. Since t_{FAW} applies from the time when an ACT of rank ***r*** is issued to the time when the fourth next ACT of rank ***r*** can be propagated from L3 to L2, if the constraint is activated it effectively replaces the delay of four t_{RRD} constraints (generated by the CAS that starts t_{FAW} and the next three CAS commands of rank ***r***) and three Δ_{IA} times (for each of the next three CAS; see also the example in Figure 4.3). Furthermore, the total number of t_{FAW} constraints that can be activated for CAS commands of rank ***r*** after t1 is K = b(Mr - 1) = 4c, since we need at least four CAS commands to block the fifth one.

In summary, if $t_{FAW} \leq 4 \cdot t_{RRD} - 3\Delta_{IA}$, then t_{FAW} is not activated after t_0 and the final bound on t_{IA} is then obtained by summing the delay $t_1 - t_0$, Mr - 1 times the delay t_{RRD} (once for each other requestor on rank ***r***), and Mr times the delay Δ_{IA} (once for each other requestor on rank ***r*** plus once for the requestor under analysis), yielding a bound: $t_{FAW} \cdot 4t_{RRD} + (Mr\text{-}1)t_{RRD} + Mr \cdot \Delta_{IA}$. If instead $t_{FAW} \geq 4t_{RRD} - 3\Delta_{IA}$, the bound on t_{IA} can be obtained as: $t_{FAW} - 4t_{RRD} + K \cdot t_{FAW} + (Mr - 1 - 4K)t_{RRD} + (Mr - 3K)\Delta_{IA}$, where for each of the K times the t_{FAW} constraint is activated, we replace a term $4t_{RRD} + 3\Delta_{IA}$ with a term t_{FAW}. To end the proof, it suffices to notice that in Eq.(3) we consider the maximum of the two bounds. ■

4.1.2 CAS-to-Data

We now focus on computing a bound on t_{CD} for a request using rank ***r***. Similarly to the case of t_{IA}, we prove that the current request suffers worst case interference when all $M_r - 1$ other requestors have an active CAS command arriving at the same time t_0 as the requestor under analysis, which is then serviced last according to FCFS arbitration. Our proof scheme proceeds as follows. We first compute the delay for successive CAS commands of rank ***r***. Specifically, Lemma 1 computes the delay for a read followed by a read and a write followed by a write (which we denote as t_{RRD} and t_{WWD}, respectively), while Lemma 2 covers the cases of write-to-read transition and read-to-write transition (t_{WRD} and t_{RWD}), which are more complex due to the t_{WTR} and t_{RTW} constraints. Then, Lemma 3 computes the delay for the first CAS of rank ***r*** issued after t_0. Finally, Theorem 3 uses the computed delays to derive the final value of t_{CD}. The timing constraints that contribute to the worst case latency are shown as solid black horizontal arrows.

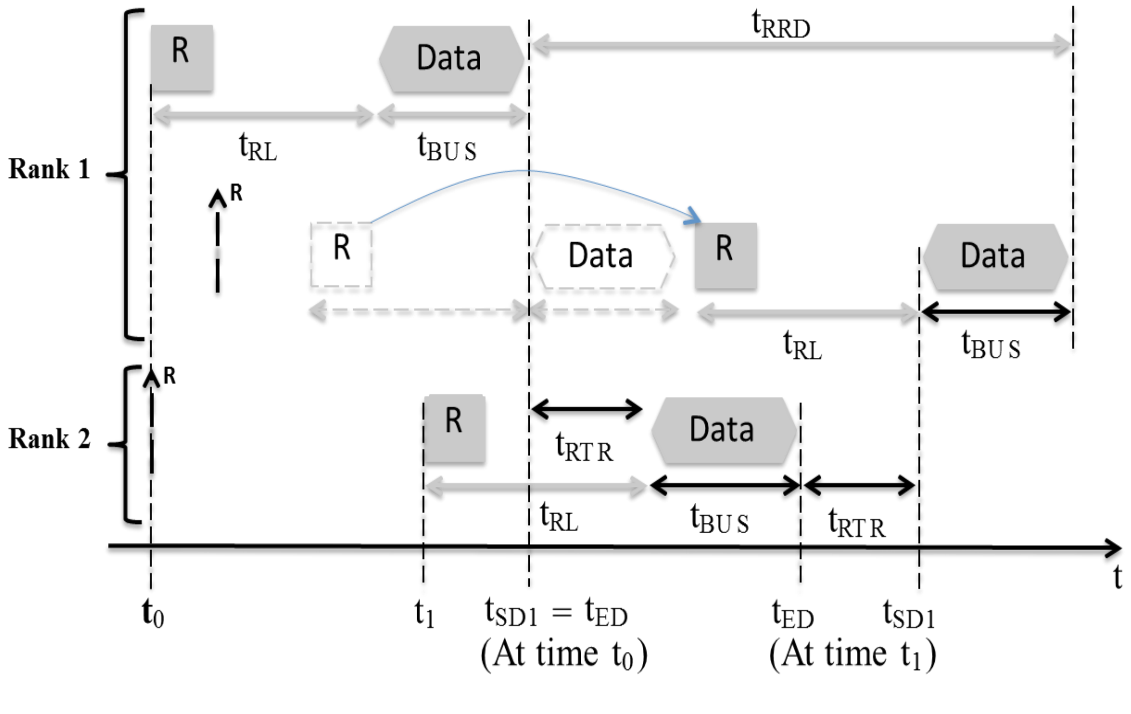


Figure 4.4: Read to Read Latency, R = 2 and r = 1

Lemma 1: Assume that the L3 CAS Arbiter for rank ***r*** prop-agates a read command to L2 immediately after a previous read command of rank ***r*** is issued (i.e., the L3 CAS Arbiter is backlogged). Then the worst case latency between the completion of data transmissions for the first read command and for the second read command is:

$$\mathbf{t_{RRD} = R(t_{BUS} + t_{RTR}) \qquad (4)}$$

Similarly, for the case of a write followed by a write, the worst case latency is $t_{WWD} = t_{RRD}$.

Proof: We prove the lemma for t_{RRD}; the proof for t_{WWD} is equivalent, by exchanging read with write commands and t_{RL} with t_{WL}.

Let t_0 be the time at which the first read command of rank ***r*** is issued; then by definition after t_0, $t_{ED} = t_0 + t_{RL} + t_{BUS}$ (see Figure 4.4 above).

Since there are no timing constraints between consecutive read commands of the same rank, the second read command of rank ***r*** (dashed boxes in Figure 4.4) could start data transmission at time $t_{SDr} = t_{ED}$ if other ranks were not serviced before it.

After the first read command is issued at time t_0, rank ***r*** will be re-en-queued at the back of the L3 CAS Arbiter FIFO at time $t_0 + 1$; in the worst-case, $R - 1$ ranks can be en-queued before the rank under analysis. Note that whenever another rank issues a CAS command after t_0, the value of t_{ED} will be updated; due to the t_{RTR} timing constraint between different ranks, the value of t_{SDr} will instead be updated to $t_{ED} + t_{RTR}$ (see the example in Figure 4.4 after a CAS of rank 2 is issued at time t_1). In any case, the condition t_{SDr} $t_{ED} + t_{RTR}$ always hold. Due to this reason and based on Arbitration Rule 5, each of the other $R - 1$ ranks can issue at most one CAS command before the second read of rank ***r*****.** Furthermore, each such $R - 1$ data transmissions (let us say, of rank j) must begin at most t_{RTR} time units after the previous data transmission has finished; otherwise, the condition t_{SDj} $t_{ED} + t_{RTR}$ would be violated and rank j could not issue a CAS before rank ***r*** according to Rule 5. In summary, at most R CAS commands must be issued, including the second read of rank ***r***, and each data transmission incurs a delay of at most $t_{RTR} + t_{BUS}$. Hence, the lemma follows. ■

Lemma 2: Assume that the L3 CAS Arbiter for rank ***r*** prop-agates a read command immediately after a write command of rank ***r*** is issued. Then the worst case latency between the completion of data transmissions for the write command and for the read command is:

$$\mathbf{t_{WRD} = \max(R\,(t_{BUS} + t_{RTR})\,,\; t_{WTR} + t_{RL} + 2\,t_{BUS} + t_{RTR} - 1)} \qquad \textbf{(5)}$$

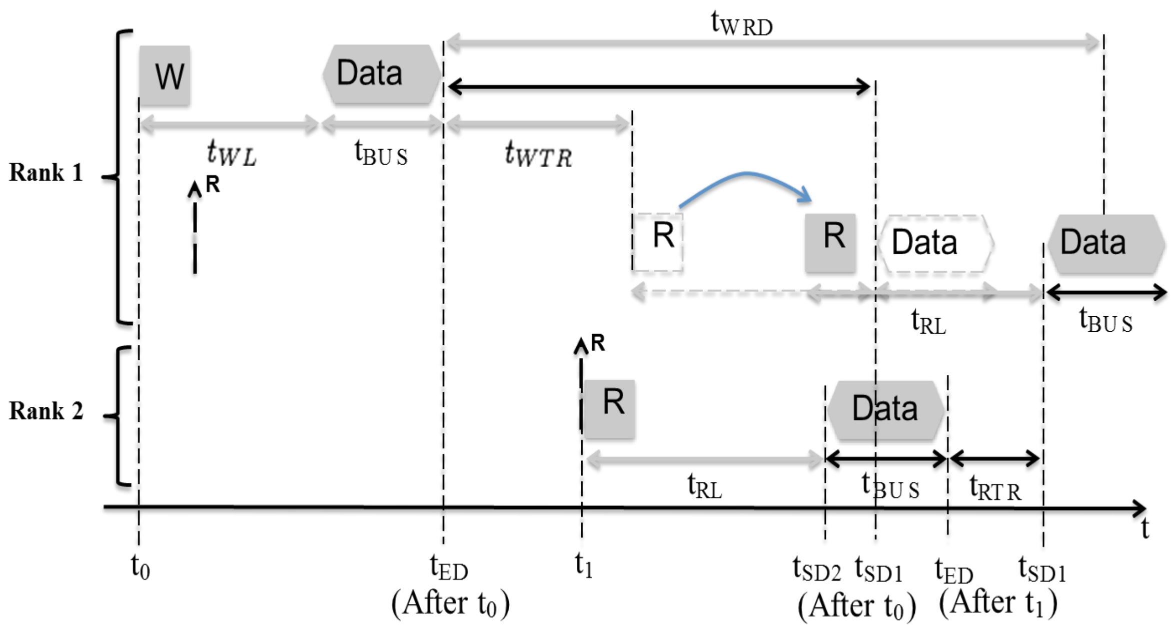


Figure 4.5: Write to Read Latency, Case (a) with R = 2 and r = 1

Similarly, for the case of a read followed by a write, the worst case latency is:

$$\mathbf{t_{RWD} = \max\,(R\,(t_{BUS} + t_{RTR}),\, t_{RTW} + t_{WL} - t_{RL} + t_{BUS} + t_{RTR} - 1)} \qquad \textbf{(6)}$$

Proof: We first compute t_{WRD}. Let t_0 be the time at which the write command of rank ***r*** is issued; then by definition, the CAS Arbiters set $t_{ED} = t_0 + t_{WL} + t_{BUS}$ (see Figure 4.5 above). Due to the t_{WTR} constraint, the L3 CAS Arbiter of rank ***r*** will also set a time $t_{SDr} = t_{ED} + \Delta$ for the start of the successive read command, with $\Delta = t_{WTR} + t_{RL}$. Since t_{WTR} and t_{RL} are larger than t_{RTR} and differently from Lemma 1, we have $t_{SD}r > t_{ED} + t_{RTR}$. We consider two possible cases.

Case A: In this case, the read command of rank ***r*** is delayed by a CAS command of another rank j en-queued after r in the L2 CAS Arbiter FCFS order. This is possible if $t_{SDj} < t_{SDr}$; in the worst case shown in Figure 4.5, $\mathbf{t_{SDj} = t_{SDr} - 1}$, resulting in a latency $t_{WRD} = \Delta - 1 + t_{BUS} + t_{RTR} + t_{BUS.}$ Note that after the rank under analysis is delayed by a command of j, it will hold $\mathbf{t_{SDr} = t_{ED} + t_{RTR}}$ and thus rank ***r*** cannot be delayed by another rank en-queued after it.

Case B: The read command of rank r is delayed by CAS commands of ranks en-queued before r in the L2 CAS Arbiter FIFO, similarly to the case in Lemma 1. Note that for a rank j to be en-queued before r in the FIFO, the CAS command of rank j must have been propagated to Level 2 before or at time t0 + 1 (dashed arrow for Rank 1 in Figure 4.6 below). We distinguish two sub cases within Case B: Case B_1, the CAS command of rank j is not delayed by a t_{WTR} timing constraint. In this case, the data transmissions of rank j can start at $t_{ED} + t_{RTR}$. For example, see Rank 3 in Figure 4.6. In Case B_2, a previous write command of rank j has been issued before t0, and the successive read command is thus delayed by the t_{WTR} constraint (Rank 2 in Figure 4.6). In this case, the read command of rank j could be associated with a value $t_{SDj} > t_{ED} + t_{RTR}$. However, since the preceding write command of rank j must have completed its data transmission at least $t_{BUS} + t_{RTR}$ before the write command of rank r completes its data transmission, it must also hold that the difference between t_{SDj} and t_{SDr} is at least $t_{BUS} + t_{RTR}$ (see the dotted boxes in Figure 4.6 below). Hence, rank j alone cannot delay the read command of rank r, unless there are other ranks that can start data transmission at $t_{ED} + t_{RTR}$. In either sub case, it follows that the read of rank r can only be delayed if other ranks continuously transmit data every $t_{BUS} + t_{RTR}$ time units starting at $t_{ED} + t_{RTR}$. Furthermore, following the same reasoning as in Lemma 1, in this case no rank en-queued after rank r can cause delay on rank r. Hence, we obtain the same expression as for t_{RRD}, i.e. $t_{WRD} = R\,(t_{BUS} + t_{RTR})$. Finally, taking the maximum of Case A) and B) will yield Equation (5).

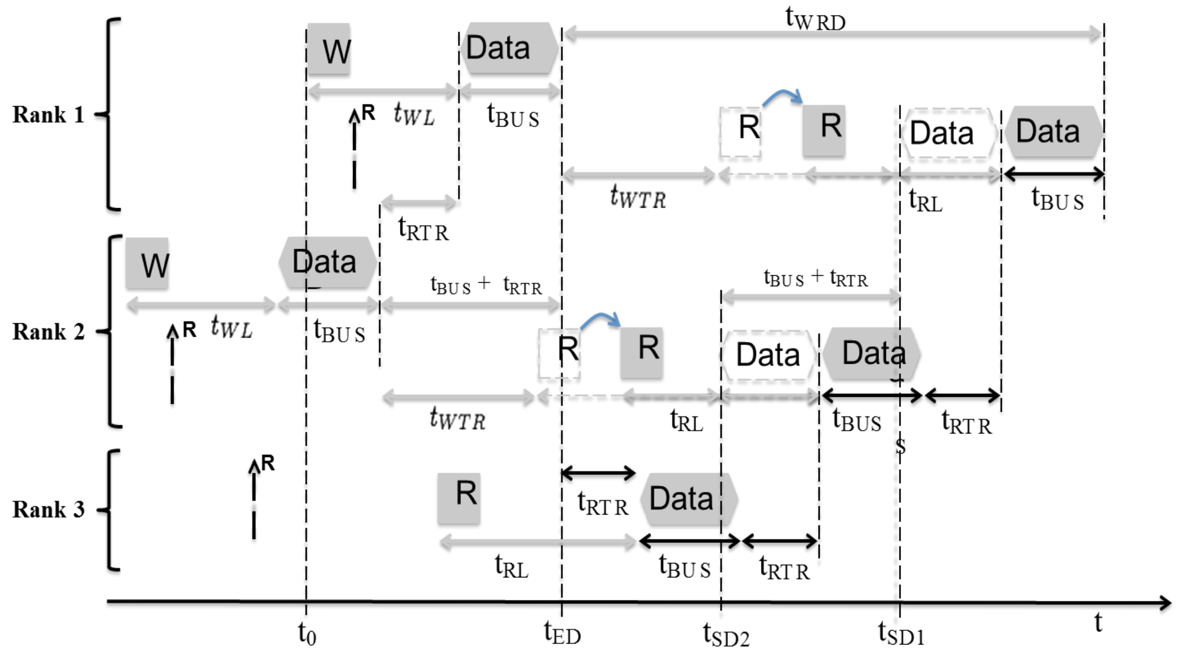


Figure 4.6: Write to Read Latency, Case b) with R = 3 and r = 1

For $\mathbf{t_{RWD}}$, it suffices to note that the distance Δ between the end of data transmission for the read and the start of data for the successive write is $\Delta = t_{RTW} + t_{WL} - t_{RL} - t_{BUS}$ Again, taking the maximum of Case A) and B) will yield Equation (6). ■

It is interesting to note that for the DDR3-1333H device in Table 2.1 and for R = 4, the term $R(t_{BUS} + t_{RTR})$ in Eq.(5), (6) is maximal, meaning $t_{WRD} = t_{RWD} = t_{RRD} = t_{WWD}$; hence, in this condition ROC guarantees a data bus utilization of $t_{BUS}/(t_{BUS} + t_{RTR}) = 2/3$ to a backlogged system. Furthermore, the worst-case latency is completely unaffected by the t_{WTR} and t_{RTW} timing constraints.

Lemma 3: Assume that a CAS of the requestor under analysis in rank ***r*** becomes active at time t_0, and that at t_0 there are other Mr −1 requestors with active CAS commands before it in the L3 CAS Arbiter FCFS order.

Then if the first CAS of rank ***r*** issued after t_0 is a read, the worst case latency between t_0 and the completion of data transmission for the first read command is:

$$\mathbf{t_{RD} = max(t_{RL} + t_{BUS} - 1 + R\ (t_{BUS} + t_{RTR}),\ t_{WTR} + t_{RL} + 2\ t_{BUS} + t_{RTR} - 1);} \qquad \textbf{(7)}$$

Otherwise if the first CAS is a write, the worst case latency is:

$$\mathbf{t_{WD} = t_{RL} + t_{BUS} - 1 + R\ (t_{BUS} + t_{RTR})} \qquad \textbf{(8)}$$

Proof: The proof is similar to Lemma 2. The main difference is that now a requestor of another rank could issue a request immediately before t_0 and still be en-queued before rank ***r*** (see Rank 2 in Figure 4.7 as an example for t_{WD}); this contributes the additional delay term $t_{RL} + t_{BUS} - 1$. ■

Theorem 3: The worst case CAS-to-Data latency for a write or read command, respectively, is:

$$t_{CD}^{\text{Write}} = \left\lceil \frac{Mr - 1}{2} \right\rceil t_{RWD} + \left\lfloor \frac{Mr - 1}{2} \right\rfloor t_{WRD} + \qquad \textbf{(9)}$$

$$\{\, t_{RD} \text{ if } Mr \text{ is even or } t_{WD} \text{ if } Mr \text{ is odd} \,\};$$

$$t_{CD}^{\text{Read}} = \left\lceil \frac{Mr - 1}{2} \right\rceil t_{WRD} + \left\lfloor \frac{Mr - 1}{2} \right\rfloor t_{RWD} + \qquad \textbf{(10)}$$

$$\{\, t_{WD} \text{ if } Mr \text{ is even or } t_{RD} \text{ if } Mr \text{ is odd} \,\};$$

Proof: We show that the pattern in Lemma 3 results in the worst case latency t_{CD}; intuitively, we maximize the number of requestors of rank ***r*** that interfere with the requestor under analysis. Hence, we can compute the latency for the first CAS of rank ***r*** after t_0 as either t_{RD} or t_{WD}; for each of the other Mr − 1 requestors of rank ***r*** (including the one under analysis), we then add a term t_{RRD}, t_{WWD}, t_{WRD} or t_{RWD} based on the sequence of CAS commands.

