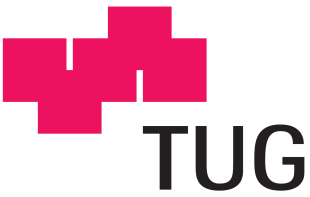
Magisterarbeit

Design and Implementation of a Memory Controller for Real-Time Applications

Markus Ringhofer

Institut für Technische Informatik
Technische Universität Graz
Vorstand: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß



Begutachter: O. Univ.-Prof. Dipl.-Ing. Dr. techn. Reinhold Weiß
Betreuer: Ass.-Prof. Dipl.-Ing. Dr. techn. Christian Steger

Eindhoven und Graz, im September 2006

Abstract

A memory controller is proposed where the behavior is predictable and which can be used for real-time systems. The memory controller is designed to be used in any system where a predictable memory controller is required, but integrates very well with Æthereal. It offers predictability per flow, which guarantees minimum bandwidth and maximum latency, without the necessity of simulation. Furthermore, the memory controller translates the peak memory bandwidth (gross bandwidth) to a net bandwidth, which is the effective available bandwidth for the IP blocks. The proposed solution has a worst-case gross-to-net bandwidth translation efficiency of 80%. The maximum latency for the highest priority requestor is $375ns$. The memory controller is synthesized for CMOS12 and results in a small scalable memory controller with $0.04mm^2$ for 6 requestors.

Keywords: Memory Controller, Network-on-Chip, DDR2, SDRAM

Acknowledgments

This thesis was written at the Institute of Technical Informatics at the Graz University of Technology in cooperation with the Embedded Systems Architecture on Silicon Group at Philips Research in Eindhoven, The Netherlands.

I would like to acknowledge Dr. Kees Goossens for supervising me during my stay at Philips Research. I would also like to thank Benny Åkesson for the close collaboration throughout the course of this project.

I also owe my gratitude to Prof. Lambert Spaanenburg for organizing the road trip from Lund University to Philips Research, which resulted in an internship opportunity at the Philips Research Labs and thus leading to this thesis. Furthermore, I thank him for the supervision on the thesis. Special thanks go to Vince as well, who as a work mate and a friend, made coffee breaks and lunches more fun.

Danksagung

Diese Diplomarbeit wurde im Studienjahr 2005/06 am Institut für Technische Informatik an der Technischen Universität Graz und in der Gruppe Embedded Systems Architectur on Silicon bei Philips Research in Eindhoven (Niederlande) durchgeführt.

Spezieller Dank geht an meinen Betreuer Herrn Ass.-Prof. Dr. Christian Steger von der TU Graz für die akademische Betreuung.

Bedanken darf ich mich auch für die Unterstützung und das Verständis von meiner Freundin Sophie. Bei meinen drei besten Freunden Günter, Herbert und Wolfgang bedanke ich mich fuer die netten Gespräche und für gelegentliche gemeinsame Bierchen. Vielen Dank auch an Andy, Karin und Thomas, die in ihrer Wohnung immer Platz für mich hatten, wenn ich in Graz war.

Im Speziellen bedanke ich mich bei meinen Eltern, Erwin und Maria, die mich in meinen Entscheidungen stets bestärkt haben, und auf deren Unterstützung ich mich immer verlassen konnte.

Eindhoven und Graz, im September Markus Ringhofer

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter gives an introduction to the research area and starts with a motivation of the topic in Section 1.1. In Section 1.2 the goal of the thesis is stated. In Section 1.3 the structure of the thesis is outlined.

1.1 Motivation

Systems-on-Chip (SoC) and in particular embedded real-time systems typically consist of several computational elements. These elements fulfill different tasks for processing an overall solution. Let's take a set-top box for TV sets as an example [GGRN04]. A set-top must generate a TV-signal for a particular TV channel from a digital satellite signal. This process takes different tasks. One task is to split the incoming digital signal into data streams, such as video and audio. Another task is to convert the video stream into an actual TV-signal. One more conversion has to be made to turn the audio stream into an audio signal for the TV set. Meanwhile, another task handles the user input such as changing the channel when the remote control is pressed. All these tasks have to be done in parallel and are bound by real-time deadlines. The cost of missing these deadlines is visible as black boxes on the screen or audible as noise. This is unacceptable and therefore it is necessary to always deliver this data within hard real-time deadlines.

These computational elements are either general-purpose processors or digital signal processors. Nowadays, multiple of them are integrated into a System-on-Chip solution [CS99]. A processor needs to interact with other processors, memories or I/O devices to complete a task. Currently busses are used to interconnect these IP blocks. The current research in the field suggests using Networks-on-Chip (NoC) to interconnect IP blocks, because NoCs allow more flexibility than busses [GGRN04, RGAR$^+$03].

However, to get NoCs accepted as communication paradigm in SoCs there are still left open research questions according to [OHM05]. An example is how to deal with voluminous storage. High volume storage is usually put off-chip as dynamic memory. The separation is necessary because the manufacturing process is a different one for memories to that for standard logic. Dynamic memories provide high data rates and high storage capabilities. Nevertheless, dynamic memories lack flexibility in accessing random memory locations efficiently and require refresh cycles to keep the content. Allowing efficient communication between IP blocks and a shared memory requires a memory controller. IP blocks, which

communicate with the memory, are named hereinafter requestors. The memory controller arbitrates between the requestors and manages the memory accesses. The goal of this thesis is to design a memory controller to provide hard real-time guarantees, and provides an efficient communication between IP blocks and the memory. The memory controller should also be easily/efficiently combined with the interconnection.

1.2 Goal

The goal of this thesis is to provide a memory controller for real-time systems, which can be used with Æthereal [GDG+05]. Æthereal is a NoC proposed by Philips Research. The memory controller is co-designed with the network infrastructure of Æthereal. The design of the memory controller and Æthereal focus on predictable performance. To keep the overall system predictable the memory controller has to be predictable as well. Identifying the key elements of Æthereal leads to distinct goals, which are required for a memory controller suitable for real-time systems. The identified requirements for the memory controller are:

- predictable
- gross-to-net bandwidth translation
- efficient resource utilization
- requestor isolation

Predictability means that the memory controller can be analyzed analytically and worst case scenarios can be calculated at design time. A gross-to-net bandwidth translation allows to calculate from the peak memory bandwidth (gross bandwidth) the net bandwidth, which is the effective available bandwidth for the requestors. Efficient resource utilization is another goal for this memory controller, allowing an economical use of the resources of the system. The requestor isolation allows that every requestor can be analyzed individual without considering the traffic patterns of the other requestors in the system. The memory controller is designed in a way so it can be used in any system where a predictable memory controller is required.

1.3 Structure

The thesis is organized as follows: In Chapter 2 the related work is discussed. In Chapter 3 the design process of the memory controller is described and the design of the memory controller is proposed. Chapter 4 describes the implementation and the verification process. Furthermore, synthesis results are presented. Chapter 5 shows the simulated performance of the memory controller with a Philips Mid-End multimedia SoC use-case. Chapter 6 concludes the thesis with a summary and questions for further research work.

Chapter 2

State of the art

This chapter summarizes the current technologies and architectures proposed in the field, which are related to the topic of the thesis. It starts with an introduction on dynamic memories in Section 2.1. In the following Section 2.2 the concept of memory controllers is discussed and it is illustrated with a memory controller proposed by others. The real-time requirements are discussed in Section 2.3. Networks-on-Chip architectures are discussed in Section 2.4, where the focus is on Æthereal. Section 2.5 discusses the related work.

2.1 Dynamic memory models

This section gives an overview of dynamic memories. Starting from the history and background of memories in Section 2.1.1 the architecture is discussed in Section 2.1.2. Memory efficiency is discussed in Section 2.1.3 and the memory mapping is discussed in Section 2.1.4.

2.1.1 History

Storage plays an important role in virtually any computer system and in system development. Since 1975 DRAMs are used in computer systems as the main memory [HP03]. Robert Dennard invented the concept of DRAMs in 1968. The first letter stands for dynamic, and was introduced to allow reducing the number of transistors per stored bit. One transistor is used per bit, and the information is stored as charge in a capacitor. The capacitor is part of the transistor. Reading from a memory cell can disturb the information in the transistor, therefore it is necessary to precharge a memory cell after accessing it. However, not only accessing a memory cell disturbs the information. The self-discharge of the capacitor also disturbs the information. Therefore, a regular refresh must be invoked to recharge the memory cells [HP03].

The advantage of DRAMs compared to static RAMs (SRAMs) is that DRAM memory cells are smaller in size and hence allow a higher integration capacity per area unit. SRAMs use a bistable Flip-Flop to store information. A SRAM bit cell requires 6 transistors. The advantage of SRAMs is that they are faster and do not require a recharge mechanism. Furthermore, SRAMs have lower power consumption. It is possible to put a SRAM into an effective low-power mode or standby mode. SRAMs find their application on high levels of the memory hierarchy in areas where fast memories are required, such as in caches or

in scratchpads. The application of DRAMs is where more storage is required, but the latency is not so critical compared to the application domain of SRAMs [HP03].

2.1.2 Memory architecture

The architecture of DRAMs allows packing more storage into the memory than with the architecture of SRAMs. This causes that the width of the address lines to grow in size too. With the concern of costly pin count in mind, the address is separated into column and row information and is multiplexed onto the address bus of the memory. The architecture of an SDRAM (Synchronous DRAM) is three dimensional (bank, row, and column) and is illustrated in Figure 2.1.

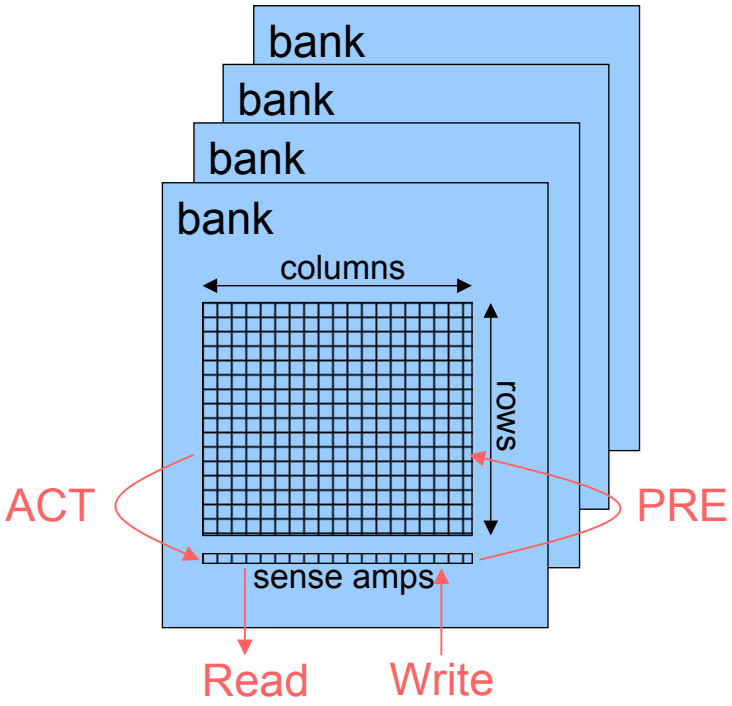


Figure 2.1: DRAM architecture and access cycle

Double Data Rate SDRAM (DDR) memories are DRAM memories introduced as high bandwidth memories. DDR memories transfer data at both clock edges. The DDR standard is an open standard and defined by the Joint Electron Device Engineering Council (JEDEC). The DRAM memory is specified in [Ass04].

The row address is sent first to access a memory cell, called the row access strobe (RAS). The information of the cells is fetched into an intermediate buffer, the sense amplifiers. With providing the column address and the information of the type of the access (read or write), the correct columns in the sense amplifiers are selected. For a read access a data burst is sent to the output. The memory location is now ready to be precharged with the unmodified content of the sense amplifiers. This mechanism is required since reading the memory cells results in changing the content of the memory cells. For a write access a data burst is fetched from the data input and is stored at the selected column

position of the sense amplifiers. The memory cells are updated with a precharge with the current information from the sense amplifiers [HP03, Ass04].

The access pattern of DRAMs follows the below stated and simplified principle[1], and is shown in Figure 2.1:

1. Activate a row (ACT)
2. Read or write to a column within the row (RD/WR)
3. Precharge the row (PRE)

A memory controller translates the accesses from the requestors into commands understood by the memory. Table 2.1 lists the most frequently used commands.