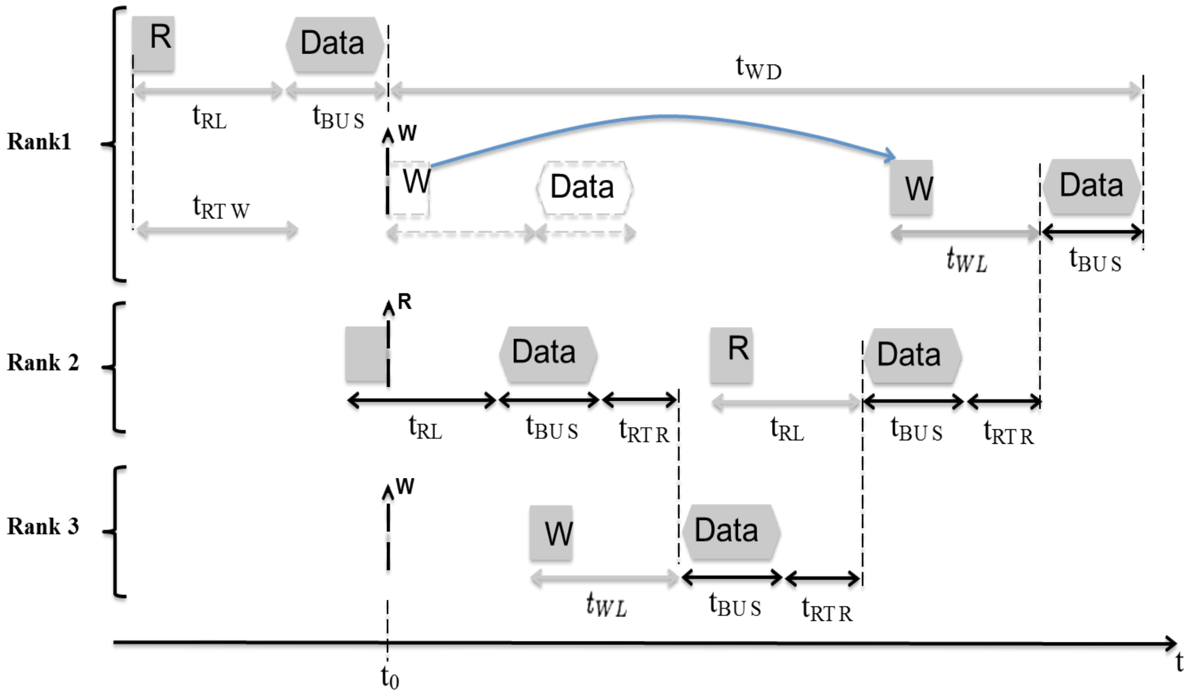


Figure 4.7: Initial Write Latency, R = 3 and r = 1

Now note that $t_{WRD} \geq t_{RRD} = t_{WWD}$ and $t_{RWD} \geq t_{RRD} = t_{WWD}$; hence, we prove that the worst case sequence is an alternation of read and write commands (also notice that $t_{RD} \geq t_{WD}$, but we prove that the effect of alternating read and write commands on the worst case latency is larger compared to starting with a read rather than a write). To conclude the proof, note that if the requestor under analysis issues a read, then in an alternating sequence of Mr Commands there are $\lceil (Mr - 1)/2 \rceil$ write-to-read transitions and $\lfloor (Mr - 1)/2 \rfloor$ read-to-write transitions, and vice-versa for a write. ■

So far, we have seen the theoretical analysis of our memory controller design. In the next Section, we show you the detail implementation of our rank-switching open-row memory controller design.

Chapter 5

Memory Controller Implementation

The proposed rank-switching memory controller consists of front end and back end as shown in Figure 5.1. This memory controller design can accept requests from number N of both critical and non-critical real time requestors where 0 < N <= 32. The front end logic receives memory requests that have the physical address and the request type in order to generate the corresponding DRAM commands such as ACT, PRE, REF and CAS. The generated commands from the front end are dispatched to the command queues in the back end. The command queues are the first clocked interface between front and back end. The back end logic is responsible for requestor arbitration, rank arbitration and command arbitration. Each level of arbitration consists of sequencers to check if the chosen command satisfies the DRAM protocol timing. Once the timing check is satisfied, it would dispatch the command to appropriate ranks, banks in the physical DDR Memory Device.

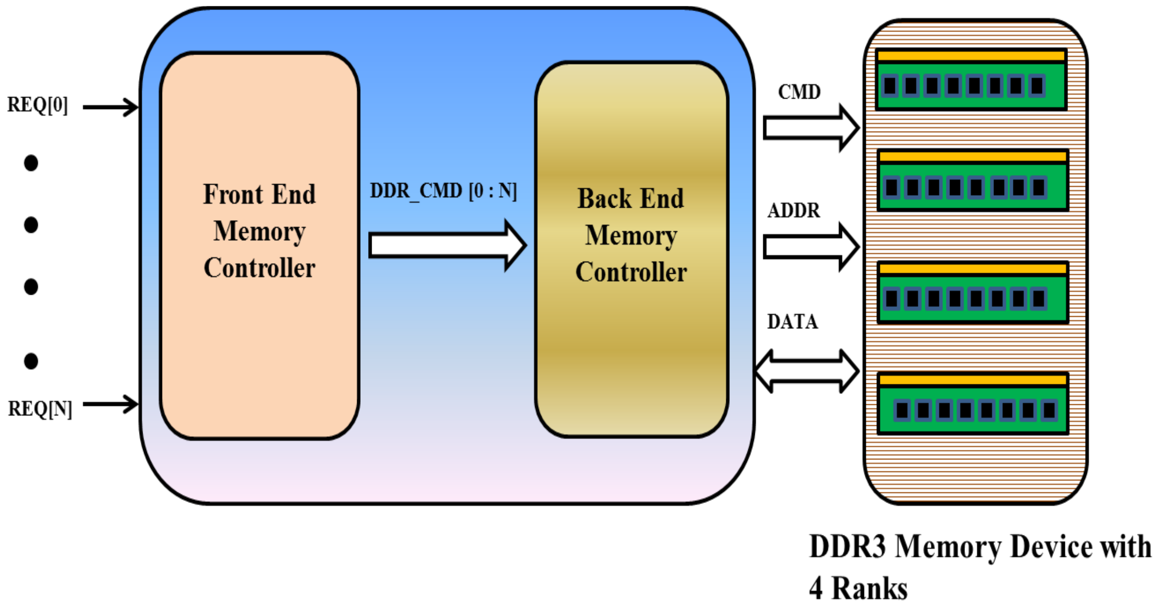

REQ[0]
REQ[N]
Front End Memory Controller
DDR_CMD [0 : N]
Back End Memory Controller
CMD
ADDR
DATA
DDR3 Memory Device with 4 Ranks

Figure 5.1: Memory Controller with Front and Back end logic.

The objective of this proposed rank switching open row memory controller design is to achieve worst case upper bound latency for critical requestors and average bandwidth for non-critical requestors. Both front end and back end logic works with one clock domain. As you can see from Figure 5.1, there are three buses such as DATA bus, ADDR Bus and CMD bus that connect our memory controller with the memory device. This CMD bus represent important memory device signals such as CS, RAS, CAS, WE. Similarly, the ADDR Bus represents the device signals such as A0 to A15. Further B0 to B2 represent the 8 banks and CS signal represents the number of Ranks as per JDEC standard. To carry out the evaluation and testing, the design was implemented in such a way that the number of requestors, ranks and banks can be customized as per user request. Next, the front end will be discussed in detail.

5.1 Front End Memory Controller

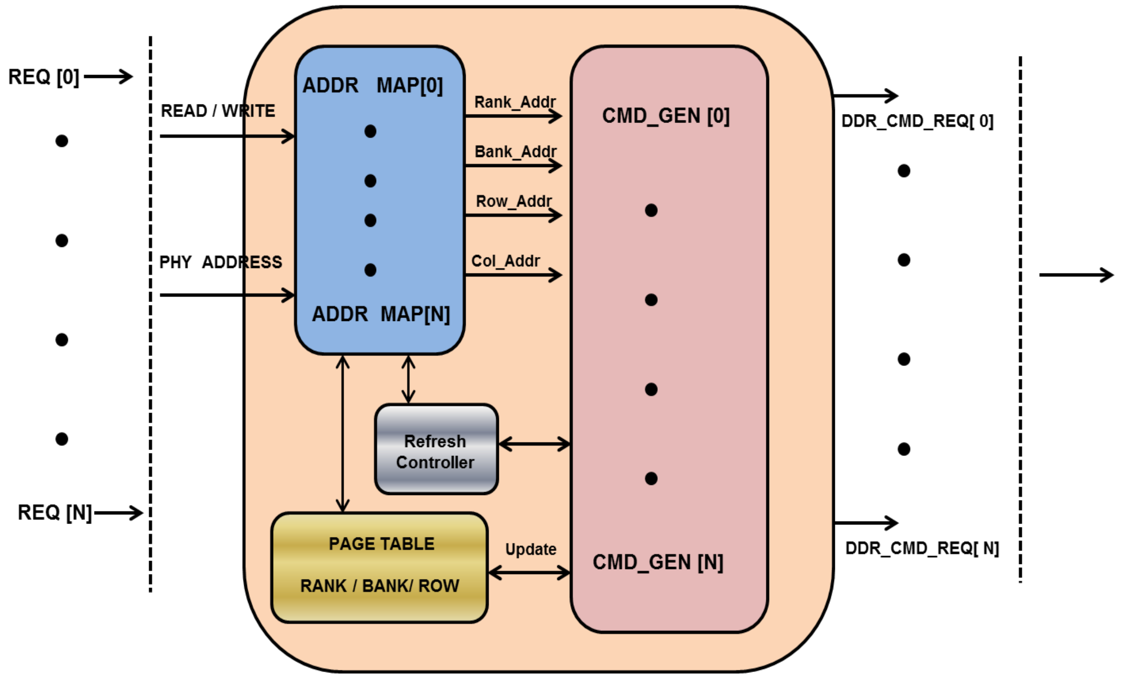


Figure 5.2: Front End Memory Controller

The front End Memory Controller consists of Address Mapping, Command Generator, Refresh Controller and Row Table as shown in Figure 5.2. The N number of requestors would require N number of address mapping logic blocks and N number of command generators in the front logic shown in Figure 5.2.

First, the incoming physical addresses from all N requestors would go through N number of Address Mapping logic blocks which would split the incoming physical addresses into normalized rank, bank, row and column Addresses. Then, the normalized rank, bank, row, column addresses are fed into their respective N number of command generators for the proper command generations. The Command Generators are responsible for generating the necessary DDR Memory commands such as PRE, ACT, REF and CAS based on the incoming request type (read or write), physical address and the status of row table that indicate if the row is open or close. When a request targets a specific rank, bank, row combination, that particular request entry is entered into the row table. The row table keep the record of which rows, banks and ranks have been accessed for each incoming request. Using the previous access record in the row table, the command generator is able to generate the right command. Further, maintaining the row status of previously accessed memory requests also enhance the command scheduling efficiency. Since private bank mapping is used, every requestor is assigned to one bank or set of banks in a rank. Two requestors cannot access the same bank in a rank. The number requestors, N, can range from $0 < N <= 32$. But, for this implementation, we considered maximum N = 16 requestors.

In this proposed design, the row table updates its entry into one of the following three scenarios in order to assist the command generators to generate the corresponding commands such as PRE, ACT, REF and CAS.

Row Conflict: A row conflict occurs when there is a new request to row in a particular bank which already has different row opened. In this scenario, the command generator would generate the following commands in order. First, it would generate PRE command to close the already opened row. Second, it needs to issue an ACT Command to open the row for the new request. Finally, it would issue the Read or write CAS command.

Row Miss: A row miss occurs when there is a request to a row in a particular bank of DRAM that does not have any row already open. The Command Generator needs to send ACT command (row open) and then Read or write CAS command.

Row Hit: A row hit occurs when there is a request to a row in a particular bank of DRAM that is already opened by previous request. In this case, there is no need to open the bank again and therefore, it simply sends read or write CAS command.

Refresh Controller

Every row in a DRAM needs to be regularly refreshed to avoid data lost which would eventually make sure the memory access predictable. Refresh process is separately handled by a refresh controller which generate the refresh command for every refresh period. Refresh period depends on the DDR Memory device. The refresh of a memory rank is partitioned into 8,192 smaller refresh operations. Such refresh operation has to be issued every 7800 ns (64 ms divided by 8192). In other words, a refresh operation must be performed every 7800 ns on average to refresh the entire DRAM in 64ms (retention time). This 7800 ns interval is referred to as the refresh interval, t_{REF}. Each Refresh operation lasts for a time limit that is referred to as the refresh cycle time, t_{RFC}, which depends on the devices. For our design simulation, DDR3 1333-H device is used where t_{RFC} = 160 ns and t_{REF} = 7800 ns. These parameter values might be different for other high end memory devices. All of the banks must be pre-charged before a refresh command is issued. During every refresh period, all the banks and ranks need to be refreshed.

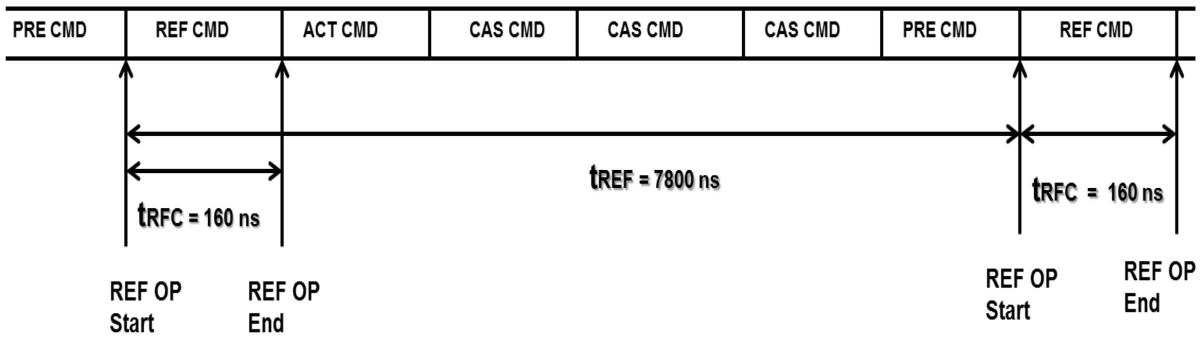


Figure 5.3: Refresh Controller Timing

5.2 Back End Memory Controller

This section will analyse the detailed implementation of each logic component that was used to build the back end logic as shown in Figure 5.4 below. The back end logic is designed as 3 levels of arbiters and 4 stages of pipeline architecture. Each level of arbiters is categorized as requestor arbitration (L1), rank arbitration (L2) and Command arbitration (L3). The design was implemented in a 4 stage pipeline architecture in order to increases the number of commands throughput by executing operations in all four stages in parallel. Each pipelined stage is separated by a sequential register element. In the next section, let us look at the logic behind command queues which are the first interface unit of this back end logic.

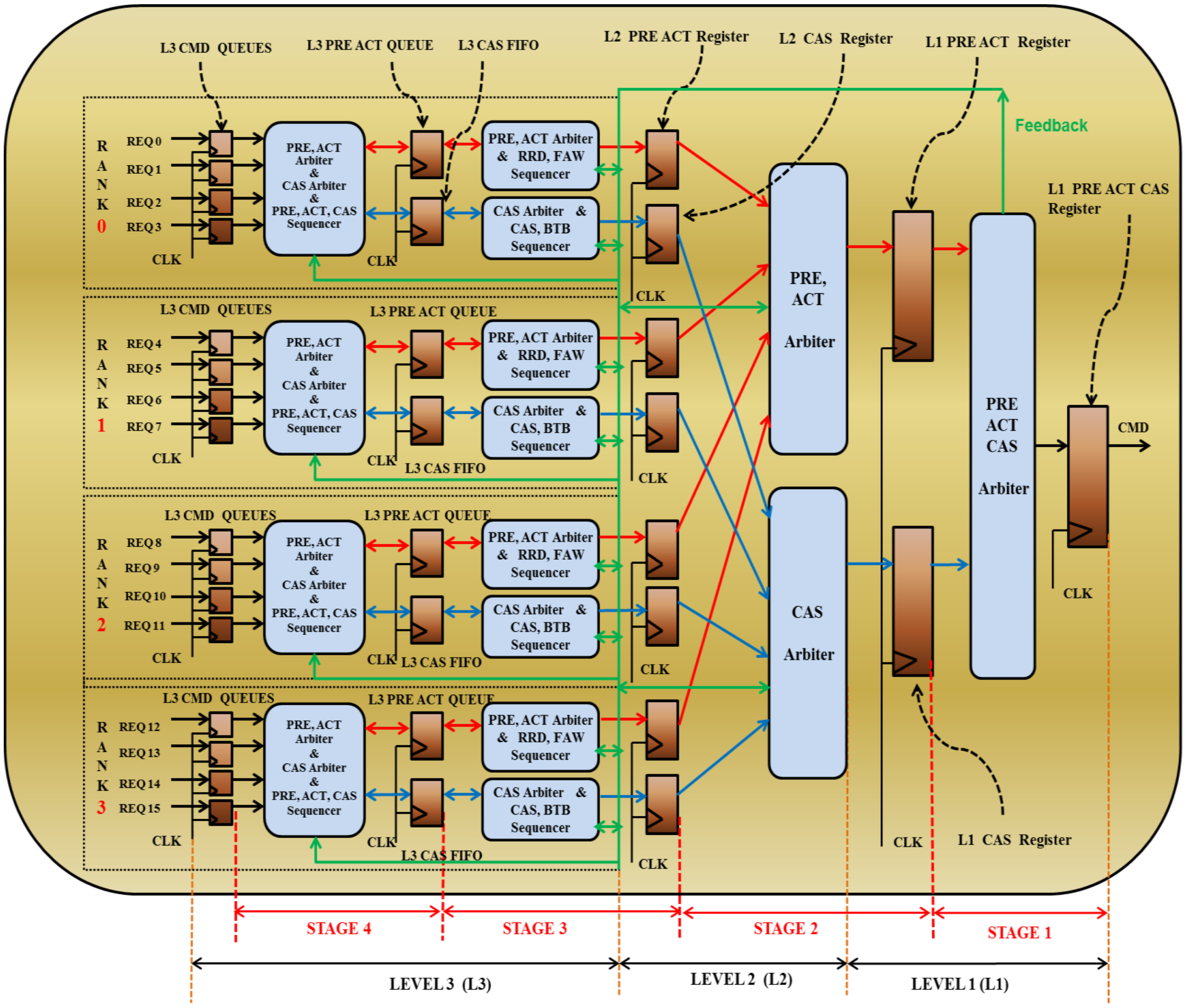


Figure 5.4: Back End Memory Controller

5.2.1 Command Queues in Stage 4

The commands generated by the front end are stored into the command queues as shown in Figure 5.5. It is the first sequential interface to the back end. The command queue controller is specially designed to increase the efficiency of this proposed memory controller in the following manner. In a regular FIFO controller design, the read enable should be sent out to the queue before reading out the data. But, this customized queue controller is designed to function like a look-ahead manner where the head of the queue is visible to the receiver so that the receiver logic is able to decide if the next command is a PRE or ACT or REF or CAS. This look-ahead feature helps to differentiate CAS versus PRE, ACT, REF commands. Therefore, this look-ahead feature process the CAS and PRE, ACT commands separately in parallel by separate arbiters in order to minimize the latency.

As outlined in the arbitration rule (1 A), the command at the head of each commands queue is active only if the corresponding timing constraints of the previous commands of the same requestor are satisfied. Only if the command satisfies the timing, the command is allowed to be fetched from the particular command queue and will be propagated to stage 3. If not, the command will be staying in the command queue until its DDR timing is satisfied. The number of command queues depends on how many requestors are connected with the memory controller design. For a system with sixteen requestors, there will be sixteen command queues to store their respective commands. The design was implemented in such a way that number of command queues can be dynamically configured during the run time depending on the number of requestor that were chosen by the user.

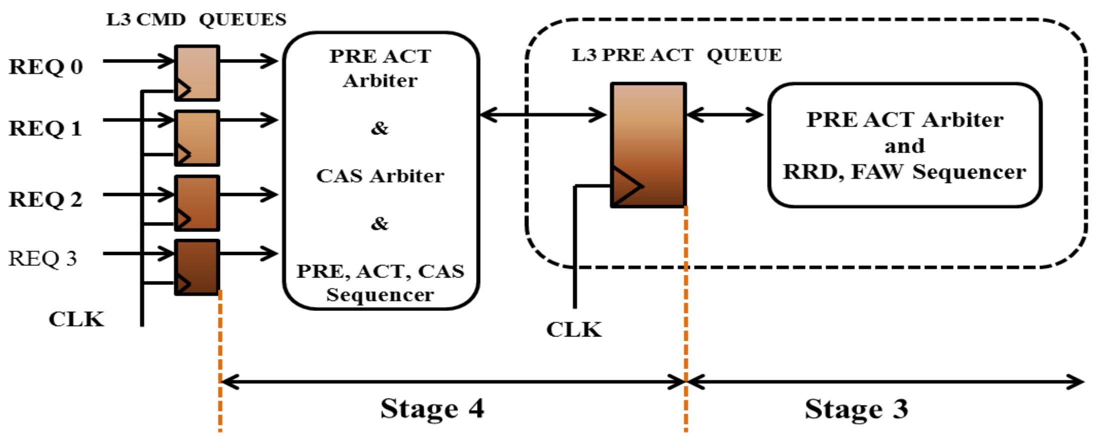


Figure 5.5: L3 CMD Queue, L3 PRE, ACT Queue and RRD FAW Sequencer

5.2.2 PRE, ACT Arbiter and CAS Arbiter in Stage 4

The stage 4 consists of two arbiters namely PRE ACT Arbiter and CAS arbiter as shown in Figure 5.5. Both arbiters work in parallel to arbitrate among commands waiting at the head of L3 CMD queues. The Arbitration rule (2A) says the requestor is put in the back of a L3 PRE ACT Queue in stage 4 as soon as it has an active PRE or ACT command and it is removed from the queue once the command is finally issued by L1. But, all the commands in CMD queues cannot be dispatched all once instantly as stated in rule (2A). From the hardware implementation point of view, each command in the L3 CMD queue can only be dispatched one per clock cycle. Therefore, we need an additional arbitration mechanism to choose the PRE, ACT command per clock cycle from L3 CMD queues. Similarly, we also need an arbitration mechanism to choose one CAS command per clock cycle from CMD queues. The CAS arbiter arbitrates a L3 CMD queue that has CAS commands at the front (head) of the queues. As per the arbitration rule (1B), a CAS command does not become active until the data of the previous CAS command of the same requestor has been received by the memory controller from the memory device. While CAS arbiter is in action, PRE ACT arbiter arbitrates a queue that has PRE or ACT commands at the front (head) of the queue. Both arbiters ignore those queues that do not have any command to be served.

Having two separate arbiters to arbitrate at the same time plays a major role in reducing overall latency in our proposed rank switching DDR memory controller design. This L3 PRE ACT Arbiter is designed based on the modified FCFS arbitration style. This modified FCFS ensures that when the requestor has an ACT waiting for its t_{RRD} or t_{FAW} constraints, the PRE commands do not have to suffer due to t_{RRD} or t_{FAW} constraints. This allows the late arriving PRE command to get propagate. On the other hand, the CAS arbiter is built from the regular First Come First Served (FCFS) arbitration style. Having two separate arbiters for PRE ACT and CAS really help to reduce the overall latency. Let us analyze a scenario with an example where REQ 1, REQ2, REQ3, REQ4 queues receive the commands in the order of ACT, PRE, ACT, and CAS respectively. If one Arbiter was used to handle all the commands, then, the arbiter would start arbitrating from REQ1 => REQ2 => REQ3 => REQ4. The CAS command that is waiting at REQ4 has to wait until the arbiter finished arbitrating REQ1, REQ2 and REQ3. This is not efficient and there is no reason to keep waiting the important CAS command at REQ4. Instead, this CAS command should be dispatched as early as possible to save the latency in getting the data back from memory. The proposed design has one arbiter to handle PRE, ACT and other arbiter to handle CAS in order to reduce the waiting time of the important CAS commands inside the Command queues and thereby minimizes the total latency.

5.2.3 PRE, ACT, CAS Sequencer in Stage 4

When the command is available at the head of the L3 CMD queue, the PRE, ACT, CAS sequencer in stage 4 will start verifying the timing check and make sure that the current command from the requestor satisfies the timing constraint with the previous command from the same requestor as shown in Figure 5.5. The look-ahead nature of L3 CMD queue allows this sequencer to scan the commands at the head of the command queues and verify the timing without actually fetching it from the command queue. Note that both PRE ACT Arbiter and CAS arbiter carry out their arbitration task while checking with this sequencer to determine if the particular command can be chosen for the arbitration. Each CMD queue represents the commands that are solely coming from a single requestor. This sequencer checks the timing constraint such as RAS to CAS (RCD), CAS Latency (CL), RC, RAS, RP for the commands from the very same requestor. It is important to note that the timing constraint such as ACT to ACT (RRD) or Four Active Windows (FAW) for the commands that come from different requestors targeting different banks are not verified by this sequencer.

As shown in Figure 5.5, PRE ACT arbiter, CAS arbiter works in parallel with PRE ACT CAS sequencer to check if the particular command satisfied the timing before the arbiters can choose the command for its arbitration process. If the command did not satisfy with the timing, then, the arbiters would not choose that command and instead, it would move onto the next requestor in its arbitration path. The Stage 4 PRE, ACT, CAS Sequencer was designed with large set of down counters to represent all the DRAM timing constraints as shown in Table 2.1. These timing counters in this particular sequencer in stage 4 would only check the timing constraint for the commands within each requestor. As indicated in Figure 5.4, the feedback shown in green color is a combinational path coming from level 1 towards level 2 and level 3. This feedback indicates that the command is dispatched at level 1 and this acknowledgement is used by the timing sequencers in level 3 and level 2 to initialize the corresponding timing counters for the next commands. It is important to note that the timing constraint such as Row to Row (RRD) or Four Active Windows (FAW) for the commands that come from different requestors, targeting different banks in the same rank are not verified by this timing sequencer in stage 4 and it will be discussed in section 5.2.5.

5.2.4 PRE, ACT Queue in Stage 4

Once the DRAM timing constraint has been satisfied by the PRE, ACT, CAS sequencer, it would propagate the PRE and ACT commands to PRE ACT queue for a temporary storage as shown in Figure 5.5. This PRE, ACT queue receives commands from requestors targeting different banks within the same rank. This queue is a custom made design and it is not based on either regular First in First out or Last in First out architectures. In regular queue architecture, the data will be fetched on the next cycle after receiving the read enable from the receiving block. But, this PRE ACT queue was designed in such a way where the next available command is automatically just visible without waiting for the read enable from the receiving block. This look-ahead feature allows the receiving block, RRD FAW sequencer, to validate corresponding RRD, FAW timing check even before actually fetching it. Further, this queue was designed to offer higher priority to PRE command over ACT command when ACT was subjected to additional delay due to the RRD, FAW timing constraint. This PRE ACT queue receives ACT commands that are coming from all four requestors targeting different banks in the same rank. Therefore, those ACT commands targeting different banks in the same rank are expected to satisfy the timing parameters such as Row to Row Delay (RRD) and Four Active Window (FAW).

The PRE ACT sequencer evaluates the timing of ACT command and decides when the ACT command can be fetched from the PRE, ACT queue based on its RRD, FAW sequencer timing. While the ACT is waiting for its corresponding RRD, FAW timer to elapsed, there is no reason to hold those PRE command inside the queue. In this scenario, PRE are given higher priority than ACT command. To facilitate this priority, the PRE ACT queue was designed to make necessary up shifting of PRE commands so that PRE commands can be released early as shown in the following Figure 5.8. This early release makes a reasonable improvement over the overall latency of the operation of our proposed design. The next section describes in detail how this RRD FAW sequencer works in parallel with PRE ACT queue.

5.2.5 PRE ACT Arbiter and RRD FAW Sequencer in Stage 3

The section describes the task of the PRE, ACT arbiter and RRD, FAW sequencer as shown in stage 3 of Figure 5.5. This logic unit consist of PRE, ACT arbiter and RRD, FAW sequencer. The arbiter logic arbitrates commands that are in PRE, ACT queue. If the front (head) of the queue is an ACT command, the arbiter would choose the ACT commands only if the Row to Row Delay (RRD) and Four Active Window (FAW) delay is satisfied for that ACT command by the RRD, FAW Sequencer logic. The arbitration rule (2C) says that an active PRE command can always be issued; an active ACT command could instead by blocked by t_{RRD} or t_{FAW} constraints caused by other requestors in the same rank. Now, let us analyse how this rule (2 C) is implemented. The RRD and FAW timings are verified by RRD, FAW Sequencer logic for an ACT command. Once first ACT command is inserted into the L3 PRE ACT queue, the Row to Row Delay (RRD) counter and Four Active Window (FAW) counter will be initialized to RRD value and FAW value respectively. When it receives the second ACT command, it would check if the RRD counter has been already elapsed or not. If it is not elapsed, this logic block will wait and will not send the read enable to the PRE ACT Queue until the RRD timing is satisfied. Only after RRD timer is elapsed, it would allow this second ACT command from PRE ACT Queue to propagate to next stage in the pipeline. At the same time, it would send back the read enable to the PRE ACT Queue so that queue would pop up the next available command to its head. This process would keep on continuing every time there is a new ACT command. The FAW counter will allow only four ACT commands to pass within the FAW time window.

5.2.6 PRE, ACT Arbiter in Level 2

As shown in the full system level view of Figure 5.4, four PRE ACT queues are in L3. This L2 PRE ACT Arbiter was designed based on the round robin arbitration architecture as explained in Section 3.1.4 Selection of arbiter type. When a L3 PRE ACT queue has the command at the head of the queue waiting to be sent, first, it would send the request (REQ) to this Level 2 PRE ACT Arbiter. In a full system level operation, the L2 PRE ACT Arbiter receives REQs from all four PRE ACT queues residing in level 3. The arbitration rule (2 A) says that the PRE ACT command is removed once the PRE ACT command is issued by L1. This sub-rule is implemented as follows. The L2 PRE ACT arbiter will pick one of the L3 PRE ACT queue representing different ranks.

L2 PRE ACT arbiter will issue the acknowledgment (ACK) to that chosen L3 PRE ACT queue. Only after receiving the ACK, the corresponding L3 PRE ACT queue will release the PRE or ACT command to this L2 PRE ACT Arbiter. Note that REQ and ACK exchange between L3 to L2 are combinational logic and it does not consume any clock cycle. The arbitration rule (4) says that this level 2 PRE ACT Arbiter can use either FCFS or Round-Robin (RR) arbitration. But, this level 2 PRE ACT was implemented using RR due to the ease of implementation in hardware compared to FCFS. The command received by this level 2 PRE, ACT Arbiter would then dispatch it to the next pipelined storage which reside between level 2 PRE ACT Arbiter and level 1 Arbiter.

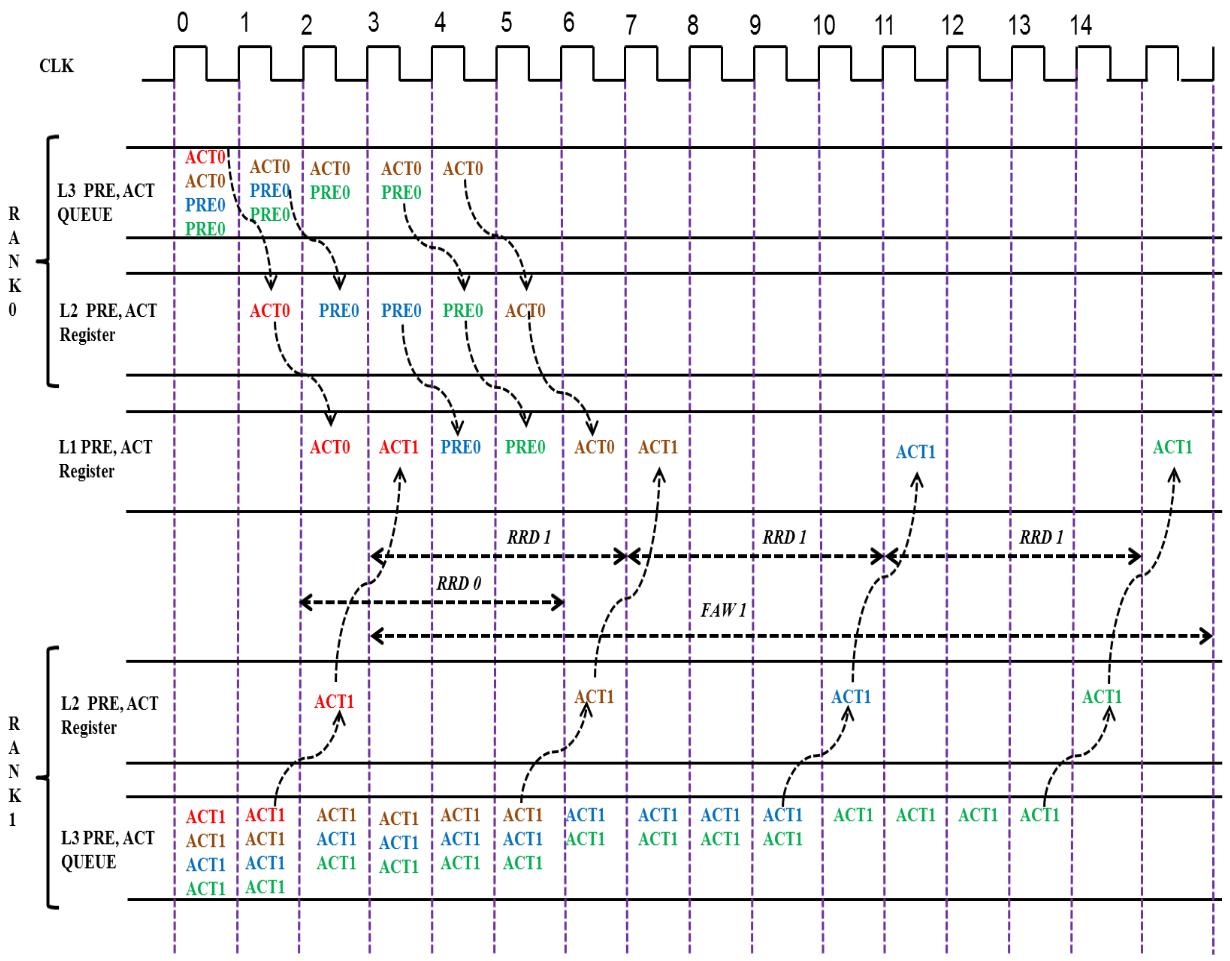


Figure 5.6: Scheduling of PRE, ACT CMDs through L3, L2, L1

Let us look into the detail of scheduling events of PRE, ACT commands from levels of L3 to L2 to L1 as shown in Figure 5.6. At clock cycle 0, L3 PRE, ACT queue of rank 0 contains ACT0, ACT0, PRE0, PRE0 commands. Similarly, at clock 0, L3 PRE, ACT queue of rank 1 contains all four ACT1 commands. At clock cycle 0, assume that rank 0 takes higher priority than rank 1. Even though both rank 0 and rank 1 send the REQs to L2 PRE ACT arbiter, L2 PRE ACT arbiter choose the L3 PRE ACT queue from rank 0 due to the higher priority. Therefore, PRE ACT queue of rank 0 will own the bus ownership and will dispatch the ACT0 command into L2 PRE, ACT at clock cycle 1. Next, the L2 PRE ACT will dispatch the ACT0 from L2 to L1 at clock cycle 2. Due to the Row to Row Delay (RRD) between ACT commands of the same rank, the earliest time that L1 PRE ACT can receive the next ACT0 command would be at clock cycle 6. Therefore, earliest time the next ACT0 command from L3 PRE, ACT queue of rank 0 could be dispatched is at clock cycle 4.

The moment that ACT0 is dispatched by L1 PRE ACT register at clock cycle 2, the RRD counter is initialized to a RRD value. The design of this down counter take into the fact that RRD counter is actually getting its initialized value on the next clock cycle so that initialized value should be RRD – 2 so that the counter would be down countered to zero at clock cycle 5 where next ACT0 command will be checked for its RRD constraint to be elapsed at the correct time so that next ACT0 command is dispatched to L1 PRE ACT at clock cycle 6. Please note that the counter initial value modification is done carefully in our design for all the counters involved in checking the DDR timing constraints.

On other hand, the L3 PRE, ACT queue of rank 1 will dispatch its first ACT1 command from L3 PRE ACT queue into L2 PRE ACT register at clock cycle 2, only after the ACK is given by the L2 PRE ACT arbiter. Note that rank 1 is less priority than rank 0 as per our assumption. Next, L2 PRE ACT register will dispatch the ACT 1 command into L1 at clock cycle 3 after ACT0 from rank 0 is dispatched. Note that ACT commands from different ranks does not have any timing constraints between them and can be processed back to back at clock cycle 2 and clock cycle 3 as shown in Figure 5.6. The PRE command has no impact on either RRD or FAW timing. Therefore, if there is a PRE command in the queue, there is no need for the PRE command to wait unnecessarily while waiting for either RRD or FAW. The Figure 5.6 clearly illustrate scenario of early dispatching of PRE0 at clock cycle 2 while ACT0 command is waiting for RRD or FAW at L3 PRE ACT queue for rank 0.

5.2.7 CAS FIFO in Stage 4

Once the PRE, ACT, CAS Arbiter & Sequencer unit chooses a CAS command, it propagates that CAS command to the temporary storage of CAS FIFO as shown in Figure 5.7 below. The CAS FIFO has a variable depth size depending on the number of requestors connected. When either 4 or 8 requestors are connected in the above logic unit, the CAS FIFO size would take either 4 or 8 respectively. This CAS FIFO would not release (pop) the CAS Command to the next stage downstream until the required DRAM timing constraint is successfully checked by the CAS BTB Sequencer which will be described in next section. Only when the CAS FIFO is chosen as the winner by level 2 CAS Arbiter the CAS FIFO would release the CAS command and send it to level 2 CAS Arbiter as shown in Figure 5.10. If the CAS FIFO is not chosen as the winner, then, it would not release the CAS Command from the CAS FIFO. The winner signal coming from level 2 CAS Arbiter is used as the read acknowledgement for this CAS FIFO logic to get the next command to pop up at the head of the CAS FIFO for the next operation.