Command	*Description*	*Opcode*
NOP	do nothing	none
ACT	Activate	row
RD	Read	column
WR	Write	column
PRE	Precharge current row	row
REF	Refresh	none

Table 2.1: Some SDRAM commands

To access the memory the timing parameters of the memory have to be considered. See Table 2.2 for a selected choice of timing parameters taken from the DDR2 specification in [Ass04]. After activating a row, the time t_{RCD} has to be elapsed to continue with a read or write access. To precharge a row again the time t_{RAS} after the activate command has to be elapsed to invoke a precharge command. For the analysis of the memory efficiency in the next Section 2.1.3 the timing parameters are considered in more detail.

The time t_{RC} is the minimum time which has to elapse until an activate command is allowed to follow a previous activate command to the *same* bank. Whereas the time t_{RRD} is the minimum time which has to elapse until an activate command is allowed to follow a previous activate command to a *different* bank. The time t_{RCD} has to elapse after an activate command until a read or write command can be invoked to the same bank. The time t_{WTR} is the required time on the data bus for switching the directions from write to read.

2.1.3 Memory efficiency

Designing embedded systems requires being economical with the use of integrated components. Embedded systems have the requirement to be small in area size and should have a low energy consumption to allow extended operation time on battery power. However, when using dynamic memories, 100% efficiency cannot be reached due to protocol limitations. Dynamic memories have the advantage to provide more storage at a lower cost but also have the drawback that the memory content needs to be refreshed regularly. During

[1] It is possible to access another column in the same open row, without the necessity of precharging and activating the same row first.

Parameter	*Min. time [ns]*	*Min. time [cycles]*	*Description*
t_{CK}	5	1	Clock cycle time
t_{RAS}	45	9	Active to precharge command delay
t_{RC}	60	12	Activate to activate command delay (same bank)
t_{RRD}	7.5	2	Activate to activate command delay (different bank)
t_{RCD}	15	3	Activate to read/write delay
t_{RFC}	75	15	Refresh to activate command delay
t_{RP}	15	3	Precharge to activate command delay
t_{REFI}	7800	1560	Average refresh to refresh command delay
CL	15	3	Column-access-strobe (CAS) latency
t_{WTR}	10	2	Write-to-read turn-around time
t_{WR}	15	3	Write recovery time

Table 2.2: Selected timing parameters for DDR2-400 256 Mb device CL=3

the refresh cycle no data can be transferred to or from the memory. Therefore, those cycles are lost. Regular refreshes are not the only cause why dynamic memories cannot reach 100% efficiency. Equation (2.1) gives a general definition for measuring efficiency in a system with a dynamic memory [Wol05, HP03].

$$efficiency = \frac{data\_cycles}{total\_cycles} = \frac{total\_cycles - lost\_cycles}{total\_cycles} \tag{2.1}$$

The following paragraphs will discuss other sources of efficiency losses. The following losses are referenced in literature as most important [Wol05, HP03]:

- refresh efficiency
- data efficiency
- bank conflict efficiency
- read/write efficiency
- command conflict efficiency

Keeping the efficiency high overcomes the necessity of introducing faster and wider memories. Introducing faster and wider memories cause the system to be more expensive and to consume more energy.

Refresh efficiency

To retain the content of a dynamic memory, it is necessary for the memory content to be refreshed regularly. During the time when the memory is being refreshed, no data can be transfered to and from the memory. This causes a loss of data transfer cycles, and has negative affects on the efficiency. For DDR2 devices, it is necessary to refresh the

memory once every t_{REFI}, 7.8μs according to Table 2.2. The maximum amount of lost cycles occurs when all banks have to be precharged before the refresh takes place. The refresh efficiency is traffic independent. The specification of the DDR2 memory allows the postponing of refresh cycles up to 9*t_{REFI}. However, after this interval, 8 refresh commands have to be sent consecutively. The refresh efficiency is calculated as shown in Equation (2.2), where N is the number of consecutive refreshes.

$$refresh\_efficiency = 1 - \frac{1}{t_{REFI} * N} (t_{RFC} * N + t_{PrechargeAll}) \tag{2.2}$$

$t_{PrechargeAll}$ is the sum of t_{RAS} and t_{RP}. Calculating the *refresh_efficiency* for $N = 1$ and using the timing parameters specified in Table 2.2 results in the following efficiency:

$$refresh\_efficiency = 1 - \frac{1}{1560 * 1} (15 * 1 + 9 + 3) = 98.27\%$$

The efficiency loss for the refresh cycle is 1.73% and can be lowered when invoking more refresh commands in a consecutive sequence. The reason is that all banks are precharged in a single go, and so the efficiency is improved [Wol05, Å05, Ass04].

Data efficiency

Data is retrieved from the memory in bursts. The bursts cannot start at any arbitrary address. If the required data is contained in the middle of a burst then the remaining data is transmitted without any purpose. See Figure 2.2 for an illustration.

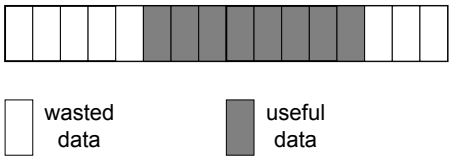


Figure 2.2: Half of the data is lost due to poor data alignment

Data efficiency depends on the burst size. The efficiency is higher for a smaller burst size. Nevertheless, allowed choices of burst sizes depend on the memory type used. The memory controller cannot influence the data alignment nor do the memory controllers usually solve it. It is considered a software problem because software is more susceptible to influence the efficiency of data alignment [Wol05, Å05].

Bank conflict efficiency

A bank conflict happens, when a row in a bank is open and the next access targets another row in the same bank. The contents of the open row are still in the sense amplifiers and therefore the current row has to be precharged. After invoking the precharge command, the time $t_{PrechargeToACT}$ has to elapse until another row in the same bank can be activated. This delay causes a loss of cycles and results in a lowered efficiency. The bank conflict problem can be overcome by reordering the bursts or requests. The goal is to delay the access to the same bank and schedule other non-conflicting bursts in the mean time. This

solution leads to an increase in efficiency. However, the latency increases when reordering is considered and has to be factored in the design specification.

Furthermore, when reordering read/write accesses, read/write hazards have to be considered. Read/write hazards occur when a later arriving write access is scheduled earlier than a previously arriving read access to the same address. In the case of reordering, the read access will get modified data back, instead of the old data as expected. It is possible to track read/write hazards by analyzing the addresses of the accesses. When two accesses are identified as possible read/write hazards, the system does not allow reordering of these two bursts. This mechanism comes at the expense of additional logic. Nevertheless, it allows free reordering of the bursts and requests to increase efficiency. However, data reordering comes with the additional expense of buffering allowing reordering the requests [Wol05, Ass04].

Read/Write change efficiency

The memory controller is communicating with the memory for transmitting read and writes data over a bidirectional data bus. When a read access is followed by a write access additional time is needed to change the direction of the bus. The same is necessary in the reverse scenario when there is a switch from write to read. However, during the idle state, no data is transferred over the bus and this affects the efficiency of the memory access. The idle cycles needed to switch from read to write is defined as *read_to_write_cost* and can be calculated according to the DDR2 protocol specification [Ass04] with Equation (2.5).

$$RL = CL \tag{2.3}$$

$$WL = CL - 1 \tag{2.4}$$

$$\begin{aligned} read\_to\_write\_cost &= 2 + WL - RL = \\ &= 2 + CL - 1 - CL = \\ &= 1 \end{aligned} \tag{2.5}$$

The cost of switching from read to write is static and is one clock cycle. Switching from write to read is defined as *write_to_read_cost* and is stated in Equation (2.6).

$$\begin{aligned} write\_to\_read\_cost &= CL - 1 + t_{WTR} + RL - WL = \\ &= CL - 1 + t_{WTR} + RL - (RL - 1) = \\ &= CL + t_{WTR} \end{aligned} \tag{2.6}$$

The cost for switching from write to read depends on the timing parameters of the memory. According to the timing parameters specified in Table 2.2 the costs are 5 clock cycles. The read/write change efficiency is traffic-dependent. Woltjer [Wol05] analyzed the read/write change efficiency for CPU traffic containing 30% write accesses and 70% write accesses and a block size of 32 words, and got an efficiency of 93.8%. Improving this efficiency is possible by favoring read after read and write after write sequences and reducing read/write switches. However, the efficiency gain comes with the cost of higher latency and additional buffering [HdCLE03, ARM04].

Command conflict efficiency

DDR allows transferring one data word per clock edge to or from the memory so that two data words are transfered per clock cycle. Nevertheless, sending commands from the memory controller to the memory is only possible once every clock cycle. An access to a memory destination usually consists of three commands: Activate, read/write, and precharge. This leads to the possibility of command-bus contention. For a burst size of 4 words a read or write command has to be placed on the bus every second clock cycle. Thus every other clock cycle is available for an activate and precharge command. The utilization of the command bus by read or write commands is then 50% and the remaining 50% are left for activate and precharge commands. For longer bursts such as 8 words a read or write command has to be invoked every 4 clock cycles. This causes the command bus to be utilized for 25% and the remaining 75% can be used for activate and precharge commands. For a burst size of only 2 words a read or write command has to be issued every clock cycle and this leaves no space for additional commands. However, additional commands are required to access the content of the memory. This results in a reduced efficiency since data cannot be provided at every clock. This loss in efficiency is called the command conflict efficiency.

The command conflict efficiency can be improved by using bigger burst sizes. Using a bigger burst size also affects the bank conflicts and increases the bank conflict efficiency. Another possibility of improving the command conflict efficiency is to use read and write commands with auto precharge. This means that rows are automatically precharged after they are accessed. Furthermore, the DDR2 protocol introduced an additive latency (AL) for read and write commands. This leads to more flexibility for sending read and write commands over the command bus.

The use of autoprecharge is appropriate when having accesses, which address different rows in the same bank. The advantage with autoprecharge is that only two commands have to be invoked for an access. First activate and then read or write with autoprecharge. However, using autoprecharge is inappropriate when accessing a different address in the same row and bank. In this case, the row can be left open and accessed immediately. Advanages of open and closed page policies are investigated by [RDK+00]. Additional logic is required to track the dependencies when accessing the same row.

A different approach improving the command conflict efficiency is to influence the order of the commands. Consider the case when a command sequence consists of ACT, RD/WR, and PRE. Consider the following situation, to complete an access to an open bank a precharge is missing, but another access to another bank is waiting. In this case, one of the two commands has to be stalled. Either the read/write or the precharge command can be delayed. Stalling the read/write command results in immediate loss of data cycles and reduces efficiency. Stalling the precharge will stall future accesses to the same bank by one clock cycle because the precharge will finish later. In the DDR2 protocol additive latency is introduced to overcome this problem. This allows sending read/write commands immediately after an activate command. The read/writes are delayed internally to assure the required timing behavior. This solution does not add any additional space on the command bus, but it introduces some flexibility in the timing of sending memory access commands.

The command conflict efficiency is hard to analyze and needs to be estimated for every

application independently. Nevertheless, Woltjer [Wol05] is estimating the loss around 0 - 5%.

2.1.4 Memory mapping

A memory map is consulted to translate the logic address into a physical address. The physical address of a DDR memory consists of the information: bank, row, and column. Multiple banks are introduced to allow higher bandwidth. However, higher bandwidth is reached with an interleaving memory access pattern to hide activate and precharge times. Figure 2.3 shows an illustration of an interleaved memory map. In this example only five bits are used to keep it simple. The most significant bit determines which row to access. In the example a full burst consists of eight words, which means that the eight words have to be spread over all banks. A way to do that is to access consecutive two words in the same bank, switch the bank and access again two consecutive words, and so on. To get an access pattern as described above two consecutive words are mapped into one bank. This can be gained by taken the least significant bit and the bit right to most significant bit and arrange them together as column address. The bank bits are chosen analogously, to get an interleaved access pattern, where an eight word accesses targets 4 banks. The drawback of this memory map is that a minimum burst size is required to hide activate and precharge time, but leads to a 100% bank access efficiency [HP03, Å05].