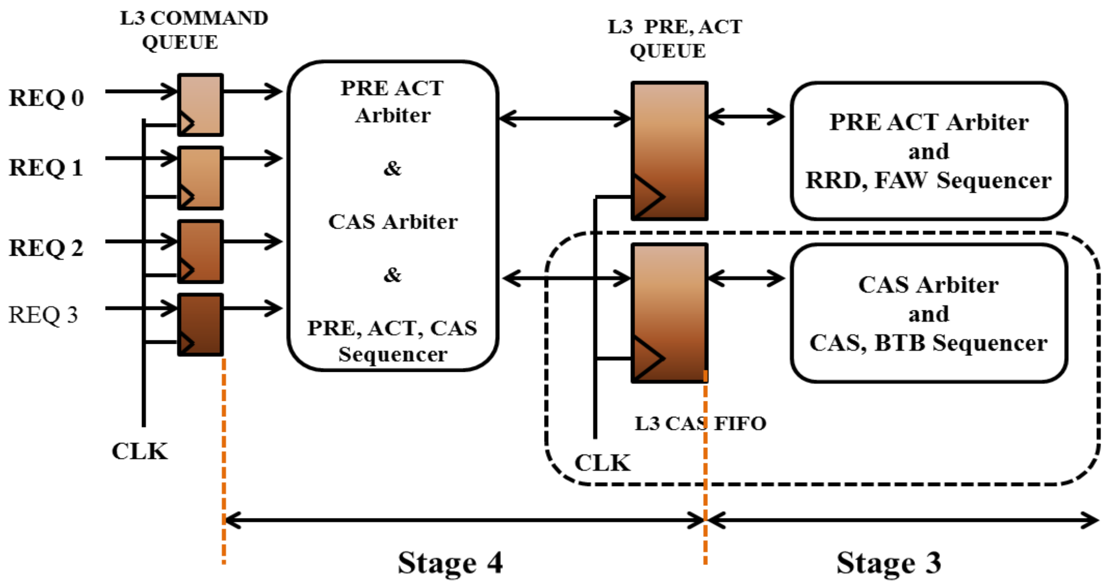

L3 COMMAND QUEUE
REQ 0
REQ 1
REQ 2
REQ 3
CLK
PRE ACT Arbiter & CAS Arbiter & PRE, ACT, CAS Sequencer
L3 PRE, ACT QUEUE
PRE ACT Arbiter and RRD, FAW Sequencer
CAS Arbiter and CAS, BTB Sequencer
L3 CAS FIFO
CLK
Stage 4
Stage 3

Figure 5.7: CAS FIFO and CAS BTB Sequencer

5.2.8 CAS Arbiter and CAS, BTB Sequencer in Stage 3

This logic block is directly interacting with CAS FIFO as shown in Figure 5.7 above. Three logic operations such as CAS arbitration, CAS timing and BTB timing are built in this logic block. As per arbitration rule (3), the commands at the head of the L3 CAS FIFO are arbitrated by the CAS Arbiter unit. This arbiter selection process depends on the timing check done by CAS sequencer and BTB sequencer logic blocks. The timing constraint such as Write to Read (WTR) and Read to Write (RTW) between banks will be verified by this CAS sequencer.

Note that the arbitration rule (5) only discusses the $\mathbf{t}_{SDr}$ and $\mathbf{t}_{ED}$ and it does not discuss about the BTB timing parameter. It is important to understand how this BTB is important for the processing of CAS commands. The BTB stands for Burst to Burst and the BTB sequencer calculates the time difference between two burst data of the different requestors targeting either same or different banks in the same rank. As per the arbitration rule (5), the BTB = $\mathbf{t}_{SDr}$ - $\mathbf{t}_{ED.}$ The calculated BTB value for each L3 CAS will be dispatched to the level 2 CAS Arbiter and this process is repeated by other CAS BTB sequencers in other ranks as shown in Figure 5.8. At the L2 CAS Arbiter, it receives CAS commands and the corresponding BTB values from CAS, BTB Sequencers representing all four ranks or two ranks depending on the configuration. In the next section, we will see how level 2 CAS Arbiter chooses one of the CAS FIFOs based on both the BTB value received and the priority of the CAS FIFOs.

5.2.9 CAS Arbiter in Level 2

The L2 CAS Arbiter consists of three important logic units and they are CAS Arbiter, BTB Comparator and Rank to Rank (RTR) Sequencer. The L2 CAS arbiter arbitrates L3 CAS FIFOs representing different ranks as shown in Figure 8 below. The arbitration task carried out by L2 CAS arbiter not only just depends on the arbitration policy itself, but also depends on the outcome of BTB Comparator and Rank to Rank (RTR) sequencer. The factors that decides on how L2 CAS arbiter is supposed to choose the L3 CAS FIFOs is as follows. If L2 CAS arbiter makes its selection as per round robin policy, each L3 CAS FIFO will be visited in an equal fairness and orderly manner regardless of which requests are waiting for a long time in the L3 CAS FIFOs. But, our design of L2 CAS arbiter gives importance to those requests which arrived early waiting at the L3 CAS FIFO to be served.

It also gives equal importance to other requests which are waiting for their read to write or write to read or burst to burst timing constraint to be elapsed. The important fact is that we cannot use either fixed priority or round robin policies for the L2 CAS arbiter without considering the request's arrival time. Therefore, we decided to choose First Come First Served structure. Since this arbitration process is also depend on Burst to Burst (BTB) values, the existing FCFS policy need to be modified and thereby, Modified FCFS (M-FCFS) is well suited for our design as the arbitration policy for L2 CAS arbiter. Now, let us look into the BTB Comparator logic which is part of the L2 CAS arbiter logic.

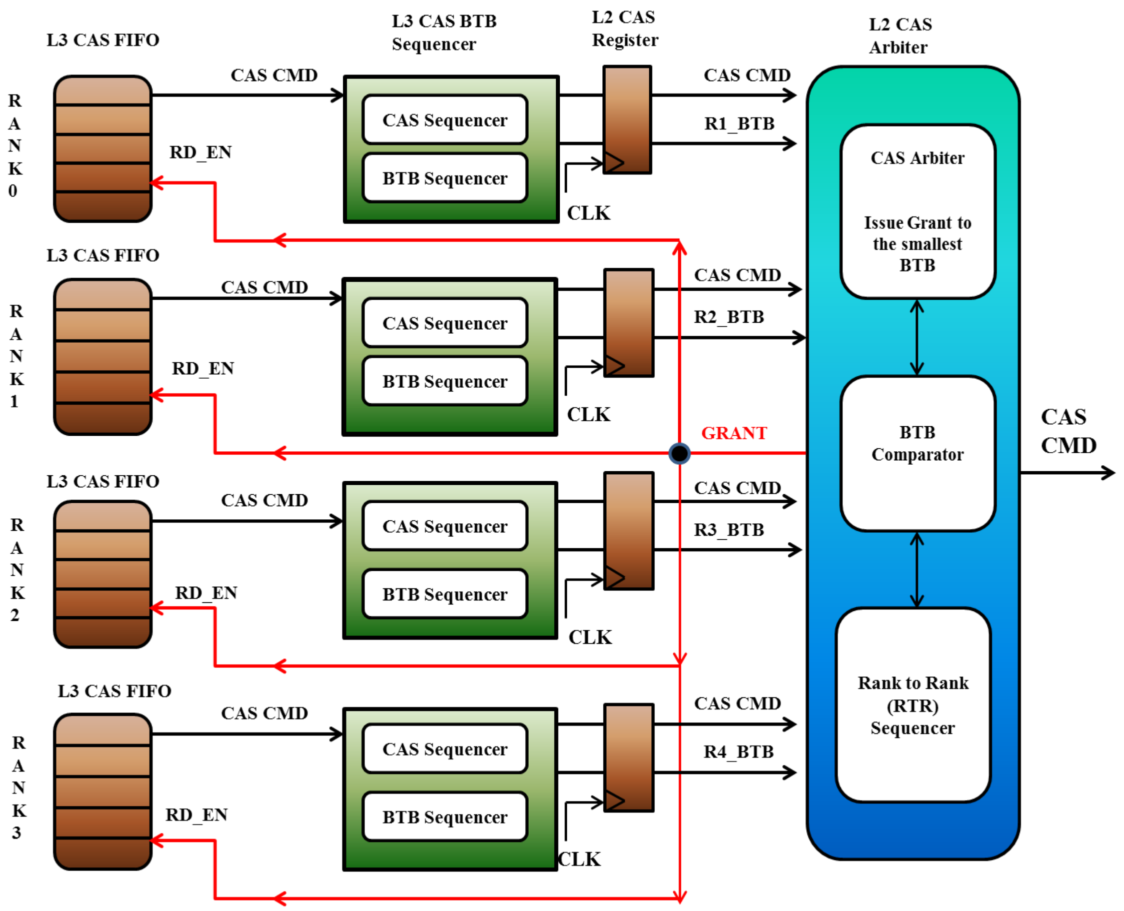


Figure 5.8: Level 3 CAS BTB Sequencer and Level 2 CAS Arbiter

5.2.9.1 BTB Comparator

Whenever there are CAS commands available at L3 CAS FIFOs, the L2 CAS Arbiter receives the Burst to Burst (BTB) value from all BTB Sequencers representing all the ranks. Note that BTB value is issued by the BTB sequencer to L2 CAS arbiter even if the CAS cannot be issued at any clock cycle. In other words, the BTB is issued at every clock cycle and the BTB comparator compares the BTB values coming from all the BTB sequencers as shown in Figure 5.9. The winner is chosen based on BTB values and based on the priority level of each of the L3 CAS FIFOs at that time. The winning L3 CAS FIFO is allowed to release the CAS command to the L2 CAS Arbiter. Other L3 CAS FIFOs which were not chosen would keep their CAS Commands. Once the CAS command is chosen from a particular L3 CAS FIFO, it still need to go through one more timing check called Rank to Rank (RTR). The RTR Sequencer unit will check the timing before the CAS Command can actually be indeed released by the L3 CAS FIFO.

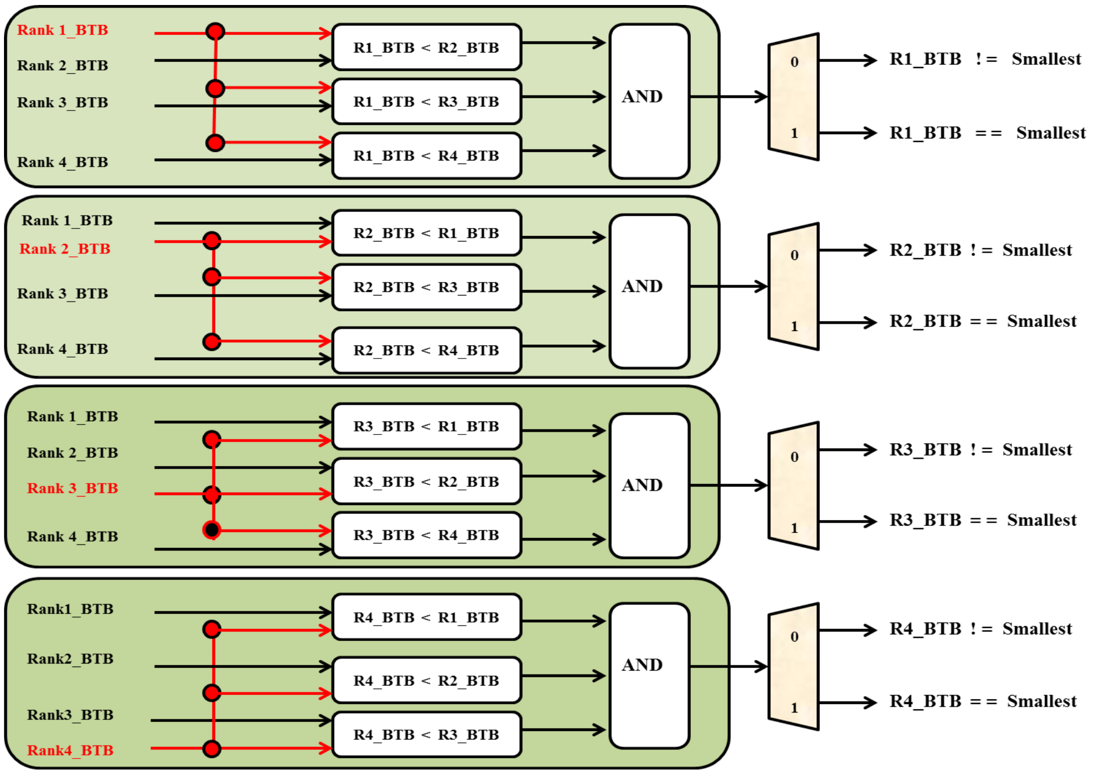


Figure 5.9: Logic to Calculate the Smallest BTB

5.2.9.2 Rank to Rank (RTR) Sequencer

Having completed the BTB comparison, the L2 CAS Arbiter also needs to do another timing check called Rank to Rank. It is important to analyze why we need this rank to rank sequencer between ranks. The synchronization time is needed for one bus master to hand off the bus ownership to another bus master. This time is called turnaround time which is inserted to account for skew on the bus and to prevent different bus masters from driving the bus at the same time. To avoid such collisions, a second rank must wait at least t_{RTR} after a first rank has finished using the bus. This synchronization time is called Rank to Rank Time, RTR.

The RTR Sequencer was designed to ensure that the second rank would not drive the data bus while the first rank is in the process of driving the data bus and avoiding collision between data movement among ranks. Within each rank itself, the timing conflict in the directions of data movements is called write to read (WTR) and read to write (RTW). But, when the data movements occur between ranks, the timing conflicts such as WTR or RTW due to the direction of data movement are no longer applicable. Instead, only Rank to Rank timing check need to be verified by this level 2 CAS Arbiter before dispatching the received CAS command towards level 1.

It is important to further analyse how the CAS commands are scheduled and propagated from L3 to L2 to L1 levels. The Figure 5.10 shows an example scenario to illustrate the scheduling of CAS command along with their corresponding BTB values.

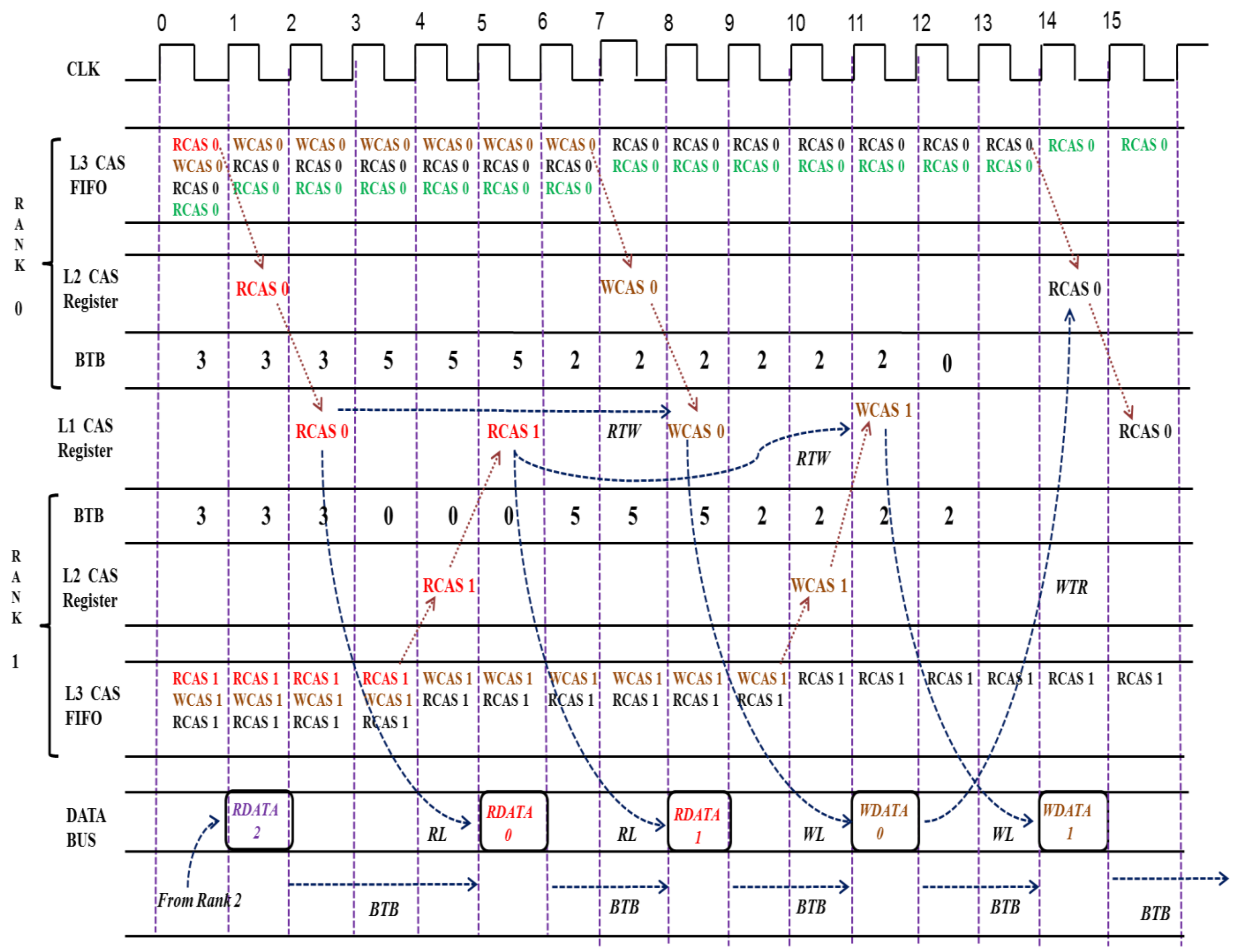


Figure 5.10: Scheduling of CAS CMDs through L3, L2, L1

Let us assume the following for our example scenario. The burst size of 1 clock cycle and read, write latency of 3 clock cycles are used as a scale down view for the illustrative purpose in this Figure 5.10. Further, assume that four CAS commands are stored in L3 CAS FIFO of rank 0 and three CAS commands are stored in L3 CAS FIFO for rank 1. Also, assume that RDATA2 is present at clock cycle 1 by rank 2 to illustrate the BTB processing for rank 0 and rank 1. Assume that the rank 0 has higher priority than rank 1.

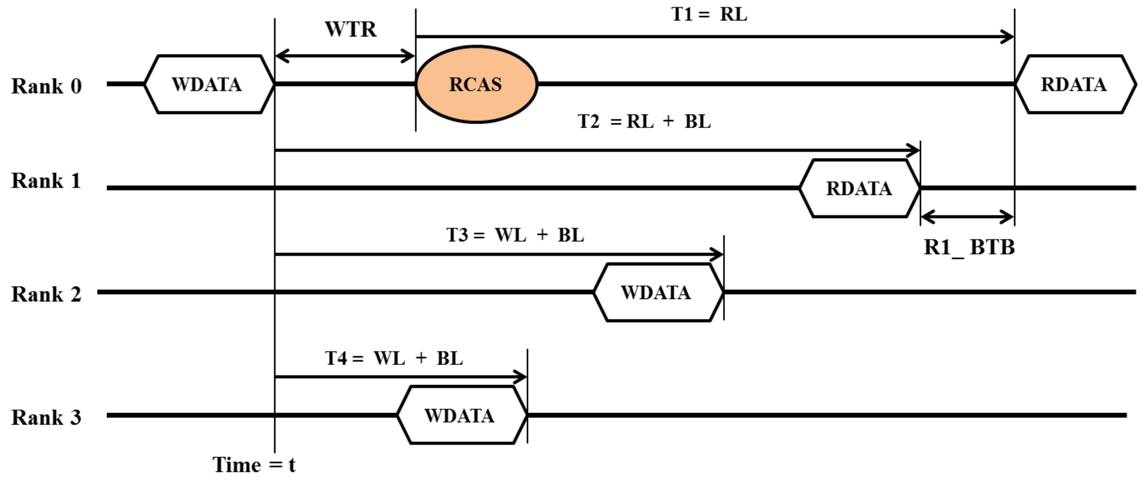
At clock cycle 0, due to the RDATA2, the BTB values for Rank 0 and Rank 1 are calculated to be 3. Since both BTB values are equal to be 3 at clock cycle 0 and since rank 0 has higher priority than rank 1, the L3 CAS FIFO (rank 0) is chosen as the winner by the L2 CAS arbiter. Therefore, RCAS0 command in L3 CAS FIFO of rank 0 is dispatched to L2 CAS register at clock cycle 1. At clock cycle 2, this RCAS0 will be further dispatched to L1 CAS for output. The read data, RDATA0 is received at clock cycle 5 after RL timing.

While this RCAS0 is propagating through L3 , L2 and L1, the RCAS 1 waiting at L3 CAS FIFO (rank 1) cannot be dispatched to L2 CAS register until clock cycle 4 due to the fact that Burst to Burst (BTB) need to be maintained between RDATA0 and RDATA1 as shown in Figure 5.10. The BTB values for both rank 0 and rank 1 are maintained to be same value as 3 for clock cycle 0, 1, 2 due to the presence of RDATA 2 from rank 2. Once the RCAS0 is dispatched at clock cycle 2, the next WCAS 0 at rank 0 is waiting until clock cycle 8 to satisfy the read to write (RTW) timing constraint. Due to this WCAS0, the new BTB value for rank 0 is updated at clock cycle 3 to be 5 as per the following equation.

$$\begin{aligned} BTB &= t_{RTW} + t_{WL} - t_{RL} - t_{Burst} \\ &= 6 + 3 - 3 - 1 \\ BTB &= 5 \end{aligned}$$

Similarly, at clock cycle 3, BTB for rank 1 is calculated to be 0, since there is no data in the bus at that time for rank 1.

The following Figure 5.11 shows detail view of how the BTB values are calculated. For an example, the RCAS from rank 0 can be dispatched only if the R1_BTB value can be satisfied between the latest RDATA from rank 1 to the future RDATA from rank 0. The T0_MAX is calculated as the maximum delay of the data from other ranks. If the RCAS command at rank 0 is followed by previous write data, then, we need to write to read (WTR) delay as part of the BTB calculations. All the relevant equations for the BTB calculations are shown in Figure 5.11.



T0_MAX = MAX { T2, T3, T4 }

R1_ BTB = WTR + RL – T0_MAX

RCAS CMD from Rank 0 can be sent when (T0_MAX + R1_BTB – T1) == 0

Figure 5.11: Example BTB calculation for RCAS CMD from Rank0

5.2.10 PRE ACT CAS Arbiter in Level 1

As can been seen from the system level view in Figure 5.4, the commands from all the requestors from different ranks arrives simultaneously at this final stage called level 1 PRE ACT CAS arbiter. This L1 arbiter arbitrates for the PRE, ACT and CAS commands coming from all ranks. This arbiter was designed as a priority style arbiter since the proposed design expects the CAS commands to be given higher priority than PRE, ACT commands. By issuing the CAS commands as early as possible offer better read latency and hence reduces the overall latency. Note that there is only one command bus and therefore, only a single command can be dispatched out of level 1 towards the memory device. Since CAS command get the highest priority, this arbiter is built with the ability to temporarily store those PRE and ACT which may arrive at the same time as CAS commands from different requestors of different ranks. Only after CAS is dispatched out of level 1 arbiter, those stored PRE, ACT commands will be dispatched as per the order they arrived. As stated in arbitration rule (6), this level 1 arbiter would send back the ACK to both level 2 and level 3 every time the commands have been dispatched out of this level 1 arbiter.

5.3 Pipeline Implementation of the Memory Controller

This section will analyse the pipeline implementation of the back end memory controller. As shown in Figure 5.12, the back end is designed to be three stage pipeline structures. This section would go into detail on each stages and the timing analysis as pointed out below.

1) Pipeline Stage 4 – Request Arbitration
2) Pipeline Stage 3 – Bank Arbitration
3) Pipeline Stage 2 – Rank Arbitration
4) Pipeline Stage 1 – Command Arbitration
5) Timing Analysis of Pipeline Stages

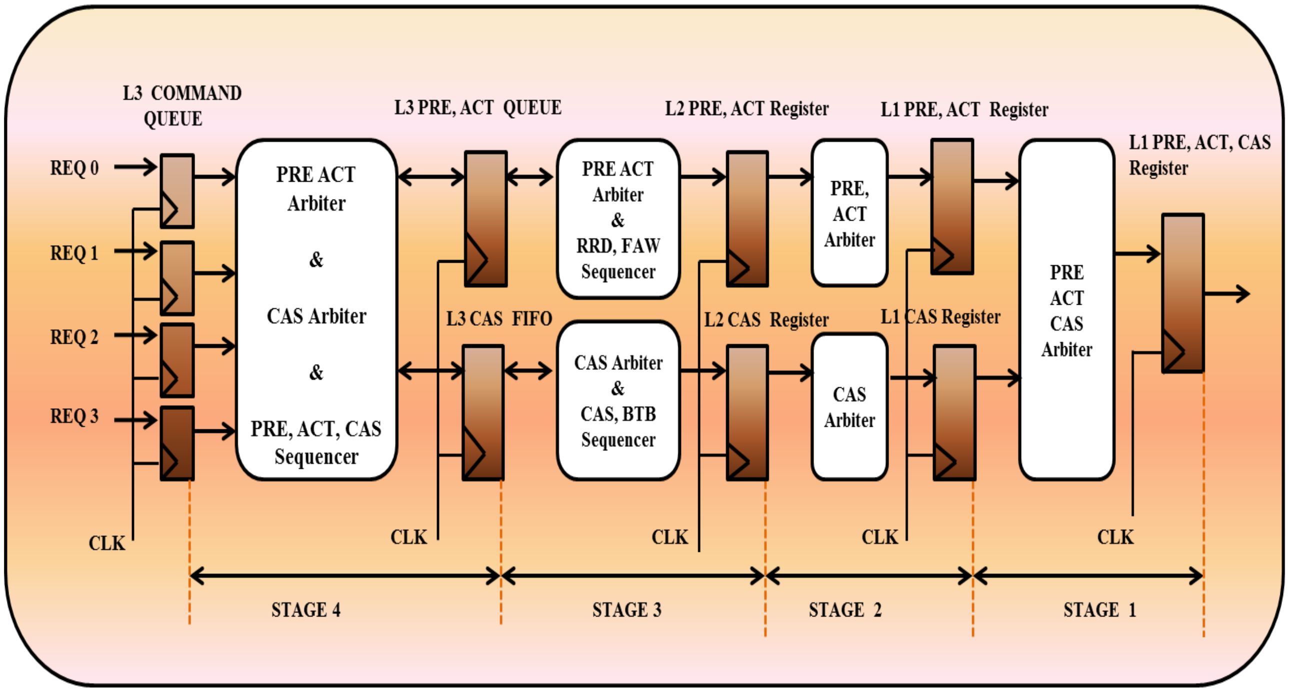


Figure 5.12: Three Stage Pipeline for the backend Memory Controller

5.3.1 Pipeline Stage 4 – Request Arbitration

This pipeline stage 4 receives the commands from a sequential unit called CMD queues. This stage contains logic units such as PRE, ACT Arbiter, CAS Arbiter and PRE, ACT, CAS Sequencer. It is important to note that this sequencer performs the timing checks on the commands coming from same requestor, not between requestors. Once the arbitration and timing checks are completed, the PRE, ACT commands are stored into L3 PRE ACT queue and the CAS commands are stored into L3 CAS FIFO as shown in Figure 5.13. The commands stored into this L3 PRE ACT queue and L3 CAS FIFO represents four requestors of the same rank.

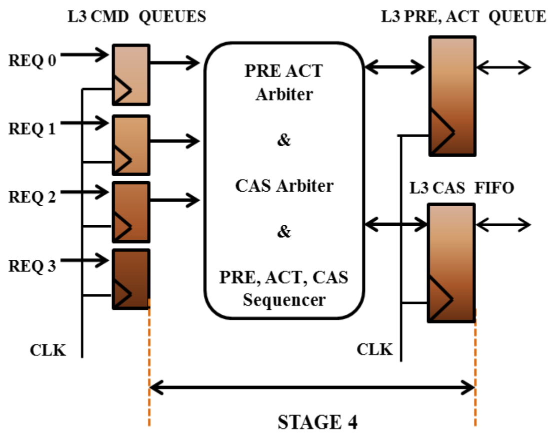


Figure 5.13: Stage-4 Pipeline

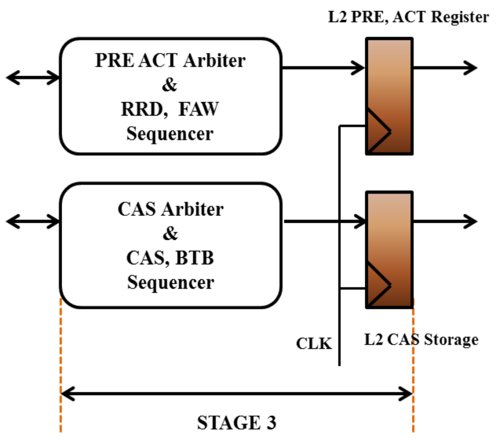

L2 PRE, ACT Register
PRE ACT Arbiter & RRD, FAW Sequencer
CAS Arbiter & CAS, BTB Sequencer
CLK
L2 CAS Storage
STAGE 3

Figure 5.14: Stage-3 Pipeline

5.3.2 Pipeline Stage 3 – Bank Arbitration

The pipeline stage 3 contains the logic block called PRE, ACT arbiter and RRD, FAW sequencer which receives the commands from L3 PRE, ACT queue of stage 4. Similarly, the stage 3 also contains CAS arbiter and CAS, BTB sequencer which receives the command from L3 CAS FIFO of stage 4. Having completed the arbitration and timing check tasks, the PRE, ACT commands will be dispatched into the L2 PRE, ACT register and CAS commands will be dispatched into L2 CAS register as shown in Figure 5.14.

5.3.3 Pipeline Stage 2 – Rank Arbitration

Stage 2 has two arbiters: PRE, ACT Arbiter and CAS Arbiter. First PRE ACT Arbiter performs the rank arbitration for PRE ACT commands that are stored into the L3 PRE ACT Queues. At the same time, the CAS Arbiter performs the arbitration on CAS Commands that are stored L3 CAS FIFOs. As you can see in Figure 5.15, there are registers between the stage 3 and stage 4. These registers are used to carry out the pipelined nature of the design. It is important to note that these registers are not buffer to store the commands for a longer period; rather these registers are used to delay the commands by just one clock cycle in order to perform the pipeline nature for our design. Actually, the L3 PRE ACT queue and L3 CAS FIFO are the storage place where the commands are getting buffered until they receive the ACK from L2 arbiters and also they wait in the buffer until their timing constraint is satisfied.

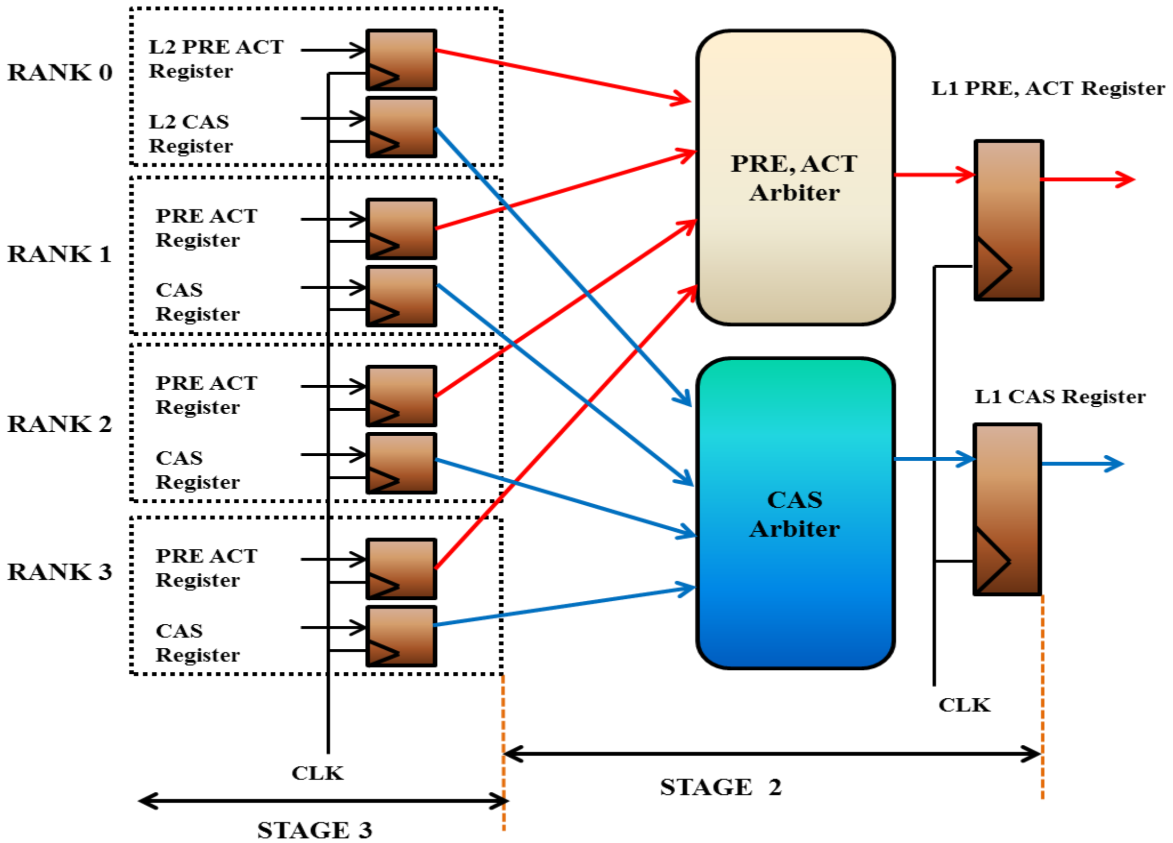


Figure 5.15: Stage 2 Pipeline – Rank Arbitration

5.3.4 Pipeline Stage 1 – Command Arbitration

As shown in system level Figure 5.4, the stage 1 receives commands from all the requestors representing different ranks. This is the final stage where all commands come together from all the requestors of all the ranks. The stage 1 contains PRE, ACT, CAS Arbiter of priority type. At this level, the command arbitration is performed where CAS command is given higher priority than PRE or ACT if all 3 commands arrive at the same time. Since stage 1 arbiter is the final stage of our memory controller design, the output coming out of this stage 1 should be clocked in order to send the sequential signals into the memory device. This will prevent any glitches coming from the stage 1 combinational logic being passed onto the memory device. The Stage 1 PRE, ACT CAS arbiter also send out the feedback signal to Level 3 and Level 2, once the commands are dispatched out of this arbiter towards the memory device. This feedback is used by the level 3 and level 2 timing sequencers to evaluate the timing constraints for the next commands.

Up to now, all 3 stages of pipeline design was presented. The full implementation of the front and back end design was done using Verilog RTL language and the code can be found at [18].