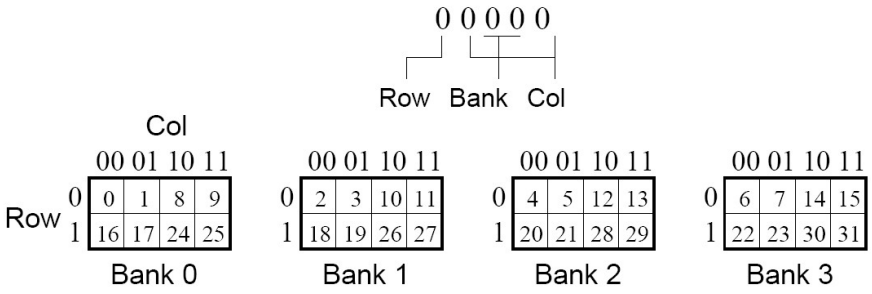


Figure 2.3: Memory map for an interleaved access [Å05]

Another way of partitioning and mapping the memory is to make separate address spaces for the accesses. Which means that every access targets only a bank. See Figure 2.4. The two most significant bits determine, which bank to access. The other bits translates the logical address into row and column information [HP03, Å05].

The memory map is a powerful tool for a system architect, which allows specifying how the requestors access the banks. Section 3.3.2 will discuss the impact on the system architecture level further.

2.2 Memory controller

This chapter briefly summarizes the architectures of memory controllers designed by others. It starts off with an introduction of the place of a memory controller in the system. After it discusses the different architectures of memory controllers.

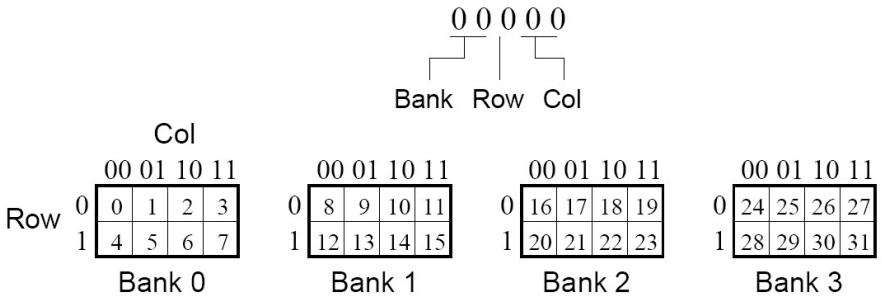


Figure 2.4: Memory map for sequential access [Å05]

A memory controller translates the accesses from the requestors into commands understood by the memory. The memory controller is placed in a SoC, where the IP blocks are interconnected with each other and further to a memory controller. The memory controller is then connected to an off-chip dynamic memory [Wol05, RDK+00, Web01]. See Figure 2.5 for an illustration.

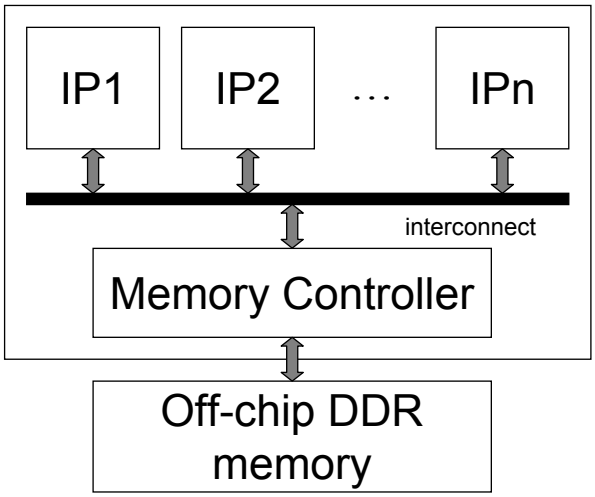


Figure 2.5: Memory controller place in the system

A memory controller is considered to consist of the following elements: memory mapping unit, the arbiter, the command generation unit, and the data path (see Figure 2.6 for an illustration). The memory mapping unit translates the logical addresses to a physical address of the DDR memory. The arbitration unit picks the requests according to the scheduling strategy. To translate the accesses from the requestors to DDR bursts, the command generation provides corresponding commands to access the DDR memory. The data is transported on the data path to and from the memory [Wol05, ARM04, Web01].

In the introduction it is motivated that current computer systems require fast access to high volume storage with high throughput. Therefore, memory controllers are required which provide a high bandwidth and low latency access to the off-chip memory. Different types of memory controllers are proposed, which make attempts to fulfill these constraints.

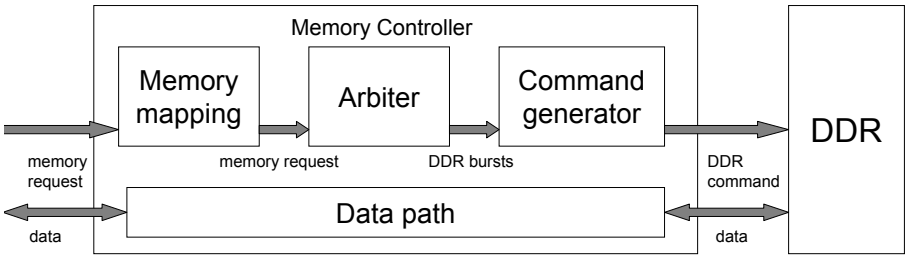


Figure 2.6: Architecture of a memory controller

An exponent of memory controllers is the PrimeCell^{TM}Dynamic Memory Controller proposed by ARM. The controller is designed to be connected to the IP blocks via an ARM bus, like AMBA or AXI. The arbitration between multiple requestors is carried out with an round-robin arbitration mechanism. Read accesses are provided with a quality-of-service (QoS) mechanism which allows providing low latency. Therefore, an adapted round robin mechanism is used. Higher priority is given to these requestors thus allowing lower latency. This mechanism is not provided for write accesses. However, there are no hard bounds provided in the case of a read request either [ARM04].

The aforementioned memory controller is a general purpose memory controller and is not optimized for a special application, unlike the memory controller proposed by Rixner et al. in [RDK^{+}00]. This memory controller provides high efficiency due to pipelined accesses with an interleaved access pattern. To get interleaving access pattern, the scheduler reorders the requests and reorders accesses within requests [RDK^{+}00]. However, requestor isolation cannot be guaranteed due to the dependencies caused by reordering of the accesses.

Lee et. al. [LLJ05], proposes a memory controller with a layered architecture and a quality-aware scheduler. The functional layers are used to achieve high DRAM utilization. Layer 0 is considered as memory interface socket, which is stated to allow a quick integration into an arbitrary system. The quality aware arbitration is integrated in Layer 1. The goal of the memory controller is to provide high memory efficiency [LLJ05]. The two aforementioned memory controllers of [RDK^{+}00] and [LLJ05] both allow providing soft-real-time guarantees. However, both lack providing hard-real-time guarantees.

Weber is proposing in [Web01] a memory controller which is connected to a network and which allows providing quality-of-service. The memory controller proposes to provide high bandwidth combined with low latency. The high bandwidth is reached with reordering the traffic to get an interleaved access pattern. The arbitration is done with a 6-stage filter, where the request is picked, which causes the lowest cost. The cost is measured with the loss of efficiency combined with the quality-of-service level [Web01].

The memory controller described by Heithecker in [HdCLE03] is similar to the memory controller suggested by Weber. However, the memory controller from Heithecker is not used with a network and proposes a lower latency than the memory controller of Weber. The lower latency is reached due to a 2-stage arbitration mechanism instead of a 6-stage arbitration filter. The memory controller is optimized application specific for high-end image processing. Due to the expected reduced application space it is implemented in an FPGA [HdCLE03].

2.3 Real-time requirements

This section describes briefly the requirements of real-time systems and their applications. Starting with the definition taken from [Lap93] where a real-time system is considered as:

> A real-time system is a system that must satisfy explicit (bounded) response-time constraints or risk severe consequences, including failure.

With other words the key element of a real-time system is to react on events (possible external) within a certain deadline otherwise the system could break thus risking severe consequences. Severe consequences could be harmful for example when a nuclear power plant is overheating or a car with digital speed control stops accepting break signals when driving 150 km/h on the motorway. However, not all consequences have to be as bad as described above. As stated in Section 1.1 with the set-top box example, the missing of deadlines could results in black boxes on the screen and does not harm people. However, this is not acceptable either. Therefore, real-time systems require to react within a certain time frame and thus providing a maximum response latency. Providing a maximum response latency at every possible situation leads to a system, which is predictable at every moment in time [Lap93].

However, there are also other requirements for real-time system as identified above. These scenarios have in common that they are all fault critical, with different consequences. To design a system, which is fault tolerant it is necessary to test the system exhaustively or to make a formal verification of the system to assure the functionality. Therefore, it is necessary to design and maintain tests, which cover critical test cases [Lap93].

Other requirements of real-time applications are, according to [Lap93], concerned to the budget of a project. This means that a selected solution should contain a hardware/software mix, so that the system is cheap to build. This includes that a decision has to be made if the system should consist of general-purpose components or should be specific for this particular problem set.

2.4 Network-on-Chip architecture

The current process technology allows the integration of complex SoCs with many IP blocks. To catch up with Moore's law, it is mandatory to integrate more IPs into a single die. This allows to have a shorter design cycle by reusing IP blocks and the design process becomes more economical [SSM+01, BM02]. Currently buses or switches are used to interconnect IP blocks. As investigated in [BM02, SSM+01, GDR05] current buses do not scale anymore with the current integration speed of SoCs. The current research in the field suggests using Networks-on-Chip (NoC) to interconnect IP blocks, because NoCs allow more flexibility than buses [GGRN04, RGAR+03].

A key idea of NoCs is to decouple computation from communication. An example of a SoC is that a processor performs a task by computing tokens, which are passed on to an other processor. This processor takes the token, performs a computation and passes the token further. However, passing on the tokens causes stalling of the sending processor if the bus arbiter does not allow the sending processor immediately access to the bus. Then the sending processor has to wait until the arbiter grants the processor access to the bus.

This behavior denotes that the processor cannot continue its usual work, and stalls. To overcome this problem a separation between calculation and communication is performed. A processor adds the information where it belongs to the token and passes it over to a communication unit. This communication unit passes the token further on to the receiver. This communication architecture is called a Network-on-Chip, and the communication unit a network interface (NI). The NI is directly connected to an IP block, preferably with standard bus protocols (like AXI, AHB, OCP) maintaining the compatibility with existing IP blocks. The IP blocks themselves take care of the computation of tasks and forward the token to a buffer in the NI. The NI distributes the data to the targets. This approach leads to a communication based design, where communication is considered as important as computation [SSM+01, RGAR+03].

The current research proposes the Network-on-Chip concept to overcome the mentioned problems of traditional bus interconnect, which is lacking flexibility and scalability [DRGR03]. Figure 2.7 shows the principle of an on-chip network.

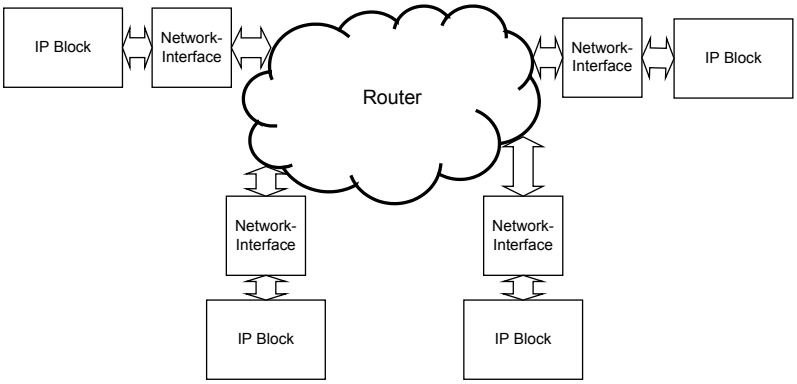


Figure 2.7: Principle of a network on a chip

Network-on-Chip research has much in common with the research carried out on computer networks. One of the differences is the use of relativly shorter wires. This guarantees short time delay between the components and further allows tight synchronization between them. On the other hand, new problems arise such as the limited amount of buffering on a chip, which is expensive in terms of chip area [RGAR+03, SLKH02].

2.4.1 Æthereal

Æthereal is a Network-on-Chip concept proposed by Philips Research Laboratories [DRGR03, GDG+05, GDR05, GRG+05]. The aim is to provide a scalable interconnect with communication guarantees between the IP blocks, and to allow connecting IP blocks with standard interfaces (like AXI) to the network. The provided communication guarantees are guaranteed throughput (GT) and best effort (BE). The GT traffic class is used to provide hard-real-time guarantees in the system.