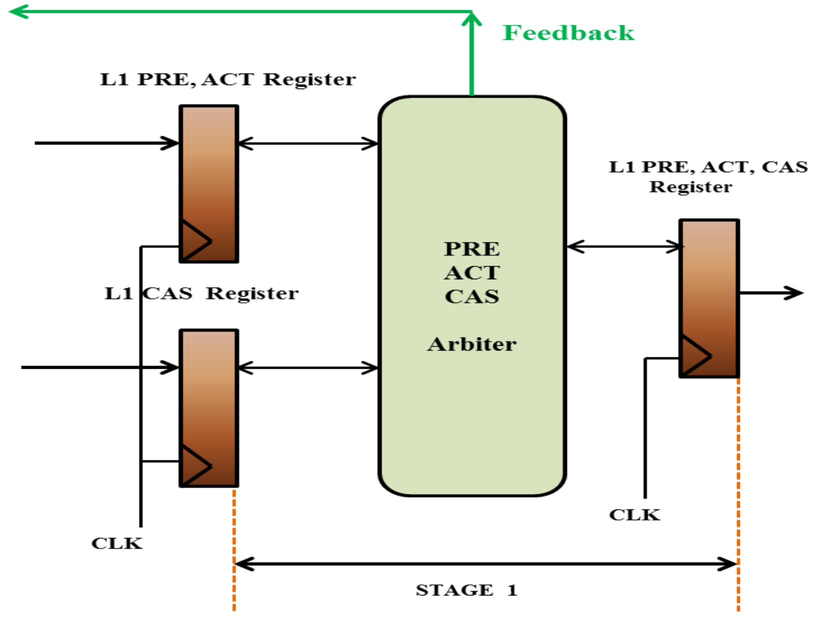


Figure 5.16: Stage 1 Pipeline – Command Arbitration

5.3.5 Timing Analysis of Pipeline stages

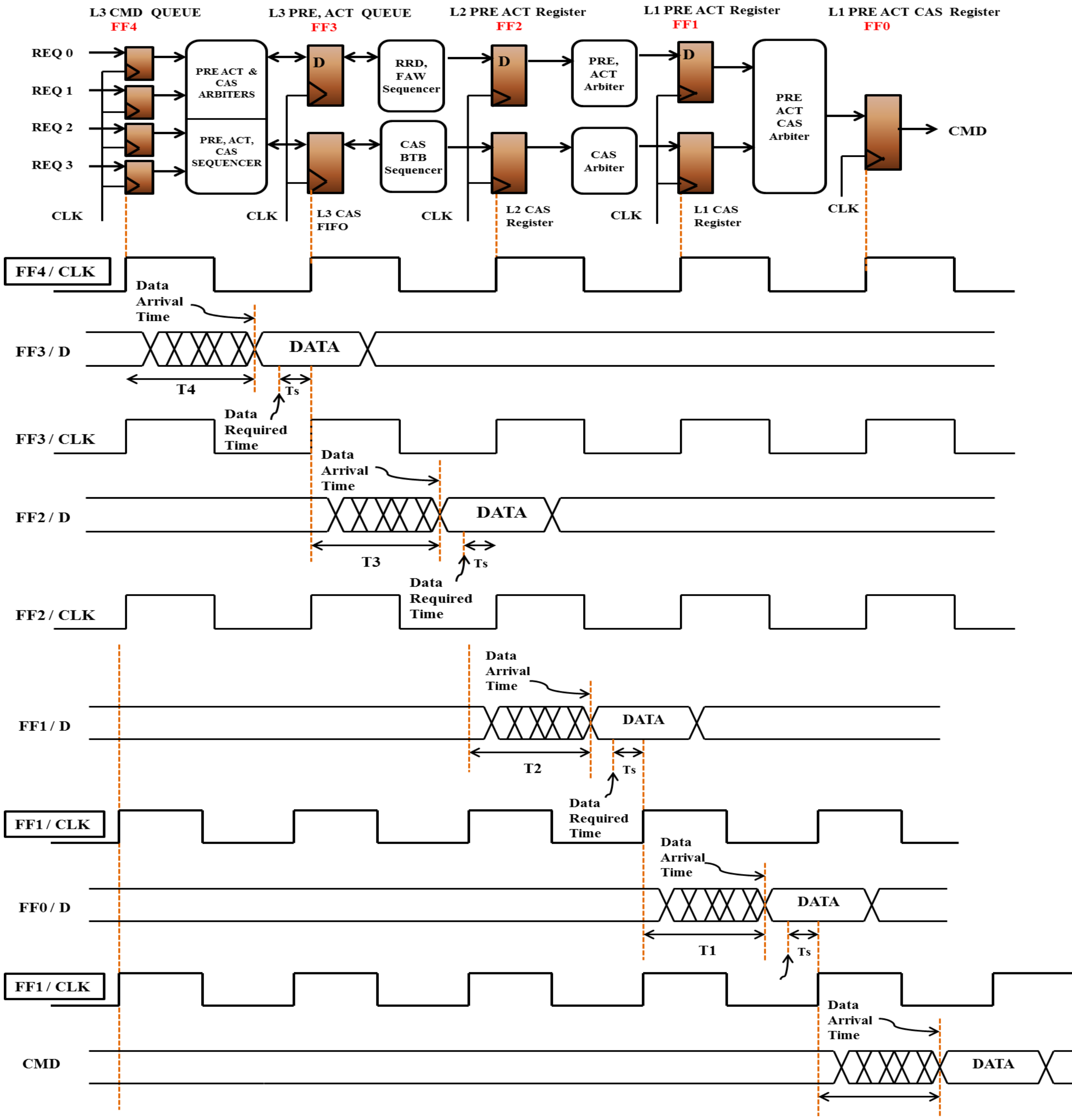


Figure 5.17: Timing Analysis of Pipeline Stages

The detailed timing of each pipeline stages is shown in Figure 5.17. The data arrival time indicates the time at which the data arrived after being subjected to the combinational delay by the logic in each stage. The data required time indicates the safest time at which the data is expected to reach to avoid the setup time, Ts. The Time duration of T4, T3, T2, T1 indicates the Data Arrival time and it includes the delay suffered by logic for the corresponding pipeline stages. In order to analyse the time taken by each stage of the pipeline, the Static Timing Analysis (STA) was carried on back end memory controller by using the Xilinx Timing Analyzer Tool. As a requirement for the tool, the User Constrain File (UCF) was created with the following four timing interfaces such as Input PAD to FLOP, FLOP to FLOP, FLOP to Output PAD and Input PAD to output PAD

The following results were achieved from the Static Timing Analysis

T4 = Time taken by the Stage 4	= 2.57 ns
T3 = Time taken by the Stage 3	= 2.47 ns
T2 = Time taken by the Stage 2	= 2.51 ns
T1 = Time taken by the Stage 1	= 2.38 ns
Ts = Setup Time of the flip flop.	= 0.29 ns
Minimum period	= 2.86 ns
Maximum Frequency:	= 350.00 MHz

5.3.6 Data Path of the Memory Controller

Our proposed memory controller is capable of reading data from the memory device as well as writing data into the memory device. Both the read and write data are sent out through 64 bit bi-directional DQ bus along with bi-directional DQS strobe signal which is used to capture the read data. For the read process, the read data DQ and strobe DQS are driven by the memory device for each of the requestor representing different banks and ranks. Similarly, for the write process, the memory controller is capable of driving the write data through DQ bus along with DQS strobe signal. The memory model is design in such a way to send and receive the data and strobe signals.

Even though the memory controller design is capable of receiving read data from memory device, the received read data is not sent to the system bus in our design. We are not concerned with the system bus for our analysis, because the absence of this additional data processing to the system bus does not have any impact on the read and write latency that we are concerned for this thesis.

5.3.7 Testing of the Memory Controller

Identifying the main corner cases and writing tests to address those corner cases are the most challenging part of the simulation setup. As inputs to the design, various read and write memory traces were used as input stimuli. Once the test cases and inputs are ready, next important task was to design the memory model that would interact with our memory controller. The memory model was designed to mimic the interface of DDR3 memory device. Next, the test bench structure was designed where Design under Test (DUT) and the memory model were instantiated. Test bench also includes the proper clocking and reset generation. For design entry, simulation and synthesis purpose, the Xilinx ISE Design suit, v14.4, was used.

To obtain the simulation results, we implemented the entire memory controller for the front-end for the command generator and back-end for the arbitration and memory timing check using Verilog RTL. Our implementation uses a fully pipelined architecture with four stages to increase the hardware speed. We synthesized the design using Xilinx Kintex 7 FPGA and obtaining a maximum command bus clock frequency of 350 MHz. While this frequency is lower than the 666 MHz frequency used in our simulations, we argue that an ASIC implementation would result in significantly higher speed. The next section details out the background information on hardware simulation setup and the evaluation results.

Chapter 6

Evaluation

The evaluation of our rank switching open-row memory controller design was carried out through hardware simulation process for various memory configurations. We evaluate the performance of our memory controller simulated with CHStone and SPEC benchmarks. Since AMC results are only suitable for critical requestors, we compare the latency results using critical requestor arbitration where CHStone and SPEC benchmarks were used. Further, the simulation for non-critical requestor arbitration was also carried out to measure the throughput for non-critical tasks. The hardware simulation used DDR3-1333H memory device with data bus size of 64 bits. Further, this analysis considers the memory device with 2 ranks or 4 ranks with 8 or 4 banks per rank respectively. This allows us to assign one bank to each requestor.

6.1 Synthetic benchmark Results

Synthetic benchmarks are used to show how the worst case analytical bound varies as a function of benchmark's parameters. Since the latency bound is a function of the number of open/close and load/store requests performed by the requestor under analysis, we decided to plot the average per-request worst case latency in nano-seconds (y-axis) for a synthetic task as we vary the row hit ratio (percentage of open requests, x-axis) and fixing the percentage of store requests to 20%. Both figures 6.1 and 6.2 plot results for data bus width of 64 bits and 16, 8 requestors respectively. Figure 6.3 shows the case of 8 requestors and 32 bits bus. From all three figures below, we can see that AMC's plot is constant in the graph since it uses close row policy; hence the latency does not depend on row hit ratio. When the number of requestors and ranks are increased, our approach performs comparatively much better. For 8 requestors with 32 bits bus and 0% row hit ratio, AMC still has 50% higher latency compared to ROC 4 rank scenario. Similarly, the latency from paper [2] is at least 50% higher than ROC 4 ranks overall cases. Note that the synthetic results does not account for the refresh latency.

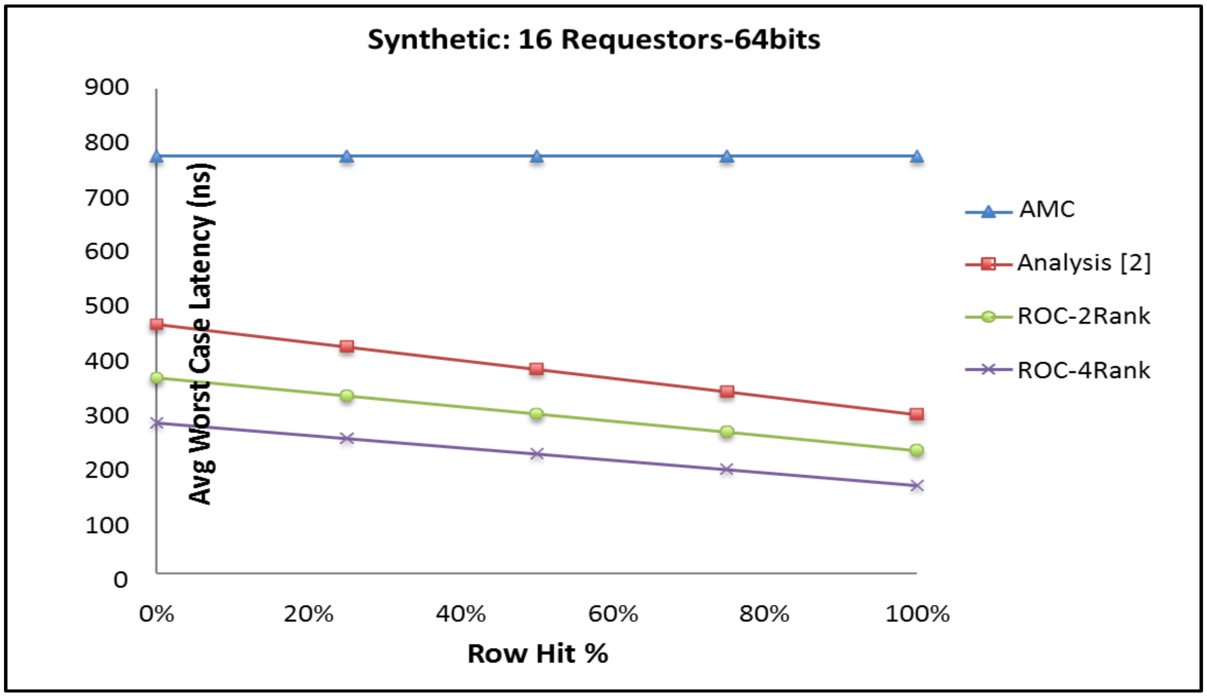


Figure 6.1: Synthetic 16 Requestors 64 bits data bus result

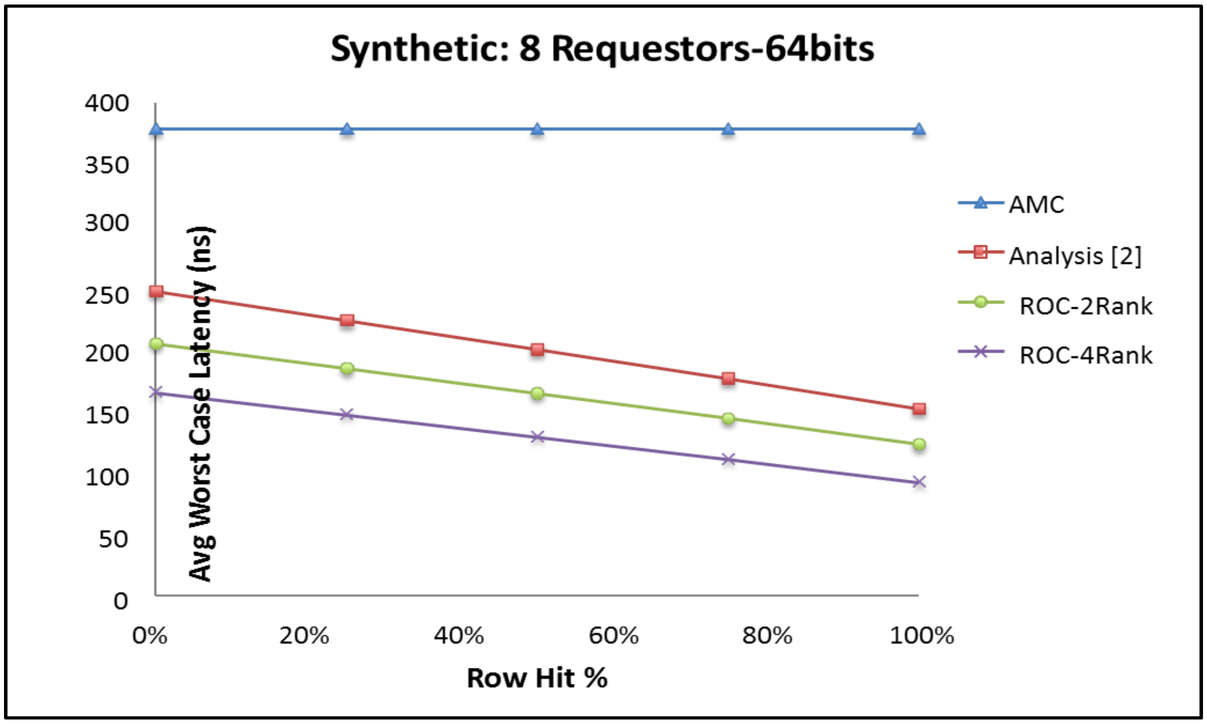


Figure 6.2: Synthetic 8 Requestors 64 bits data bus result

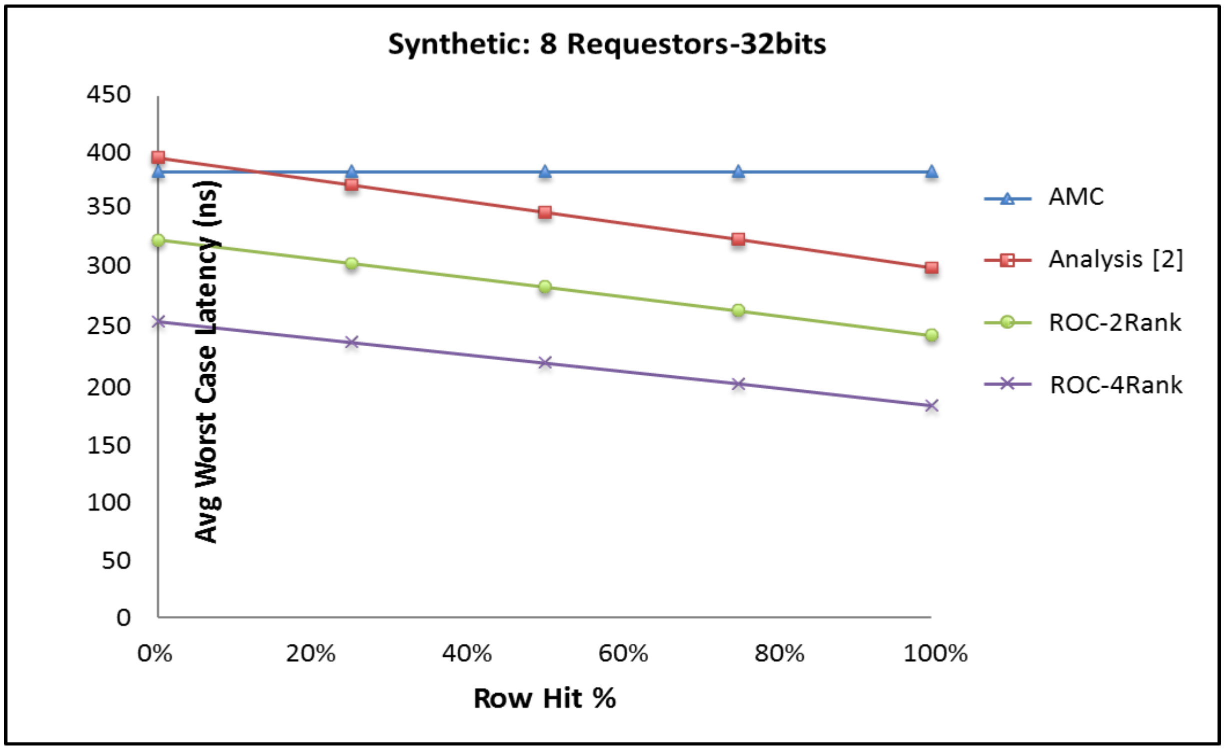


Figure 6.3: Synthetic 8 Requestors 64 bits data bus result

6.2 Latency of open and close memory read access

The memory access can be one of the four types such as open read or close read or open write or close write. In this experiment, the worst-case latency for memory read and memory write is derived from the simulation for the request under analysis, REQ0 while other requestors are used to apply interference. The latency time means the time taken from the moment the read command is sent out from the front end until the read data is received by the back end logic from the memory device. Every time a command is dispatched from the front end, it is identified as either open read or close read. For both open read and close read, the latency time is captured and recorded into an output file. Out of all the captured latency times, only the worst case values in each memory access category is considered for this analysis to be compared against synthetic analysis results. We only consider open read and close read since the read latency is much more critical than write latency. Therefore, the theoretical values are compared with hardware simulation results for the open read and close read with 0 % write scenario. The experiment was carried out for the memory configuration of 16 requestors with 4 ranks and 2 ranks as shown in Table 6.1. We experimented with many different benchmarks from CHStone family to get the worst latency delay for open read and close read. Note that theoretical analysis results did not include the delay caused by refresh. Therefore, we did this experiment by turning off the refresh operation in hardware simulation in order to make the correct comparison with the theoretical results. For the hardware simulation, the open and close latency delay is extracted as number of clock cycles and it is multiplied by the clock period of 1.5 ns to get the actual delay as shown in Table 6.1.

	REQs = 16; Ranks = 2		**REQs = 16; Ranks = 4**	
	Theoretical Analysis	**HW simulation**	**Theoretical Analysis**	**HW simulation**
Open Read 100 %row hit	**230.5 ns**	**222.0 ns**	**162.5 ns**	**136.5 ns**
Close Read 0 %row hit	**364 ns**	**277.5 ns**	**278 ns**	**211.5 ns**

Table 6.1 Latency of open, close access with 0 %write

6.3 Simulation of Critical tasks only

In this case, all 16 requestors are assigned to critical tasks. Figure 6.4 shows the Memory configuration 1 where the first Requestor, named REQ0, is assigned as requestor under analysis and the remaining requestors, REQ1 to REQ15 are used to provide extensive interference to the Requestor under Analysis, REQ0. The Requestor under Analysis receives memory request inputs from CHStone benchmark family that include mips, adpcm, aes, bf, gsm, dfadd, dfdiv, dfmul, dfsin, jpeg, motion and sha. Other Requestors, REQ 1 to REQ 15 receives memory request input from LBM of SPEC benchmark family to provide the timing interference to Requestor under Analysis, REQ0. After the simulation is completed, the total execution time for Requestor under Analysis, REQ0, was derived from the simulation output.

This process was repeated for each of the twelve memory traces of CHStone benchmark family. Having achieved the execution time for each benchmark, the graph is plotted for execution time against each of the CHStone benchmark. This process is repeated for all four configurations that are listed below where all the requestors are assigned as critical applications. Figure 6.4 shows Configuration 1.

1. Configuration 1:	Requestors = 16,	Ranks = 4,	Banks = 4
2. Configuration 2:	Requestors = 16;	Ranks = 2;	Banks = 8
3. Configuration 3:	Requestors = 8;	Ranks = 4;	Banks = 2
4. Configuration 4:	Requestors = 8;	Ranks = 2;	Banks = 4

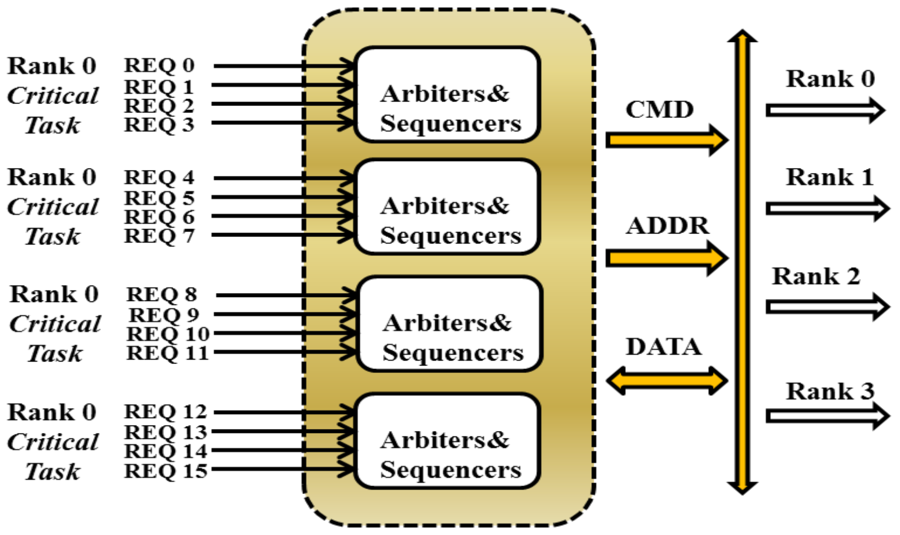


Figure 6.4 Simulation setup for Memory Configurations 1

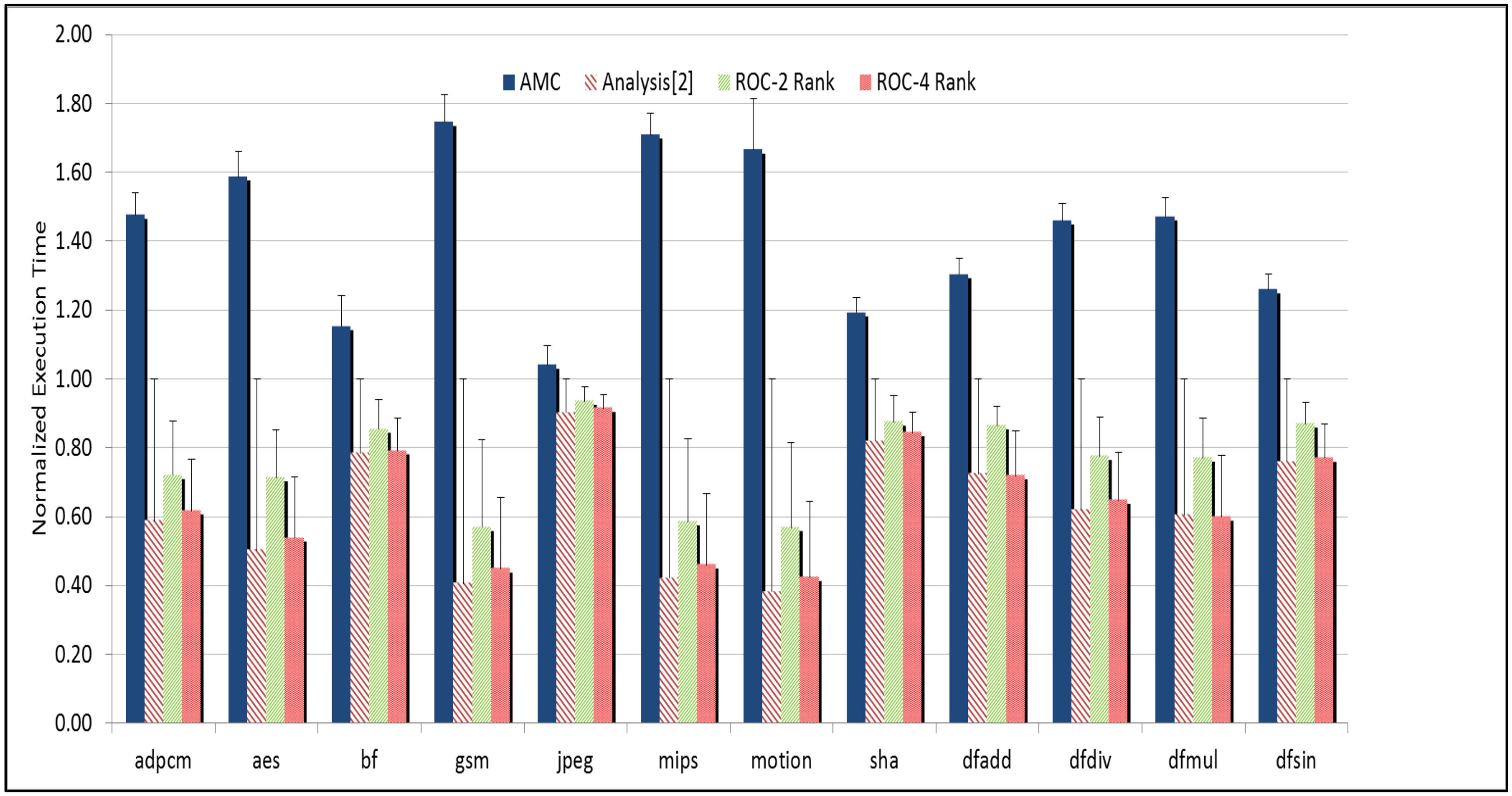


Figure 6.5: CHStone: 16 Requestors with Ranks 4 and Rank 2

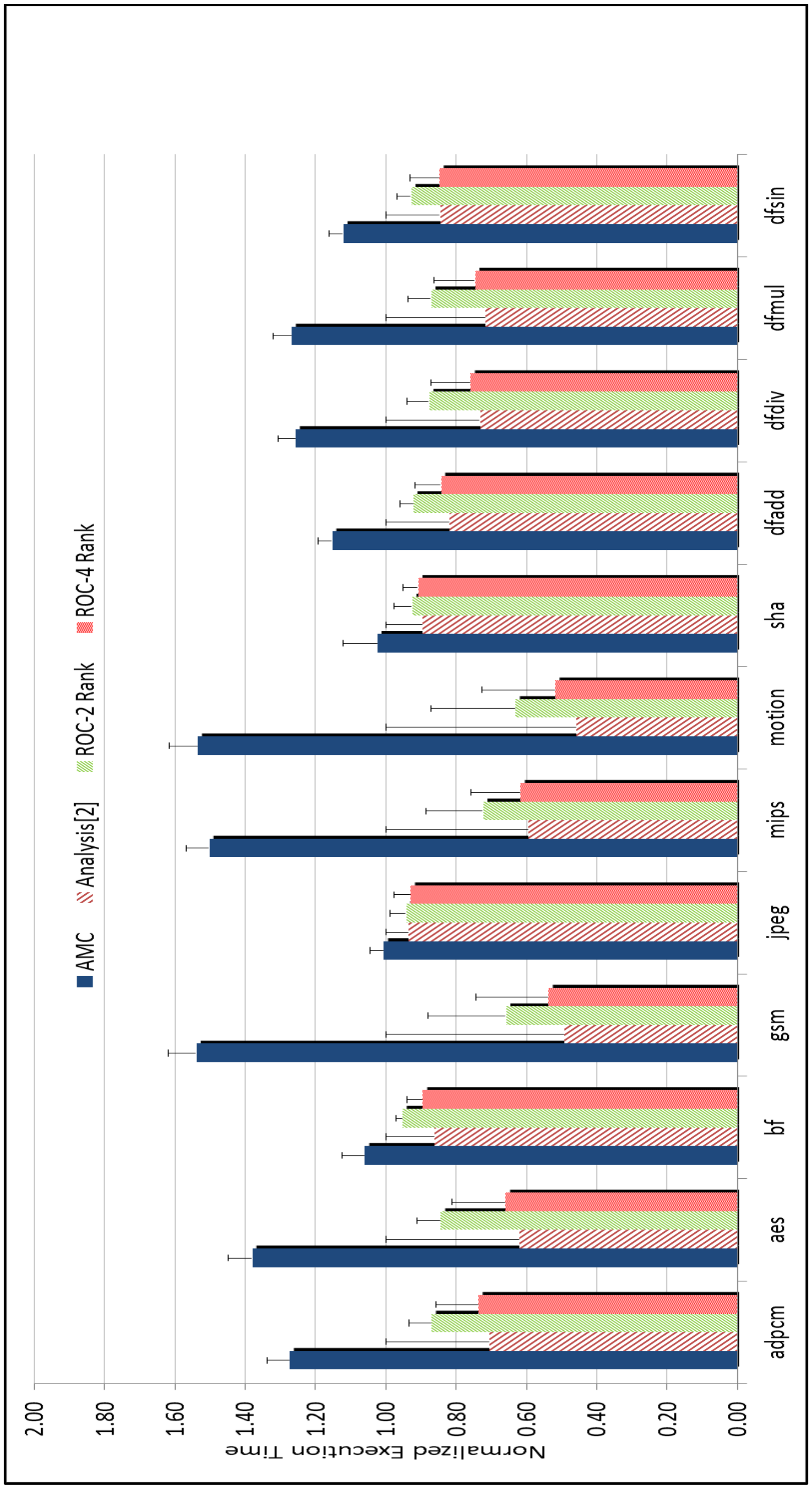


Figure 6.6: CHStone: 8 Requestors with Rank 4 and Rank 2

The Figures 6.5 and 6.6 show the total execution time (y-axis) consumed by the Request under Analysis (REQ0) for each of the memory traces (x-axis) in CHStone benchmark family. Figure 6.5 is derived when the 16 requestors are in action making memory requests to the memory controller for both 4 ranks and 2 ranks scenarios. Similarly, Figure 6.6 is derived when the 8 requestors are in action making memory requests to the memory controller for 4 ranks and 2 scenarios. The y-axis is the normalized execution time of the benchmarks against the worst case analytical bound of our published paper [2]. The T-bars are the worst case analytical bounds while rectangular boxes with shades are simulation results. In terms of analytical bounds, the results of our memory controller with 4-ranks and 2 ranks performs well compared to the theoretical results of our published paper [2]. It also performs well compared to the AMC performance. But, as you can see from Figure 6.5, and 6.6, latency from hardware simulation is higher than the software simulation results. This is due to the fact that hardware design uses the three stages of pipelines with four level of arbitration. The difference between simulated and analytical time is always quite small for AMC, less than 10%. However, our simulated time is significantly lower than the analytical bounds. This is because the analysis assumes a precise worst case pattern of interfering requests by other requestors. The probability that such pattern is produced at run-time is very low, albeit non-zero.

6.4 Simulation of Critical and Non-Critical tasks

This section analyses the scenario where both critical and non-critical requestors do make memory requests at the same time to our memory controller. Memory configuration 1 is used for evaluation with the following setup. Rank 1 and Rank 2 are assigned with 8 critical requestors whereas the Rank 3 and Rank 4 are assigned with non-critical requestors as shown in Figure 6.7. The LBM memory trace from SPEC benchmark family is split into four sub-files as per address range, 64 bytes and each sub files are sent to each of the four banks in rank 3 and rank 4. For this mixed simulation, critical tasks will be made to be in-order and the non-critical tasks will be made to be out of order. It is worth to mention how in-order and out of order requests are made at the front end. When each request arrives at the front end, it carries the type of request (read or write), physical address and a delta delay. This delta delay indicates the time difference between the last requests to current request. This delta delay between requests makes them to be in-order. On the other hand, to generate the out of order requests, the incoming requests with fixed delay will be used. This fixed delay for each request will enable the out of order transfer for the non-critical task.

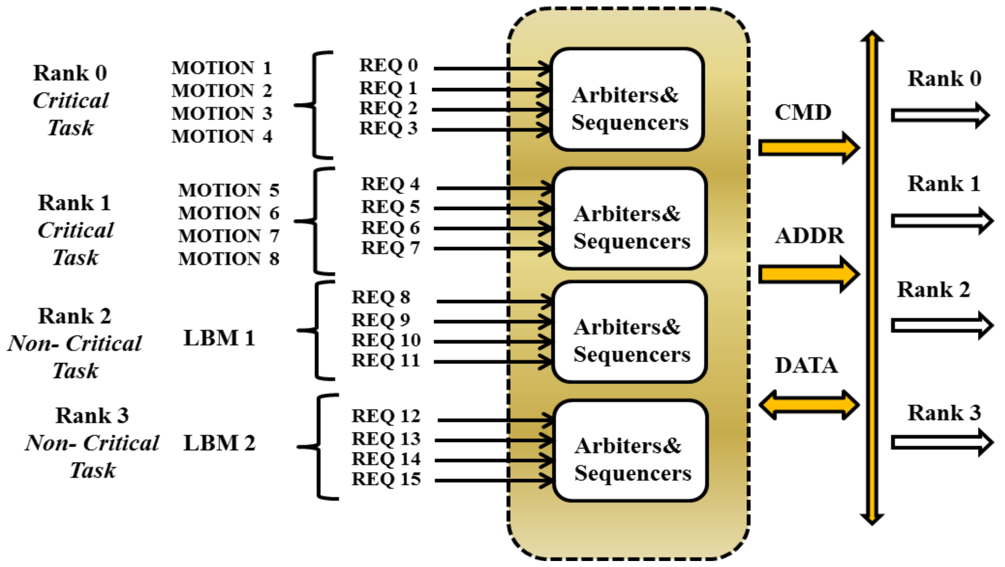


Figure 6.7 Simulation setup for Critical and Non-Critical requestors

The bandwidth calculation is carried out when both critical (motion) and non-critical (LBM) requestors are sending requests to the memory controller at the same time. As shown below, the bandwidth for critical and non-critical are calculated to achieve the total bandwidth.

$$\text{Critical (Motion)} = \frac{\text{Data Transferred by REQ0}}{\text{Time taken by requestor completed first.}}$$

$$\text{Critical (Motion)} = \frac{\text{Number of data completed * burst length * bus size}}{\text{Time taken by the requestor completed first.}}$$

$$\text{Critical (Motion)} = \frac{575 \text{ x } 64 \text{ Bytes}}{112860 \text{ ns}}$$

$$\textbf{Critical (Motion)} = \textbf{326 MB/sec}$$

$$\text{Non-Critical (LBM)} = \frac{\text{Data Transferred by (SUM \{REQ8 to REQ11\})}}{\text{Time taken by the requestor completed first.}}$$

$$\text{Non-Critical (LBM)} = \frac{\text{\# of data completed * burst length * bus size}}{\text{Time taken by the requestor completed first.}}$$

$$\text{Non-Critical (LBM)} = \frac{\{731 + 724 + 721 + 725\} * 64 \text{ Bytes}}{112860 \text{ ns}}$$

Non-Critical (LBM) = 1645 MB/sec

Total Bandwidth = Critical * 8 + Non-Critical * 2

= (326) * 8 + (1645) * 2 MB/sec

Total Bandwidth = 5.898 GB/sec

Theoretical Bandwidth = Frequency * Bus width

= 1333 MHz * 8 Bytes

Theoretical Bandwidth = 10.664 GB/sec

Since we included the rank switching techniques in our design, our total bandwidth may not reach close to the theoretical bandwidth value. Our design is able to achieve 5.898/10.664 = 55.3 % bandwidth which is somewhat close to the expected rate of 66 % due to the addition of rank switching logic in our memory controller design. For the future work, the design can be optimized further to get higher bandwidth.

Chapter 7

Conclusion

A rank-switching open-row memory controller design for mixed-critical system is presented in this thesis. Our design was built to handle dynamic command scheduling while existing memory controllers solely rely on static command scheduling. The existing memory controllers usually take advantage of the close row policy to easily handle the complex timing constrains. But, our objective is to utilize both open row policy and private bank mapping to offer the worst case latency for the critical requestors and minimum average bandwidth for non-critical requestors.

Further, our rank-switching mechanism improves the utilization of the data bus by guaranteeing that consecutive data transfers are spaced by at most one rank to rank transition delay. This delay is shorter than the write to read and read to write delays that apply to the data transfers of the same rank. As a result, our proposed rank switching memory controller design significantly improves the worst case latency of memory requests while guaranteeing the isolation among requestors.

Our evaluation is carried out for both critical and non-critical requestors. The evaluation on critical requestor has demonstrated reduction in latency for critical requestors. The outcome of latency from our hardware simulation of critical requestors is compared against our theoretical values and AMC and our hardware design results perform well. For bandwidth, we evaluated the bandwidth for the critical and non-critical and calculated the total bandwidth which is compared against the theoretical bandwidth. The evaluation results show that our rank-switching open-row memory controller performs well as the number of ranks increases. As a future work, the design can be optimized to achieve higher speed.

References

[1] Krishnapillai, Yogen; Zheng Pei Wu; Pellizzoni, R, “A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems” in Real-Time Systems (ECRTS), 2014 26th Euromicro Conference, Publication Year: 2014, Page(s): 27 – 38

[2] Z. Wu, Yogen. Krishnapillai, and R. Pellizzoni, “Worst Case Analysis of DRAM Latency in Multi-Requestor Systems,” in Real-Time Systems Symposium (RTSS), 2013.

[3] Yonghui Li, Benny Akesson and Kees Goossens, “Dynamic Command Scheduling for Real-Time Memory Controllers” in Proc. Euromicro Conference on Real-Time Systems (ECRTS), 2014

[4] Leonardo Ecco, Sebastian Tobuschat, Selma Saidi, and Rolf Ernst, "A Mixed Critical Memory Controller Using Bank Privatization and Fixed Priority Scheduling" in Proc. of the 20th IEEE International Conference on Real-Time Computing Systems and Applications (RTCSA), August 2014 => close

[5] D. T. Wang, “Modern DRAM Memory systems: Performance Analysis and Scheduling Algorithm,” Ph.D. dissertation, University of Maryland at College Park, 2005.

[6] S. Kim, S. Kim, and Y. Lee, “DRAM power-aware rank scheduling,” in ISLPED, 2012.

[7] M. Paolieri, E. Quinones, F. Cazorla, and M. Valero, “An Analyzable Memory Controller for Hard Real-Time CMPs,” Embedded Systems Letters, IEEE, vol. 1, no. 4, pp. 86–90, 2009.

[8] B. Akesson, K. Goossens, and M. Ringhofer, “Predator: a predictable SDRAM memory controller,” in CODES+ISSS, 2007.

[9] S. Goossens, B. Akesson, and K. Goossens, "Conservative Open-row Policy for Mixed Time-Criticality Memory Controllers," in DATE, 2013.

[10] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation," in CODES+ISSS, 2011.

[11] B. Akesson, L. Steffens, E. Strooisma, and K. Goossens, "Real-time scheduling using credit-controlled static-priority arbitration," in RTCSA, 2008.

[12] R. Bourgade, C. Ballabriga, H. Cass, C. Rochange, and P. Sainrat, "Accurate analysis of memory latencies for WCET estimation (regular paper)," in RTNS, 2008.

[13] I. Liu, J. Reineke, and E. A. Lee, "A PRET Architecture Supporting Concurrent Programs with Composable Timing Properties," in ASILOMAR, 2010.

[14] S. A. Edwards and E. A. Lee, "The Case for the Precision Timed (PRET) Machine," in DAC, 2011.