The data passes through the network in packets. Every IP core is connected to a network interface, which in turn is connected to a router. For transmission of data, the IP block offers transactions to the NI. The NI packetizes the transaction and passes the packets to the router where it is transported to the target NI. There, the packet is

stripped and the frame is passed on to the target IP block via a standard bus interface. Figure 2.8 shows the connection scheme and the network interfaces from the Æthereal network. The NI contains two modules, the kernel and the shell. The kernel implements the network behavior, passing the packets to the router. The shell translates the incoming and outgoing packets to standard protocols, e.g. AXI. By changing the shell, IP blocks with other interfaces are connected to the Æthereal network.

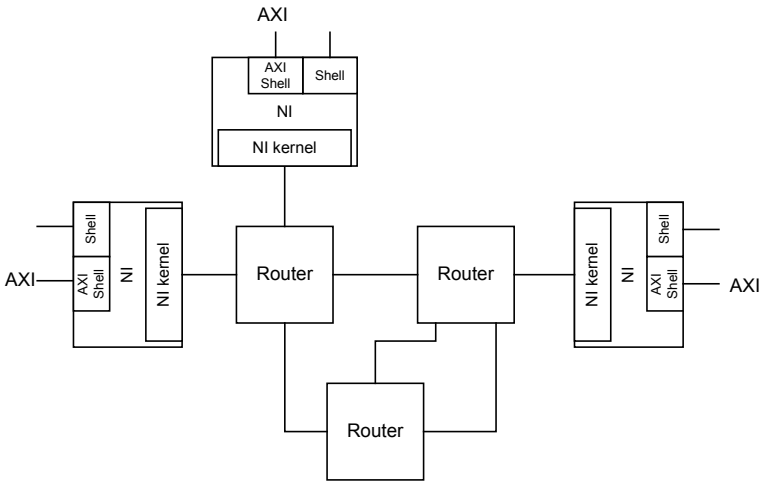


Figure 2.8: Æthereal network example

The network uses time-division-multiplexed circuit switching to route packets through the network. Source routing is applied to direct the packets. Furthermore, the packets are routed contention free. Every packet is chopped into small tokens, called flits. These flits are placed into pre-allocated slots, which belong to a particular traffic class and connection. The allocation of the slots is calculated off-line, when the network is set-up. The slot allocation algorithm calculates a global slot table, based on the information about the traffic class and the required bandwidth [RGAR+03, RDGP+05, DRGR03].

2.4.2 Proteo NoC

Proteo NoC is a Network-on-Chip concept proposed by the Tampere University of Technology [SSTN02]. This NoC has a packet switched communication network and is constructed with parameterized and reusable hardware blocks. The different functionalities in the NoC are organized according the ISO-OSI 7 layer model. IP blocks are connected to nodes, which are further connected to nodes or IP blocks. The nodes wrap the data from the IP blocks and forward them to the destination. There the packet is unwrapped and forwarded to the destination IP block. In case of a read request a request is sent to the destination IP block, which returns a response. In case of a write request the destination IP block acknowledges the received packet. However, in case of a malfunction (a packet lost in the NoC) the NoC detects the lost packet and re-transmittes the packet automatically [SSTN02].

The nodes are connected to other nodes or to IP blocks. In the case of two connected nodes an interconnection model is used for transmitting and receiving packets. The com-

munication between IP blocks is organized as point-to-point connection. A VHDL based design flow is used to derive the Proteo NoC for different use cases [SSTN02].

2.4.3 Xpipes interconnect

Xpipes is a NoC architecture proposes by the University of Bologna [BB04]. The NoC is designed as packet switched NoC, which targets Multi-GHz communication. The functional units of the NoC are designed as highly parameterized network components in synthesizable SystemC, which are re-used for different applications. The application definition is translated into a application specific NoC definition. With this NoC defintion the XpipesCompiler instantiates the network components accordingly and generates routing tables for the routing of the packet. The packets are routed with a static routing algorithm, called a street sign routine. As a result an interconnect is generated in synthesizable SystemC [BB04].

The NoC contains an error correction with packet retransmission. The errors are detected either with a Hamming code or a Cyclic Redundancy Check (CRC). The packets are sent from a transmitter to a receiver. The transmitter adds to the packets the CRC code. The receiver checks the CRC code and acknowledges the packet if received correctly. If the CRC code does not fit to the packet then a NACK (not acknowledge) is returned to the transmitter. Hence the transmitter starts to resend packets beginning from the first detected packet with an error. However, after the error was detected the receiver throws all received packets away until the re-transmitted packet arrived (even when the packets where received correctly). This mechanism allows saving buffering for reordering out-of-order packets, but increases the latency due to the unnecessary retransmission [BB04].

2.4.4 Decoupling of communication and computation

In a typical SoC, IP blocks are interconnected, and share a common external memory. The SoC performs operations on an input stream and delivers the result as an output stream. To perform an operation, the IPs has to know what to do, which means the IP blocks need instructions to run a program. The program code is fetched from an external memory or is located in a ROM. The data needed by the IP blocks is fetched from the memory or directly from other IP blocks. In this example, every IP block is accessing the memory. With the growing number of IP blocks, IPs only get a limited amount of accesses to the memory, and the memory becomes the bottleneck of the system. According to [HP03], this behavior is not only valid in this example system, it is shown that in most systems the memory becomes the bottle-neck. Having limited access to the memory means that the IP stalls when trying to fetch data from the memory, while an other IP block is already accessing the memory. A typical solution to overcome this problem is to introduce a small cache memory next to the IP. The cache keeps recently used data and instructions stored. If the IP block accesses a recently used memory location again, then it does not need to fetch the data or the instruction from the external memory, but instead the data is handed over by the cache, known as a cache hit. This makes the access to the memory faster and reduces the amount of transfers to the memory. According to [HP03], the reason this mechanism works so well is the following. The shape of the accesses follows the principle of locality, stated as rule of thumb, programs spend 90% of their execution times in 10% of the code [HP03].

However, the advent of cashes in the system not only reduces the amount of memory accesses, but it also hides the different operational speeds of the IPs with respect to the memory. The problem with external memories is that they are big in size but slow in terms of access time. A cache is faster because it runs at the same speed as the processors do, but limited in size due the chip area. This means that if there is an access to a new memory location, also known as a cache miss, then the cache fetches the requested data from an external memory and hands it over to the processor. If a cache miss occurs, the access is not faster than compared to a system without cache but applying the principle of locality leads to the conclusion that caches improve the average case [HP03].

Packet switched networks, like the Æthereal NoC, allow a clear decoupling of communication and computation. However, in order to have NoCs accepted in the SoC design process, a major concern is the aspect of buffering. The reason of the high costs of buffers is that it is expensive in terms of chip area. These buffers are needed to properly decouple computation from communication between interacting IPs and between the IPs and the network infrastructure. These buffers are integrated into the NI. The producer places data into the buffers of the NI. The network forwards the data in flits to the NI of the consumer, where it is stored and further forwarded to the consumer. If the buffers in the NI are too small, it causes the processor to stall and the processor cannot resume computing, which is not the intention of the decoupling buffers. Therefore, buffers have to be sufficient in size, but should not be over dimensioned and a waste of area.

In upcoming GALS SoC design, the IP blocks use different clock frequencies [GRG+05, SSTN02, MVF02]. Æthereal for example is equipped to handle different clock frequencies of the IP blocks and makes it possible to run the network on a faster speed to handle more data traffic. The required synchronization between the IP blocks and the network is carried out in the buffers of the NI [GRG+05].

2.4.5 Real-time guarantees

Introducing guaranteed services like GT and BE into a NoC provides real-time guarantees in a SoC. A naive way of calculating the buffer size is analyzing the traffic patterns of the producer. Assuming a periodic producer, the producer is allowed to send a guaranteed payload in a given period, see Figure 2.9 for an illustration, where Ti is the period and Di is the burst-size. The worst-case occurs if two packets are sent consecutively, which happens if the producer sends the data right before the end of the first period and then sends data again at the begin of the next period. This means that it is necessary to be able to buffer twice the guaranteed payload size. Figure 2.10 illustrates this worst-case behavior. However, knowing the arrival pattern and behavior of the system allows to optimize the buffering in the NI [GRG+05]. In [GRG+05], an algorithm for this optimization is given optimizing the buffer space and providing a full decoupling of computation and communication.

2.5 Previous work

Section 2.2 describes different types of memory controllers. Comparing this work with the described memory controllers, then this work is closest to the work carried out by Weber. However, there are certain differences in the design. This thesis proposes a memory

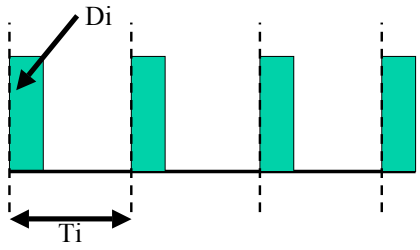


Figure 2.9: Production pattern of a regular producer

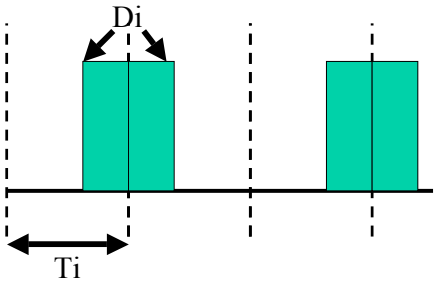


Figure 2.10: The worst-case of an irregular producer

controller, which is analyzable and is predictable. This is not the case for the memory controller designed by Weber, where the objective of the memory controller is not to be predictable. However, all other discussed memory controller lack predictable behavior and cannot provide a gross-to-net bandwidth translation.

This thesis is building on the work carried out by Åkesson in [Å05]. Åkesson is proposing an analytical analyzable memory controller which uses a static back-end schedule. The static back-end schedule is predetermined with the access parameters of the requestors. In the design phase a back-end schedule is calculated with the known mix of read and write accesses. The back-end schedule consists of a repeating access pattern. This allows translating the gross bandwidth to a net bandwidth. The drawback of this solution is that it has no flexibility of scheduling requests other than in the order described in the back-end schedule. This leads to a rather high worst-case latency. Furthermore, two (strict) traffic classes are proposed with high bandwidth (HB) and low latency (LL). Taken the work of Åkesson as starting point and identifying ways to improve the design leads to the proposed solution in this thesis. The work from Åkesson focuses on identifying the requirements for an analytical controller. The suggested solution is shown with an implementation in SystemC. In contrast the work for this thesis focuses on the design, implementation and verification of the memory controller at RTL level, which allows synthesis of the design.

Chapter 3

Design of the memory controller

This chapter describes the design process of the memory controller. It starts with a motivation of the problem set in Section 3.1. The system architecture is discussed in Section 3.2, and the requirements of the memory controller are identified in Section 3.3. Furthermore, it is explained how the memory controller can be used in an arbitrary system, and how it is used together with Æthereal. Section 3.4 discussed the design flow.

3.1 Motivation

Modern system architects face the problem that a flexible design is required for consumer products. When considering the market of mobile phones it appears that every year new product generations arise, with new features and functionalities. Comparing this with the design cycle of SoCs, where in contrast it takes years from the specification to the manufactured chip. However, in the specification phase it is hard to predict future features required from future markets. This is one of the reasons why more and more functionality is put into software and that upcoming SoC generations allow to be reconfigurable [GRG+05]. SoCs require more and more storage for software and this requires a bigger storage capacity. This makes it necessary to provide high volume storage for SoCs. The difference in the manufacturing process for high volume storage and for standard SoCs makes it necessary and economical to place the dynamic memory off-chip. DDR2 is the current standard for high volume memories. The expense of a higher pin count for going off-chip is overcome, with sharing the memory among several IP blocks. Furthermore, sharing a memory allows having common address spaces allowing memory mapped communication between processes. Almost every IP block in a SoC requires access to a storage device [HP03]. Hennessy and Patterson in [HP03] identify memory as the bottle neck of most modern computer systems since the memory is shared among different IP blocks. An efficient arbitration is necessary to provide the IP blocks with the required bandwidth and to provide an high overall system performance.

Figure 3.1 shows a system, where IP blocks (requestors) are connected to an external memory. The IP blocks are connected via an interconnect (like a bus, or a NoC) and a memory controller to the memory. The memory controller translates the requests from the requestors into accesses understood by the memory. Furthermore, the memory controller arbitrates between the requestors and allows having a shared memory.

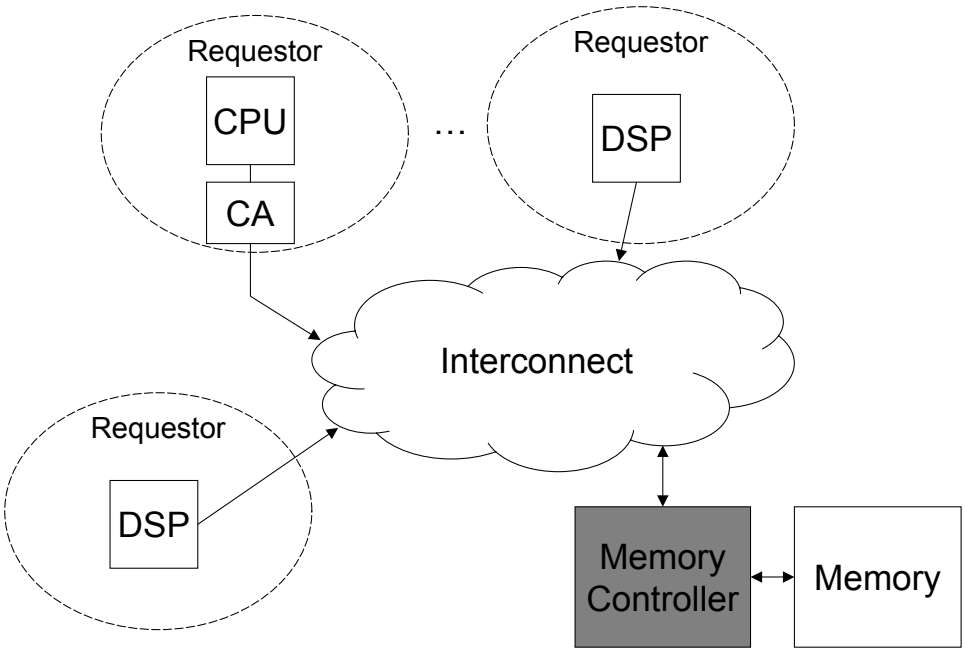


Figure 3.1: Three requestor are connected through an interconnect

The objective of this thesis is to design a memory controller for real-time systems, which is predictable. A prototype is designed for the use with Æthereal, which provides a predictable communication infrastructure. The system architecture is discussed in Section 3.2. The requirements are identified in Section 3.3.

3.2 System Architecture

This section identifies the requirements when connecting the designed memory controller to a system, where a predictable memory controller is required. Furthermore, it discusses the basic architecture of the design and states the requirements twice. In the first case the requirements are identified if the memory controller is used in an arbitrary system, and in the second case if the memory controller is used with Æthereal.

Figure 3.1 shows different IP blocks connected via a communication infrastructure to a memory controller, which in turn is connected to an external dynamic memory. Buffering is required to decouple the traffic between the requestors and the memory. Without this decoupling the requestor stalls until the access to the memory is granted. However, by introducing decoupling buffers this problem is overcome.

For a predictable memory controller design, it is necessary to implement a scheduler, which is predictable. The scheduler is responsible to provide the memory controller with the best fitting requestor according to the scheduling policy. The core memory controller translates the requests into accesses understood by the memory.

The memory controller is understood in this design process as a three component design with the following identified components:

- request and response buffering

- scheduler
- core memory controller

This three component view of the memory controller is illustrated in Figure 3.2. Summarizing this architecture, the incoming traffic is stored in the request buffer and thus decouples the traffic between requestor and memory. The scheduler arbitrates among the requestors and picks the best fitting requestor according to the scheduling policy. The core memory controller translates the requests to accesses understood by the memory.

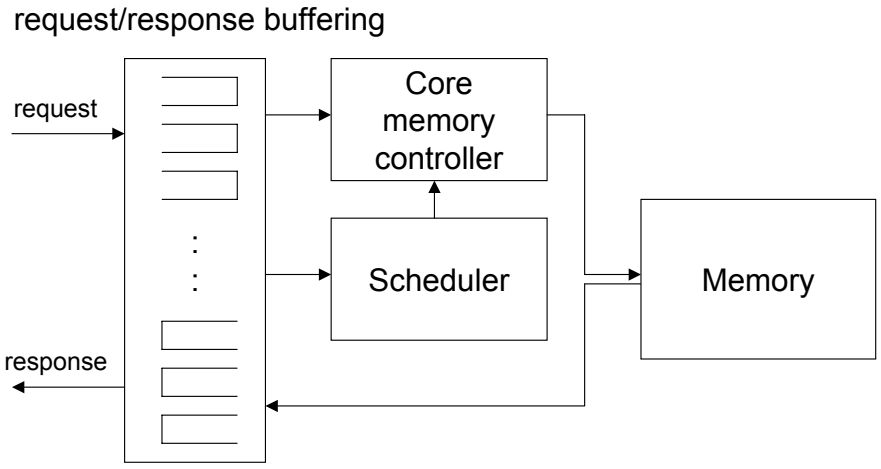


Figure 3.2: Modularization of the design

The above identified components are mandatory for the design of the memory controller. However, since the memory controller is (co-)designed for the use with Æthereal some simplifications can be made. For example the decoupling of the traffic is part of the network and it thus provides the required buffering. Section 3.2.2 discusses the integration steps, when using the memory controller with Æthereal. In contrast the next Section 3.2.1 discusses the required interface for the case when the memory controller is connected to an arbitrary system where a predictable memory controller is required.

3.2.1 Request and response buffering requirements

The request buffers decouple the traffic from the requestor to the memory controller. This prevents the requestor from stalling when accessing the memory. In the other direction response buffers are required to decouple the response path. The size of the request buffers calculates to twice the maximum packet size to reach a decoupling for periodic requestors, as stated in Section 2.4.4. The size of the response buffers is twice the maximum size of the response data for periodic requestors. Furthermore, the memory controller expects that for each requestor address and data is provided in parallel.

A write request (of a requestor) is considered as schedulable if a full request is present in the request buffers because a read request (of a requestor) is considered as schedulable if a full request is present, and if there is enough space in the response buffers so that the read response can be placed in the response buffers. If a request is seen as schedulable

only if a full request is received, then this mechanism prevents the memory controller from stalling during an access caused from outstanding data. However, below are the requirements of the request and response buffering for each requestor summarized and illustrated in Figure 3.3:

- request buffer, split up in:
  - address
  - data
- response buffer for the response data
- information
  - if a request is pending
  - if a request is read or write
  - if enough space is in the response buffers for the response data

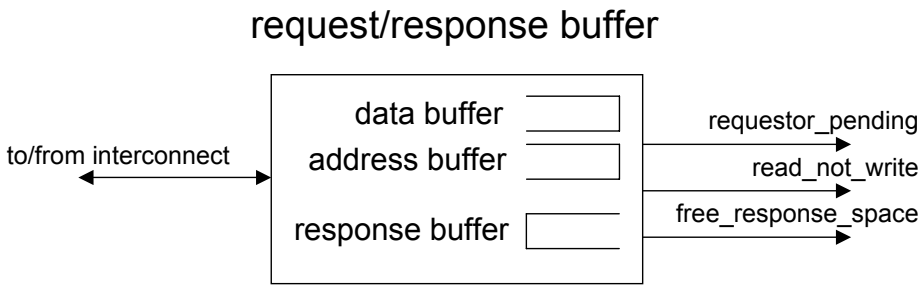


Figure 3.3: Request/response buffering for every requestor

The above described informations are required for every requestor. The core memory controller takes the data stored in the request buffer and transfers them in case of a write access to the memory. In the case of a read access the data is transfered by the memory controller from the memory to the response buffer. The scheduler requires the other above identified information's.

3.2.2 Integration into Æthereal

The objective of this thesis is to design a predictable memory controller which can be used with Æthereal. Æthereal provides an interconnection infrastructure for IP blocks [GRG+05]. The extension is to provide the functionality, of interconnecting IP blocks with an external dynamic memory. The design flow of Æthereal [GDG+05] provides the communication infrastructure and the goal is that automatically a memory controller is generated (at RTL level) with an interface to a DDR2 device, if a memory is present in the design. Figure 3.4 shows IP blocks which are connected via a NI to the network and to an off-chip memory. The off-chip memory is connected via a memory controller to the NI of the network.

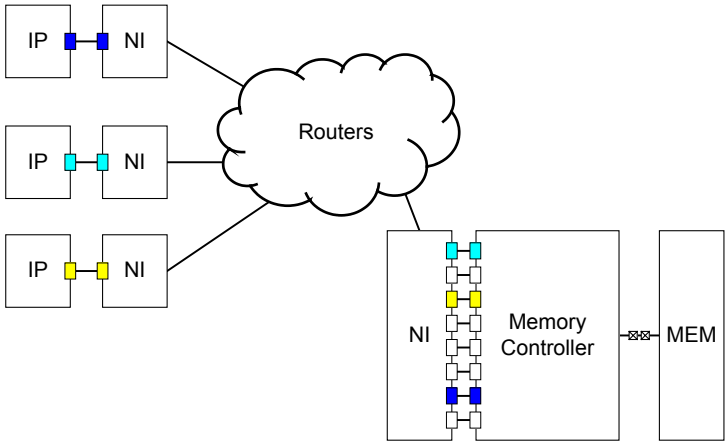


Figure 3.4: Goal of the design

As identified in the last section, to use the designed memory controller it is necessary to provide decoupling of communication and computation. However, decoupling the traffic is a key element of Æthereal and thus request and response buffering is provided in the NI. Furthermore, as identified in the last section every requestor requires to provide the information:

- if a request is pending
- if the request is read/write
- if there is enough response space (in case of a read)

The filling of the buffers in the NI provides the information about a pending request and the information if enough response space is in the response buffer. Therefore, it is necessary to adapt the NI kernel [RDGP+05] to extract this information. To generate the information of request pending, the filling of the request buffer is compared to the packet size of a request available in the first word of the transaction. If it is equal or bigger than it indicates that a request is pending. The free response space is traced for the response buffers. Enough space is present if at least one response packet can fit into the response buffer. The adoptions in the NI kernel are illustrated in Figure 3.5 with the ovals.

As identified in the last section, the memory controller expects the information address, data, and if the current request is of type read or write. Since the NI kernel does not provide the required information a simple shell is designed, which translates the Æthereal messages into address, data, and type of request. Figure 3.6 illustrates the NI shell for the memory controller.

The NI shell parses the incoming Æthereal message and sorts the information into the corresponding registers read/write, address. The message format of Æthereal is shown in [DRGR03].