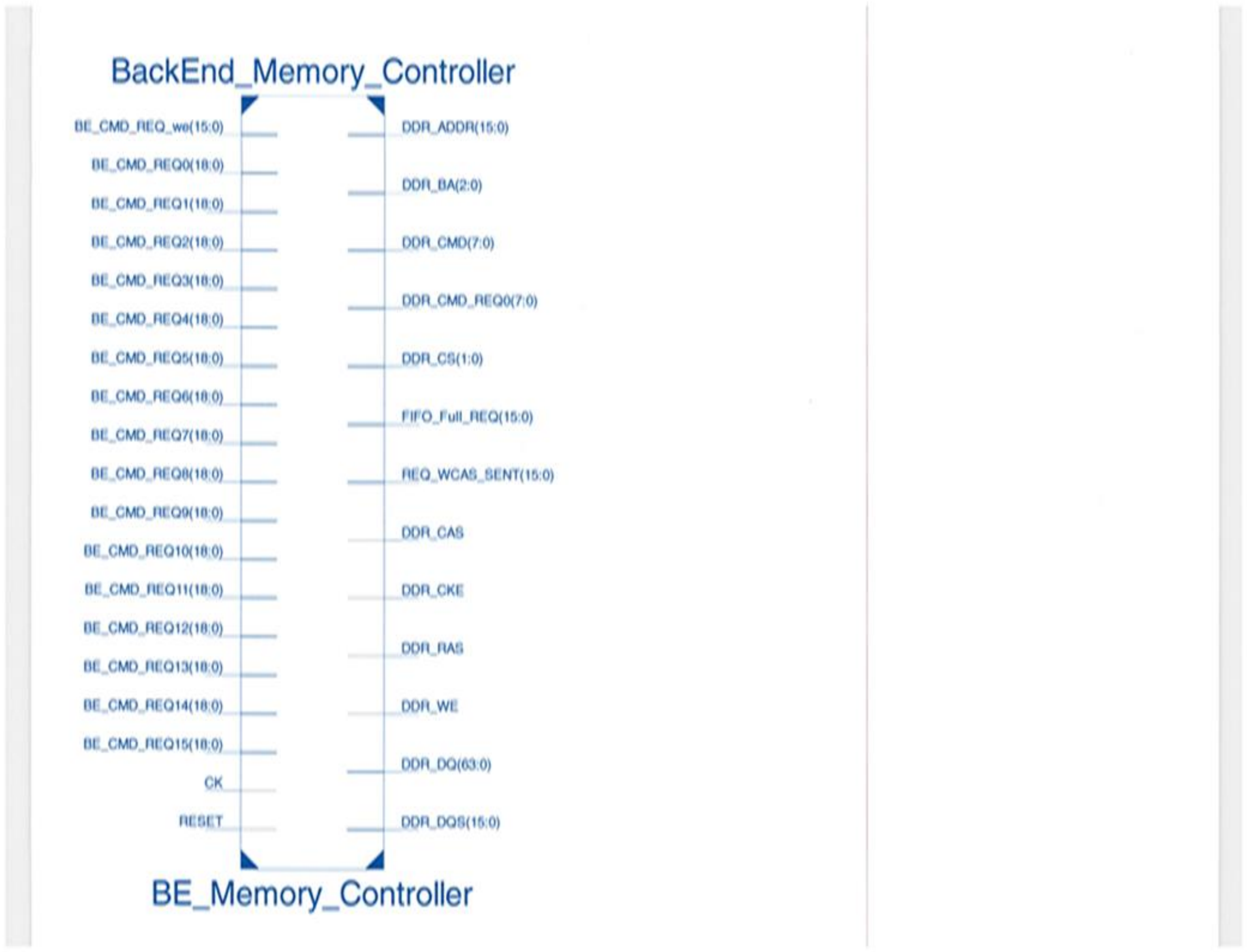
[15] D. Bui, E. A. Lee, I. Liu, H. D. Patel, and J. Reineke, "Temporal isolation on multiprocessing architectures," in DAC, 2011.

[16] JEDEC, "DDR3 SDRAM Standard JESD79-3F," July 2012.

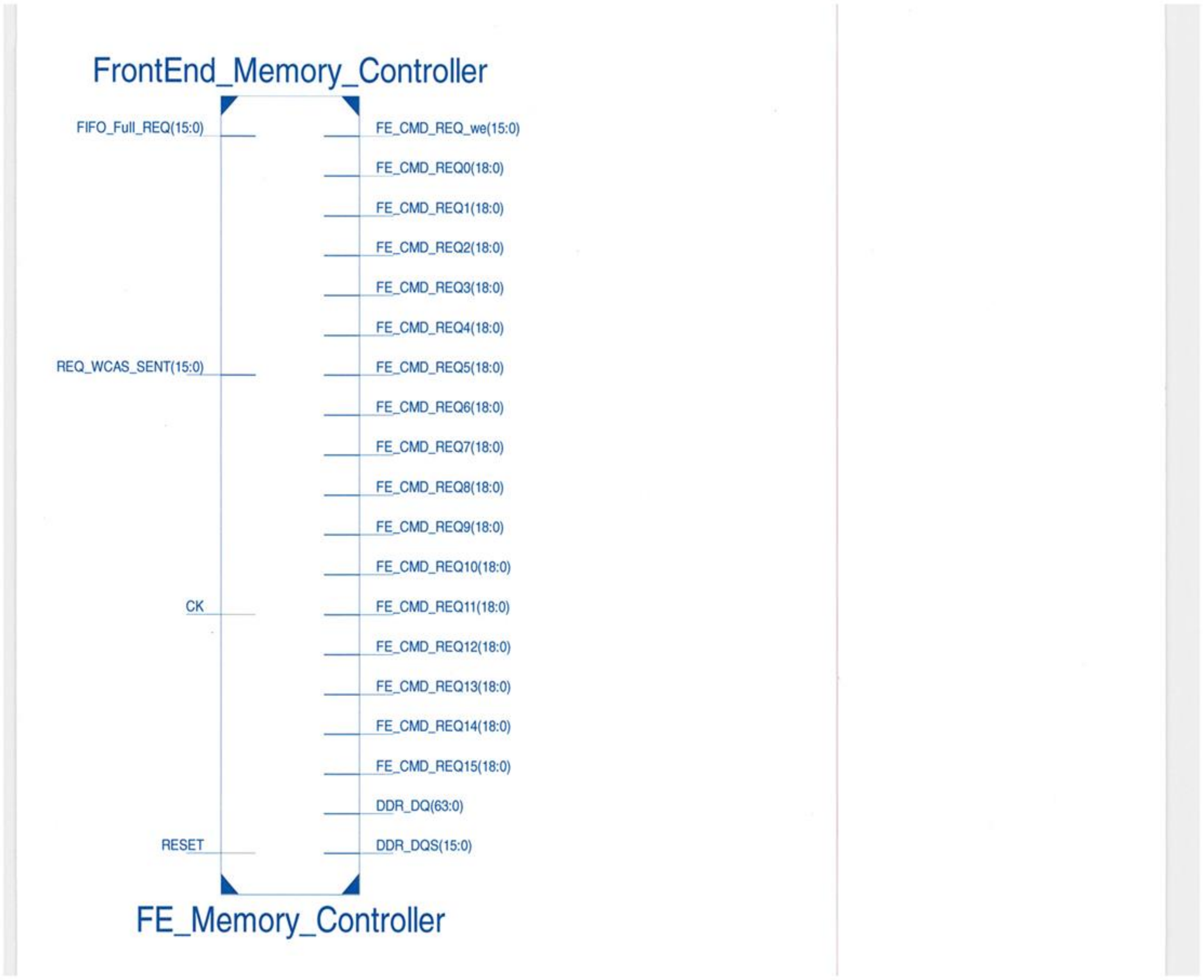
[17] Zheng. Wu, "Worst Case Analysis of DRAM Latency in Hard Real Time Systems, MASc Thesis, University of Waterloo, 2013.

[18] Memory Controller Design code designed for this thesis is available at http://ece.uwaterloo.ca/rpellizz/techreps/roccode.zip

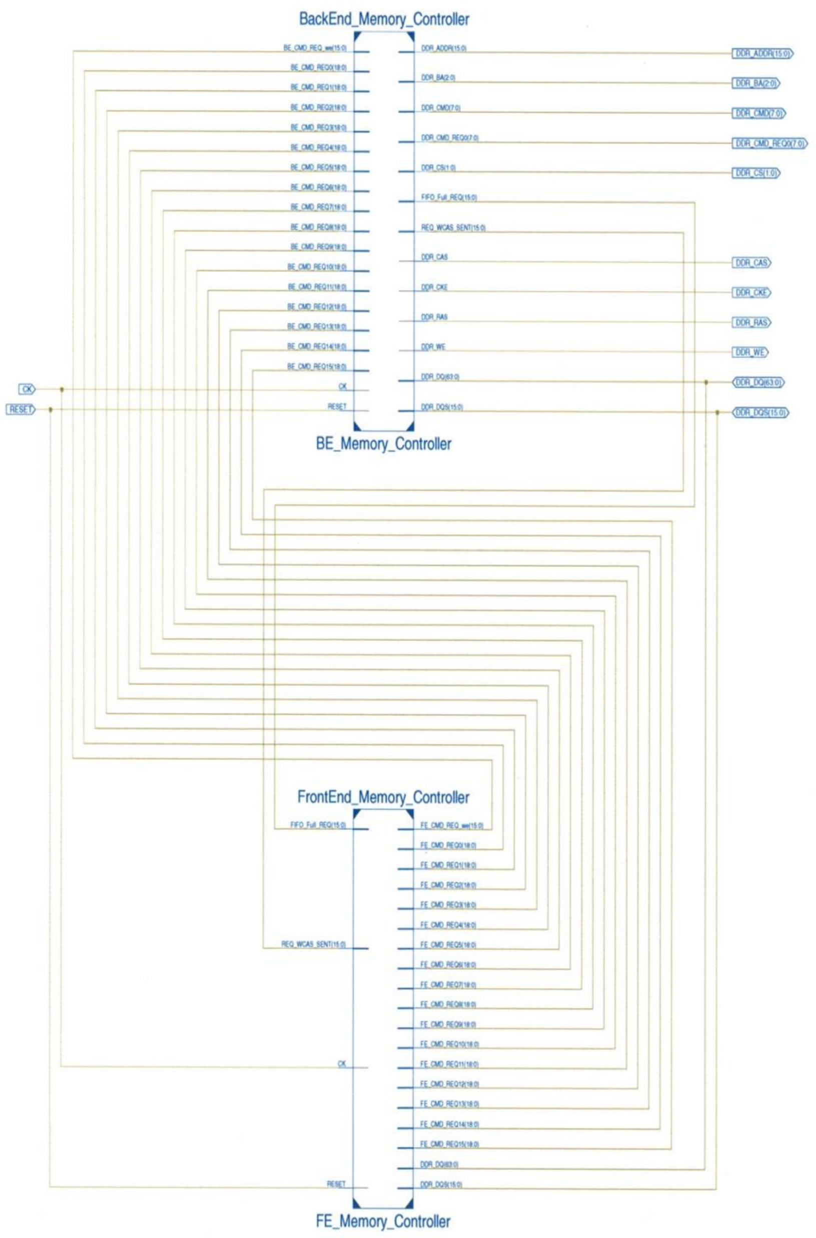
Appendix A: Design Block Diagrams



Back End Block Level View

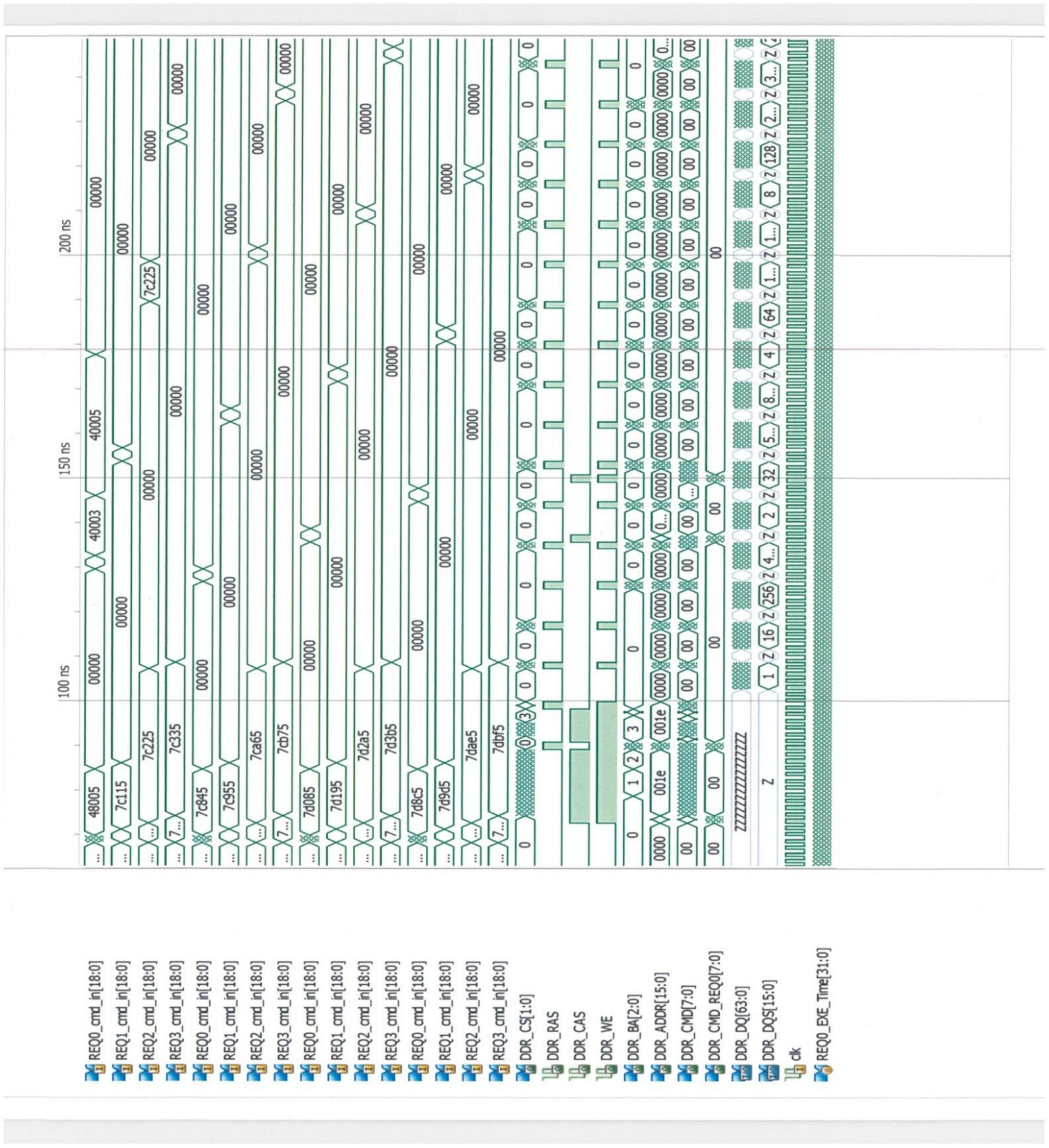


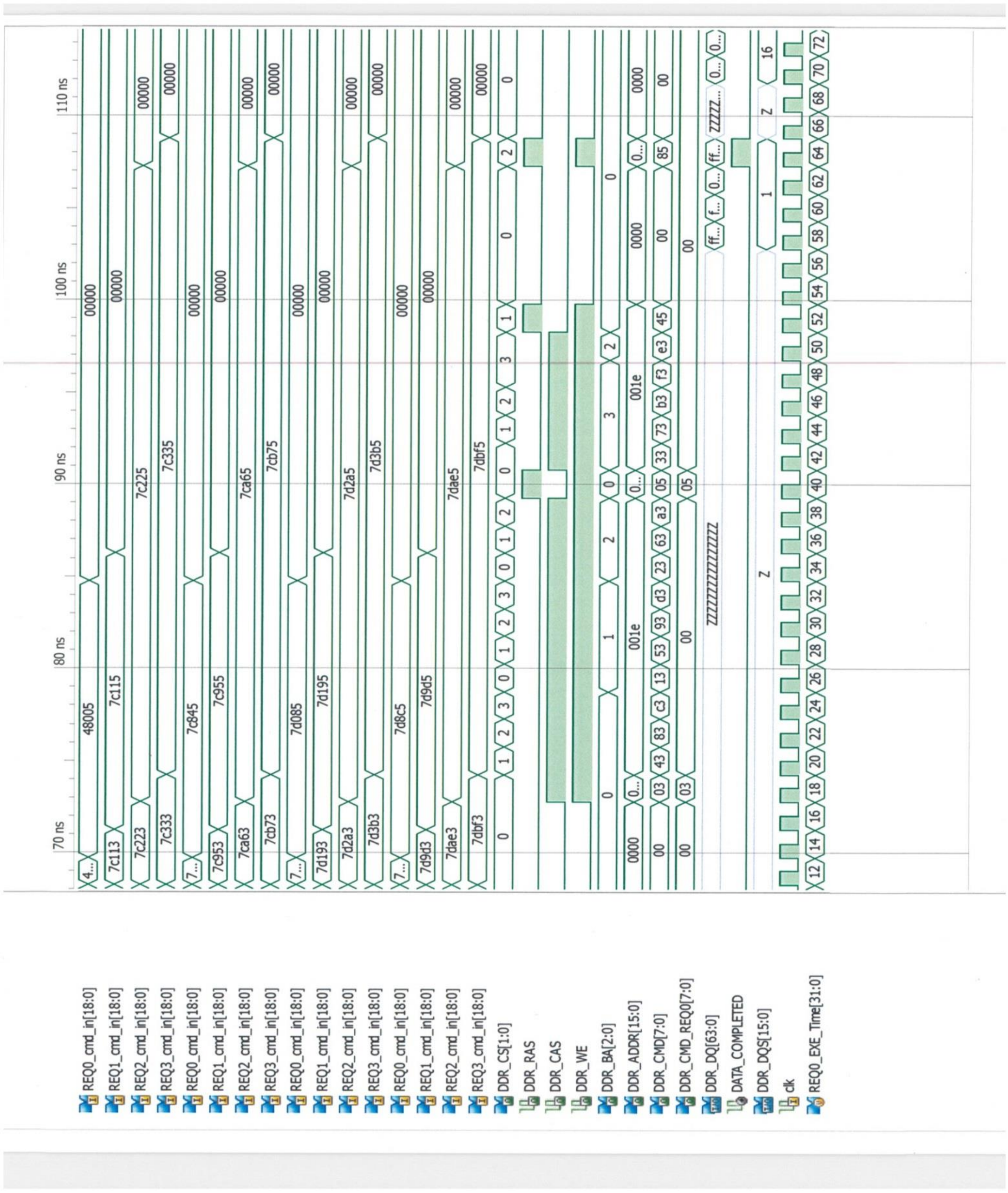
Front End Block Level View



Front and Back End Block Level View

Appendix B: Simulation Output – Example snapshot




REQ0_cmd_in[18:0]
REQ1_cmd_in[18:0]
REQ2_cmd_in[18:0]
REQ3_cmd_in[18:0]
REQ0_cmd_in[18:0]
REQ1_cmd_in[18:0]
REQ2_cmd_in[18:0]
REQ3_cmd_in[18:0]
REQ0_cmd_in[18:0]
REQ1_cmd_in[18:0]
REQ2_cmd_in[18:0]
REQ3_cmd_in[18:0]
REQ0_cmd_in[18:0]
REQ1_cmd_in[18:0]
REQ2_cmd_in[18:0]
REQ3_cmd_in[18:0]
DDR_CS[1:0]
DDR_RAS
DDR_CAS
DDR_WE
DDR_BA[2:0]
DDR_ADDR[15:0]
DDR_CMD[7:0]
DDR_CMD_REQ0[7:0]
DDR_DQ[63:0]
DATA_COMPLETED
DDR_DQS[15:0]
clk
REQ0_EXE_Time[31:0]
70 ns
80 ns
90 ns
100 ns
110 ns