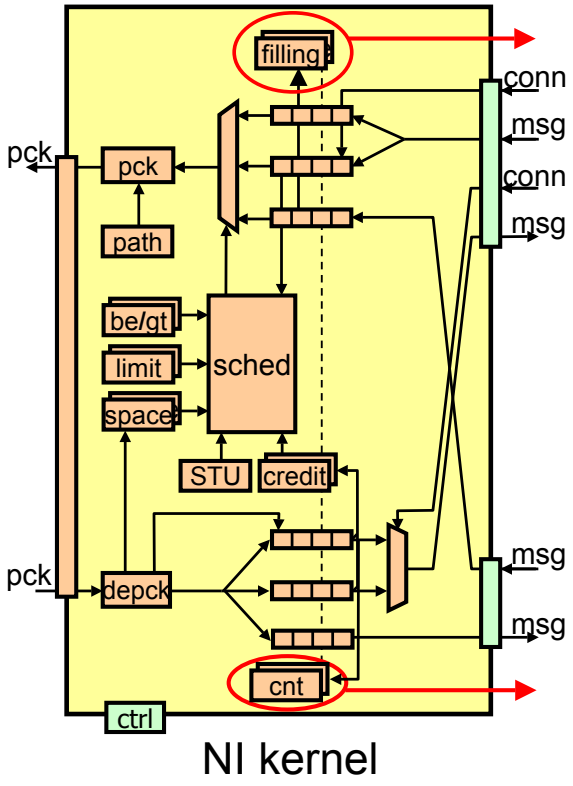


Figure 3.5: Modifications to the kernel

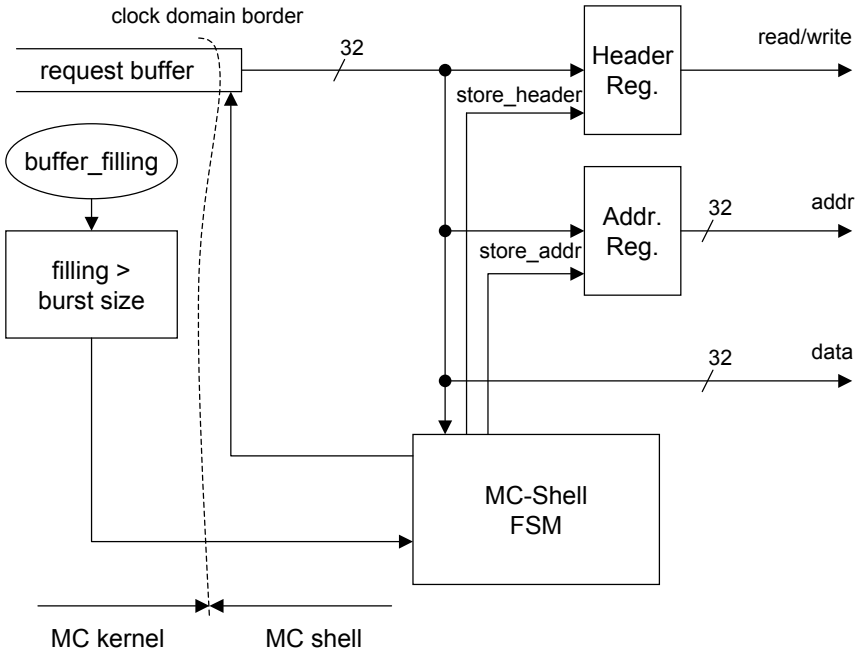


Figure 3.6: Incoming messages are parsed by the NI memory controller (MC) shell

3.3 Requirements

This section identifies the requirements for a predictable memory controller design. The requirements are derived by discussing the key elements of Æthereal (which provides a predictable communication infrastructure) and identifying there the requirements for a predictable memory controller design. However, the memory controller can be used in any system where a predictable memory controller is required when fulfilling the requirements of buffering mentioned in Section 3.2.1. The example of Æthereal is used to illustrate the requirements with an example. Further, in this section it is described how the requirements are fulfilled in the proposed design.

Æthereal provides hard real time guarantees and a predictable communication infrastructure. In the design phase it allows to specify bandwidth and latency for every IP block. The hardware for the communication infrastructure is generated automatically and the bandwidth and latency bounds are provided as specified, without the need of simulation together with other components [GRG+05]. Consult Section 2.4.1 for more details.

As stated in Section 2.4.1 a predictable system needs to be analyzable. This means to deliver an analyzable memory controller it is required to have an analyzable model of the dynamic memory. Section 2.1.3 showed that the efficiency of the memory depends on the traffic patterns and thus makes it difficult to provide an analyzable model of the memory. However, accessing the memory with a determined traffic pattern allows translating the bandwidth of the memory from a gross bandwidth to a net bandwidth and determines the efficiency of the memory. The gross bandwidth is the bandwidth specified in the data sheet of the memory, and defines the maximum throughput of the memory

without considering refresh, and other losses caused by read/write switches, etc. The net bandwidth is the effective available bandwidth to the requestors. The net bandwidth is a result of multiplying the memory efficiency with the gross bandwidth. The question is how to provide a traffic pattern, which makes the memory controller analyzable and efficient? The reasoning to this question and a possible solution is presented in Section 3.3.1. Note that an analyzable memory controller must provide a gross-to-net bandwidth translation.

As stated in Section 2.4.1 Æthereal allows to separate the design process of the functionality and the communication. The consequence is that there is decoupling between computation and communication, as motivated in Section 2.4.4. This allows every IP block to be analyzed individually. However, accessing the memory requires to first activate the bank, access a particular column in the row, and to precharge the row afterwards. Considering two requestors accessing two different banks and that the requests do not interfering with each other, then the accesses can be carried out consecutively without introducing idle states. However, considering two accesses going to the same bank, but not the same row, the second access has to be stalled until the first access has finished. The scheduling has to consider the time of precharging the row and wait the time until a new activate command is allowed to the same bank. This example shows that a memory access relies on the current state of the memory, which was caused by another requestor. A similar mechanism has to be considered when changing the directions between two accesses. In this case idle cycles on the data bus have to be inserted before the second requestor is able to access the memory [Ass04]. To provide an analyzable model for every requestor it is not acceptable to consider the current state of the memory, caused by prior accesses of different requestors. Therefore, a further requirement of the memory controller is to make the memory controller predictable providing a clear requestor isolation.

The list below summarizes the constraints and requirements, which are identified for the design of the memory controller:

- predictable minimum bandwidth and maximum latency
- gross-to-net bandwidth translation (predictable net bandwidth and latency)
- provide requestor isolation (net bandwidth and latency per requestor)
- efficient resource utilization

These constraints can be used for any predictable memory controller design. The example of Æthereal is used to motivate and illustrate these constraints. See Section 3.2 for the requirements to connect the memory controller to an arbitrary system.

3.3.1 Gross-to-net bandwidth translation

To design a predictable memory controller it is necessary to provide a gross-to-net bandwidth translation allowing the memory model to become analyzable, as stated in the previous section. However, as motivated in Section 2.1.3, the gross-to-net bandwidth translation relies on the efficiency of the memory, which in turn depends on the traffic pattern. Therefore, it is necessary to specify traffic patterns to make the memory controller predictable.

Nevertheless, considering efficient resource utilization as a design constraint, it is necessary to provide traffic patterns, which comply with the goal of high efficiency. As specified in Section 2.1.3 the most important sources of efficiency losses are the following:

- Refresh efficiency
- Data efficiency
- Command conflict efficiency
- Bank conflict efficiency
- Read/Write efficiency

However, the refresh efficiency does not rely on the traffic pattern. Therefore, it is not necessary to take refresh efficiency into account when evaluating how traffic patterns influence efficiency. Nevertheless, the data efficiency is traffic pattern dependent. A data efficiency loss is caused by a shifted data alignment and is considered a software problem and is usually not solved in the memory controller domain [HdCLE03, Å05]. Therefore, data alignment is not taken into account when analyzing traffic patterns.

As described in Section 2.1.3, the command conflict efficiency can be improved with a lower utilization of the command bus. This can be gained by using a larger burst size or by using read and write commands with an automatic precharge. The command conflict efficiency depends on the traffic pattern. Nevertheless, when using a larger burst size, the command conflict efficiency can reach 100%. This assumption about a 100% command conflict efficiency is reevaluated at the end of this section.

The two other types of efficiency losses are bank conflict efficiency and read/write efficiency. These two sources of efficiency loss have to be evaluated in more detail when analyzing traffic patterns. A loss of bank conflict efficiency happens if two accesses target different rows in the same bank within a tight time span. This leads to idle time on the data bus and thus decreases the efficiency. A 100% bank conflict efficiency can be reached if consecutive accesses target different banks in a certain way, so that no idle cycles are carried out on the data bus. Consider for a 4 bank memory the following access sequence: Bank 0, Bank 1, Bank 2, and Bank 3. This leads according to [Ass04] to a 100% bank conflict efficiency and the access is called hereinafter interleaved bank access pattern.

The read/write efficiency is determined by the number of switches from read to write and vice versa. The more switching between read and write, the lower the efficiency is. Without any switches the read/write efficiency would be 100%. This is only a fictive case because accesses would be carried out in only one direction, either reads or writes.

To analyze the traffic pattern, two extreme cases of traffic patterns are used for analysis. In the first (fictive) case, all requestors send read accesses to the memory and the accesses do not have bank conflicts. The second case contains read/write switches after every bank access and every two consecutive bank accesses conflict. The first case leads to an overall efficiency which is equal to the refresh efficiency and expressed by Equation (2.2) 98.27%. Calculating the second case leads to an efficiency below 25%. To achive a gross-to-net bandwidth translation with efficient resource utilization to the first case is the target. A 100% read/write efficiency could be reached for either read or write accesses but is not the intended functionality of the memory. Nevertheless, 100% read/write efficiency can be

reached if data is transferred between two refreshes in one direction and the direction is changed after the second refresh. This behavior puts high restrictions onto the requestor. At first, accesses are delivered in one direction and then into the other direction. If a requestor is posting an access at the wrong moment, while the current accesses are targeting the other direction then the worst-case time t_{REFI} ($7.8\mu s$) has to elapse until the requestor is allowed to be scheduled again. In this case, higher efficiency is traded for much higher latency. Under acceptable circumstances the target read/write efficiency of 100% is not reached.

Since a 100% read/write cannot be achieved, the proposed solutions targets a bank conflict efficiency of 100%. As already described above a 100% bank conflict efficiency is reached with an interleaved access pattern. For example in a DDR2-400 256Mb device which has 4 banks, the accesses are done in the following way: Bank 0, Bank 1, Bank 2, and Bank 3. Another constraint for this device is that a burst has to consist of 8 elements, to allow an interleaving bank access. A read access without bank access efficiency loss is shown in Figure 3.7. This type of access is named a basic read group by Åkesson in [Å05]. The read and write accesses are invoked with an automatic precharge to reduce possible command bus conflicts.

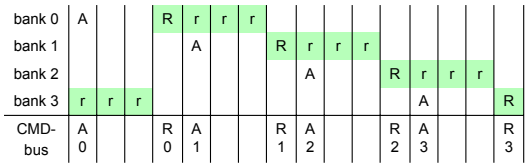


Figure 3.7: Basic read group; Activate (A), Read (R)

The basic read group consist of 16 cycles. After a start-up phase, data is transferred at every clock cycle. This leads to the target 100% bank access efficiency. An efficient write pattern is shown in Figure 3.8 and is defined as a basic write group according to [Å05].

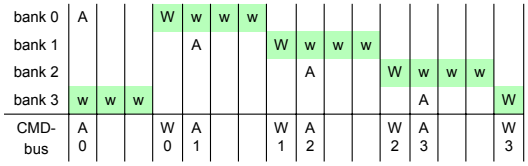


Figure 3.8: Basic write group; Activate (A), Write (W)

Analogously to the basic read group the basic write group consists of 16 cycles. After a start-up phase, data is transfered at every clock cycle, thus leading to the target 100% bank access efficiency. With the basic read/write group shown above, no loss is encountered due to bank access conflicts. However, introducing these access patterns assume that the accesses can be carried out in an interleaved way. There are several ways to achieve this. For example, by sorting the accesses from the requestors and combining them together to get an interleaved access pattern. However, this may result in a deadlock and has to be considered in the design. Another way is to assure that every request accesses as many

banks of the memory as required for an interleaved access pattern. This second approach leads to larger burst size, which in consequence leads to a higher latency.

With basic read/write groups mentioned above a read/write access looks like sketched in Figure 3.9. The NOPs in between the accesses are necessary to switch the data bus from read access to write access and vice versa and are called read/write cost respectively write/read cost. These costs are presented in Section 2.1.3

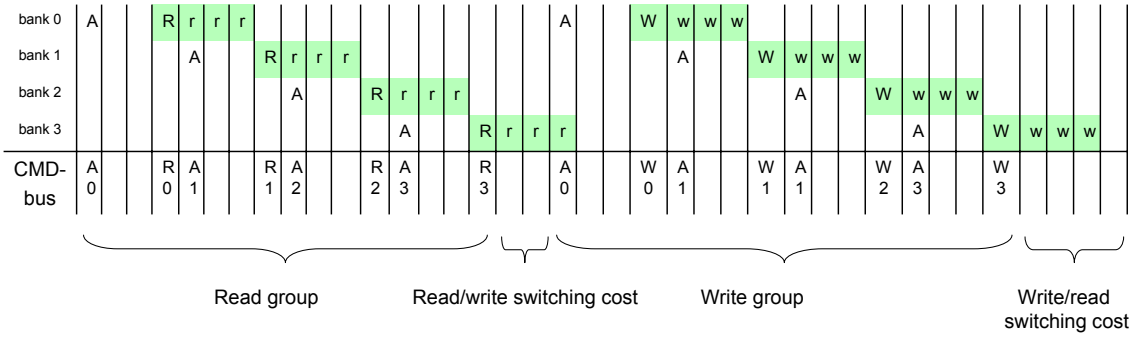


Figure 3.9: Read/write switch overhead; Activate (A), Read (R), Write (W)

Calculating the efficiency of a basic read group followed by a basic write group and repeating this traffic pattern leads to a read/write switching efficiency of 84.21%. This is the worst-case because NOPs have to be inserted between every read/write switch and causes idle time on the data bus. When arranging the read accesses and write accesses in consecutive order, like two read after an other, then the efficiency can be improved. Figure 3.10 shows a diagram with the memory read/write efficiency in respect to the number of consecutive read or write accesses.

The latency increases as a consequence of the gain in efficiency. Increasing the number of consecutive read and write accesses makes the data bus switch directions less often and increases the efficiency. However, it also increases the worst case latency of the requestor. The reason is that a requestor has to wait the time until a group, which just started accessing the other direction has finished. The bigger the group of consecutive accesses is, the higher the latency goes (Figure 3.11).

For the overall efficiency of the memory controller, the refresh efficiency has to be taken into account. The dotted line in Figure 3.10 show the combined refresh efficiency with the read/write efficiency and represents the overall efficiency. The dotted line in Figure 3.11 shows in contrast the higher latency when taking the refresh into account. The worst case latency increases because a refresh cycle has to be considered between a read/write switch before the analyzed access is carried out.

Analogously to the basic read/write groups, a basic refresh group is defined and illustrated in Figure 3.12 [Å05]. The basic refresh group consists of 24 cycles when the eight first cycles are NOPs during which all banks are precharged. The refresh time t_{RFC} (refresh-to-activate delay) has to elapse until the first activate command can be sent.

As shown above a traffic pattern exists, which allows a predictable and traffic independent gross-to-net bandwidth translation. The gross-to-net bandwidth translation relies on the read/write switch efficiency and on the refresh efficiency. Reevaluating the assumption of a 100% command conflict efficiency shows that the basic read/write groups in a

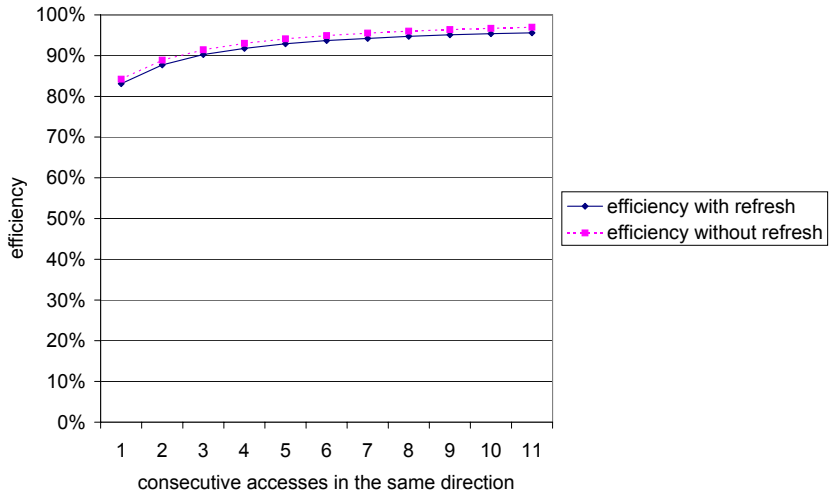


Figure 3.10: Read/write efficiency

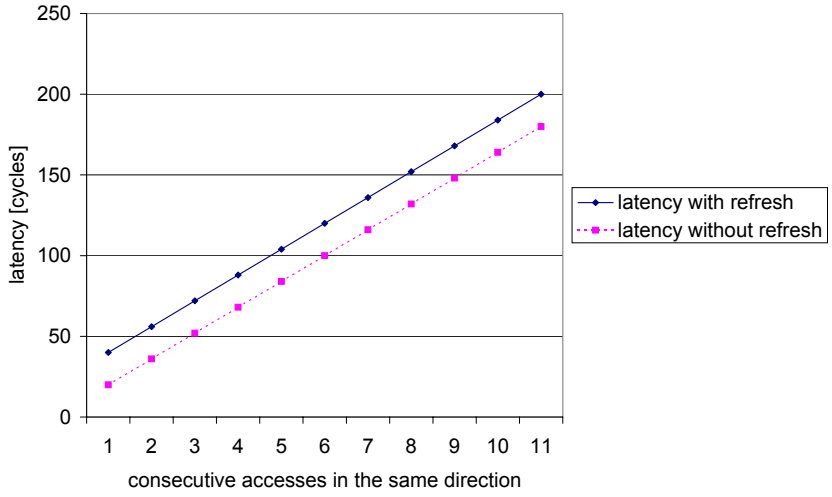


Figure 3.11: Latency

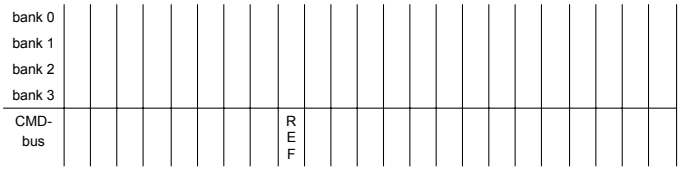


Figure 3.12: Basic refresh group; Refresh (REF)

DDR2-400 256Mb device requires a burst sizes of 8 elements and allows read/write commands with an autoprecharge. With the defined groups, this leads to the assumed 100% command conflict efficiency. Nevertheless, choosing the right traffic pattern depends on the constraints in the system design. There is a tradeoff between efficiency and latency. Higher efficiency leads to a higher latency. If a higher latency is acceptable for the system, then a higher resource utilization is reached.

3.3.2 Requestor isolation

To provide a gross-to-net bandwidth translation, it is essential that the accesses are carried out in basic groups. In Section 3.3.1 two ways to provide basic groups with interleaved bank access patterns are described. The first way is to reorder the requests and assemble a burst, which contains an interleaved bank access. By reordering the requests of different requestors the problem arises that an access of a requestor relies on the traffic pattern of another requestor. For example if all requestors want to access the same bank, then no interleaving bank access can be assembled. However, since requestor isolation is a design goal of the memory controller, this approach cannot be considered. The second approach is that the burst length of a request is long enough to allow an interleaved access pattern. In this case the burst size is calculated according to the following Equation 3.1.

$$burst\_size = \sharp elements\_per\_bank\_access \times \sharp banks \times memory\_data\_width \tag{3.1}$$

For a DDR2-400 256 Mb memory with a data width of 16 bits the burst size calculation results in a burst size of 64 bytes. Nevertheless, this packet size is rather big but allows providing the stated goals of the memory controller. In the current system design, the access granularity is growing, examples are VIPER II [GGRN04], WASABI [vEHJ+05].

An interleaved address map is used to translate the address of the burst access into physical address of the DDR2 memory in a way that always all banks are accessed. This is done according to Section 2.1.4. Nevertheless, to provide full requestor isolation the scheduler has to support requestor isolation as well. The issue of requestor isolation in the scheduler domain is discussed in Section 4.2.

3.3.3 Design conclusion

To design an analyzable memory controller it is essential to provide traffic patterns, which translates the bandwidth from gross-to-net. It is shown that a gross-to-net bandwidth translation is achieved by introducing basic groups. By either specifying the target memory latency or the memory efficiency the other parameter is derived. A system designer can

trade maximum latency for minimum efficiency as shown in Figure 3.11 and Figure 3.10. Further, it has been identified that for efficient and analyzable traffic patterns it is necessary to target a 100% bank conflict efficiency. Which is optained with an interleaved bank access pattern. The above mentioned design constraints allow the design of a predictable memory controller design. The open constraints are considered in the design domain of the scheduler and are discussed in Section 4.2.

3.4 Changes to the Æthereal design flow

The design flow described in [GDG+05] shows the steps of how an interconnection network is generated, which fulfills the requirement of a specified SoC. Starting from specifying the application requirements, such as minimum throughput, maximum latency, and the types of IP blocks, the design flow maps the IPs to a NI and generates a network infrastructure. The type of the IP blocks is characterized with the block size and the required interface, like AXI. However, an IP block of type memory with a DDR2 interface is not supported in the design flow [GDG+05].

By introducing an IP block of type memory, the design flow automatically generates a memory controller and an interface to a DDR2 memory at SystemC and RTL level. Moreover, the design flow adapts the NI for the requirements specified in Section 3.2.2, such as tracing the filling of the request and response buffers.

Since all components are designed as predictable components it is possible to calculate throughput and latency numbers without the necessity of simulation (for guaranteed connections). If the specified minimum throughput and maximum latency is reached, which is determined at design time, then the system works by construction [GDG+05].

3.5 Conclusion

A design of the memory controller is proposed which fulfills the identified requirements:

- predictable
- gross-to-net bandwidth translation
- efficient resource utilization
- requestor isolation

Analyzability is considered as requirement to get a memory controller that is predictable. The analyzability is gained with an interleaved traffic pattern, which provides next to predictability a requestor isolation at access level. Full requestor isolation is reached with the proposed scheduler, which provides requestor isolation at request level. Introducing basic groups allows a gross-to-net bandwidth translation. Furthermore, with introducing basic groups the efficiency of the memory is calculated at design time, and is above 80%. The scheduler, with a static-priority credit-based arbitration, solves the predictable arbitration between multiple requestors. Since the memory controller is predictable the minimum bandwidth bound and maximum latency bound of a memory access can be calculated analytically at design time.

Chapter 4

Implementation results and test strategy

This chapter describes the implementation and the test strategy of the proposed memory controller. In Section 4.1 the architecture of the memory controller is presented, with the following three main components: request/response buffering, the scheduler, and the core memory controller. To allow a predictable and scalable design a scheduler is proposed, which is scalable and analyzable. This issue is discussed in Section 4.2. In Section 4.3 a formula is presented for calculating the maximum latency bound for a request at design time. Section 4.4 gives an overview of the implementation process carried out for this project. Section 4.5 presents the synthesis result, where it is shown that the proposed scheduler is scalable, since the area of the scheduler grows linearly with the number of connected requestors. Section 4.6 discusses the test strategy of the memory controller, where the test is partitioned into a test for the core memory controller and a test for the scheduler.

4.1 Memory Controller architecture

As illustrated in Figure 3.2 the architecture of the memory controller is shown as a three components design, with the components: request/response buffering, scheduler, and a core memory controller. The request/response buffering was already discussed in Section 3.2.1. The requirements for the scheduler are identified in Section 4.1.1. The architecture of the core memory controller is discussed in Section 4.1.2. The architecture of the scheduler is presented in Section 4.2.2 after the arbitration mechanism is discussed in Section 4.2.

4.1.1 Scheduler requirements

The scheduler arbitrates among the different requestors and selects the best choice according a selected scheduling policy. For the design of a predictable memory controller it is essential that the scheduler is analyzable. The goal is to provide a scheduler, which allows next to analyzability an arbitration mechanism, which provides guarantees. These guarantees are specified with minimum bandwidth and maximum latency requirements.

Designing a scheduler with a general interface allows easily adapting the arbitration mechanism to different requirements. Keeping the scheduler generic allows separating the design process of the scheduler from the other components. Therefore, it is necessary to define when a requestor is considered as schedulable. As already stated in Section 3.2.1 a write request of a requestor is considered as schedulable if a full request is present in the request buffers. In contrast a read request of a requestor is considered as schedulable if a full request is present, and if there is enough space in the response buffers so that the read response can be placed in the response buffers. Therefore, it is necessary to know from every requestor:

- if a request is pending
- if a request is of type read or write
- if enough space is in the response buffers for the response data

With this input information the scheduler picks the requestor, which fits best to the scheduling policy and provides the core memory controller with this information. See Figure 4.1 for an illustration. The arbitration mechanism is discussed in Section 4.2.

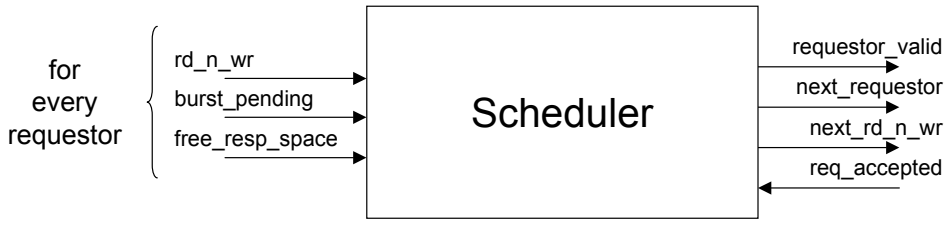


Figure 4.1: Scheduler standard interface

4.1.2 Core memory controller

According to the arbitration policy, the best fitting requestor is picked and passed over to the core memory controller. The core memory controller takes the requestor information and prepares the banks for an according access and routes the corresponding data through. The core memory controller consists of the following elements, as illustrated in Figure 4.2:

- main FSM
- address mapping
- command generation
- data path routing

The above mentioned components perform operations on different access granularities. The scheduler provides the information, which requestor is the next to pick. This information is provided at requestor level. The main FSM takes that information and generates

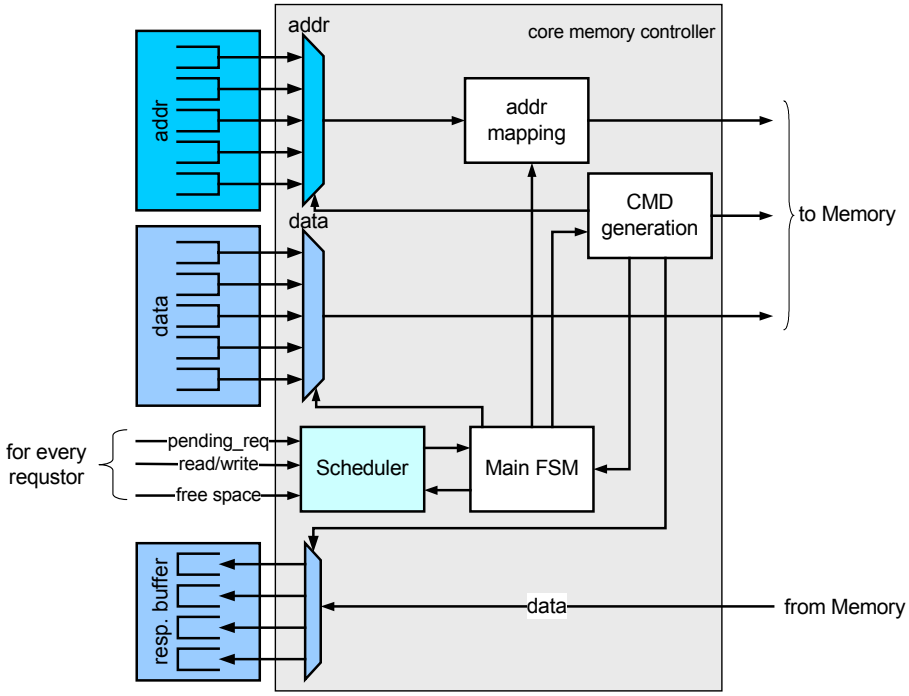


Figure 4.2: Architecture of the core memory controller

control information for the command generation at bank level. The command generation unit translates the bank accesses to DDR2 commands.

If a write request is schedulable (*burst_pending*) and the scheduler picks the request, then the scheduler indicates the chosen requestor to the main FSM, which accepts it with a handshake. The main FSM splits the request into accesses to the particular banks. Further, the main FSM steers the address mapping unit and the command generation. The address mapping unit translates the incoming logical address into a physical memory address. The command generation invokes the DDR2 commands and fetches and routes the write data to the memory.

A read request looks similar, if a read request is schedulable (*burst_pending* and *free_response_space*) and the scheduler chooses the request then the scheduler tells the main FSM in the same way as in the write case the chosen requestor. The difference is in the data handling from the command generation. The data is fetched from the memory and written to the response buffers.

The components of the core memory controller are discussed in the next subsections in more detail.

Main FSM

The main FSM breaks the access granularity from requestor level down to bank access level. The implementation focuses on DDR2 memories with 4 banks. The scheduler provides the main FSM with the information about the best fitting requestor, according to the arbitration policy. The main FSM accepts with a handshake the requestor, and

prepares the access. Therefore, it routes the target address of the selected requestor to the address mapping unit. Furthermore, the main FSM indicates to the command generation unit, which bank to access next. The command generation unit accepts with a handshake and starts to send the according commands to the memory. The interface of the main FSM is illustrated in Figure 4.3

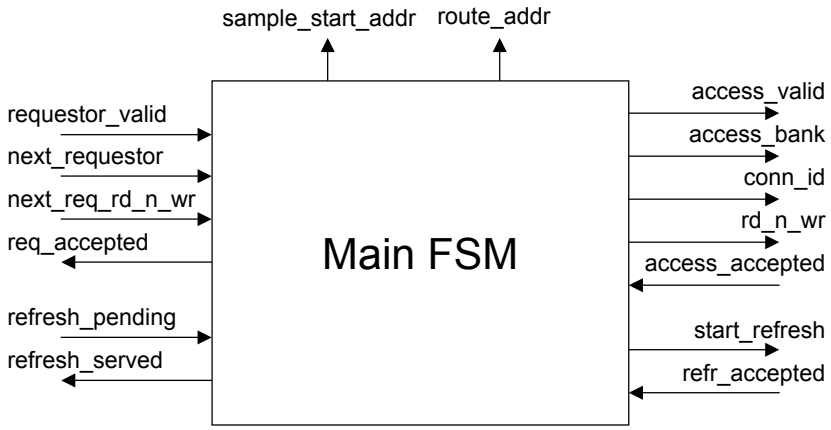


Figure 4.3: Interface of the main FSM

The signals at the left side of the interface are provided from the scheduler. The signals necessary for the refresh are also shown additionally to the already identified scheduler signals. However, the refresh can be interpreted as the best fitting requestor, which is necessary to keep the information in the memory. The signals on the right side of the interface are connected to the command generation unit. The signals at the topside of the interface are connected to the address mapping unit.

The state machine of the main FSM is illustrated in Figure 4.4. In the state **idle** the main FSM accepts from the scheduler the best fitting requestor and splits the request into bank access (states: **serve_bank0** – **serve_bank3**). In case of a pending refresh the main FSM indicates the command generation unit the scheduled refresh with the signal *start_refresh* (state: **serve_refresh**).

Address mapping

The address mapping unit translates the logical address provided in the requests to physical DDR2 memory address. The address map targets an interleaved access pattern, according to Section 2.1.4. The interface is illustrated in Figure 4.5.

The incoming logical address (*start_addr*) is converted into row and column address. Table 4.1 shows the relationship between the logical address and the physical address. However, this is only done if the main FSM indicates that a new requestor is picked and therefore the logical address has to be sampled (*sample_start_addr*). The main FSM is aware of the access pattern and provides the command generation with this information. Therefore, bits 3 and 4 (of the logical address) are not used for the address translation, which would represent the bank information otherwise.

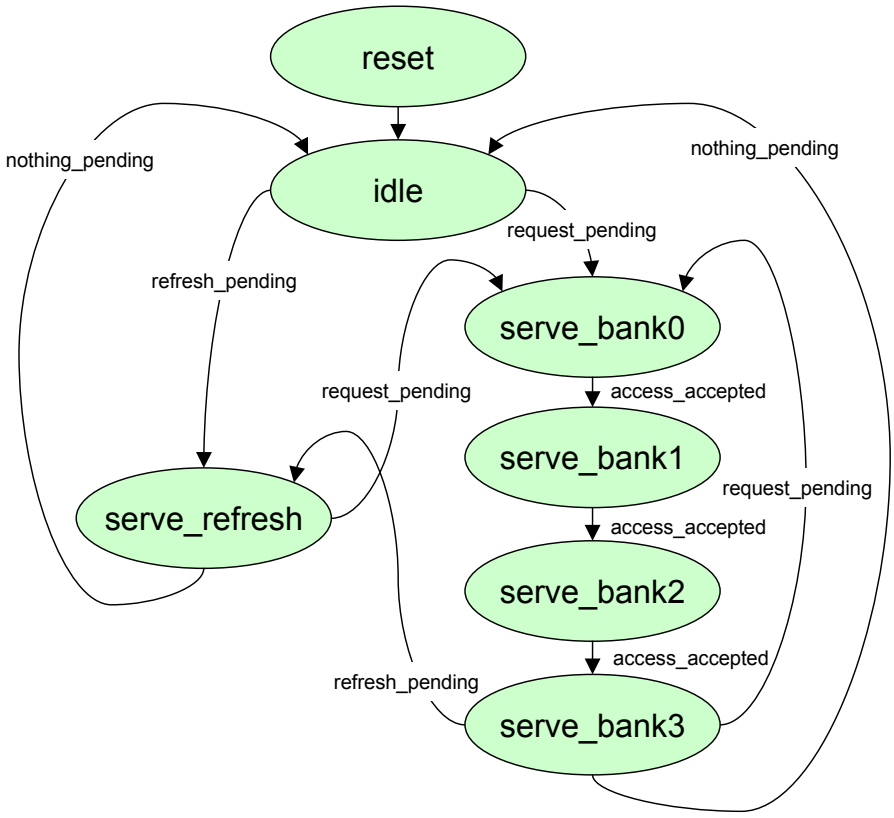


Figure 4.4: State digram of the main FSM

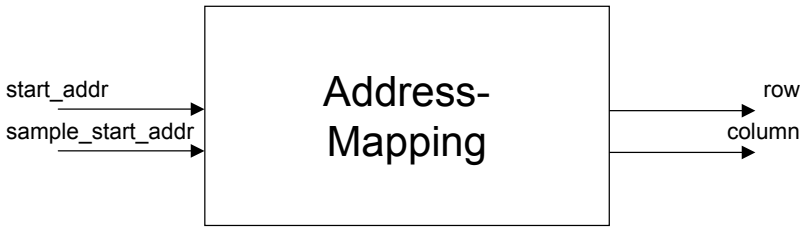


Figure 4.5: Address mapping module

	Logical address bits
Row	24 downto 12
Column	11 downto 5 and 2 downto 0

Table 4.1: Address mapping table

Command generation

The command generation translates the target bank accesses provided by the main FSM to DDR2 commands. Furthermore, it routes the data on the data path and fetches the data from the request buffers (write access) or writes the data to the response buffers (read access). The interface is illustrated in Figure 4.6.

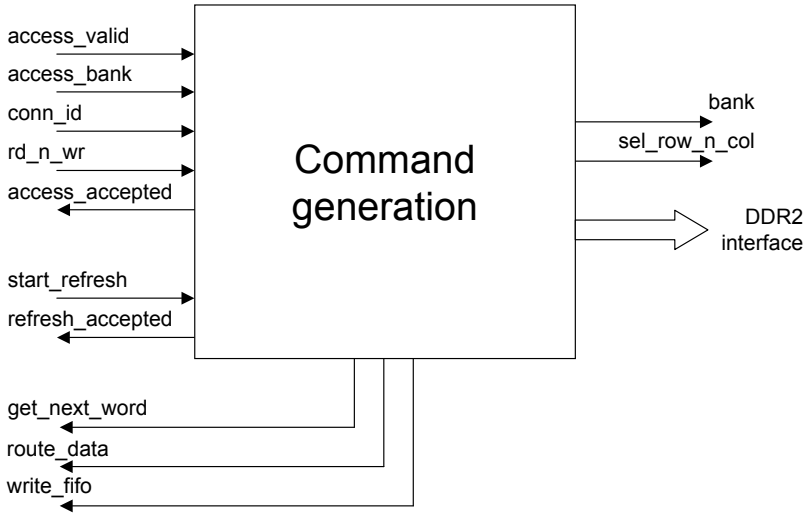


Figure 4.6: Command generation module

The signals on the left side of the interface are provided by the main FSM. A handshake is used to accept the pending accesses. The signals at the bottom of the interface are connected to the request and response buffers. In the case of a write access the data is routed (*route_data*) and fetched (*get_next_word*) from the request buffers. In the case of a read the data is written to the response buffers (*write_fifo*) after the data is read from the memory. The signals at the right hand side of the interface are connected to the DDR2 memory. The signal *bank* together with the output signals from the address mapping unit represent the physical address of a memory location. The signal *sel_row_n_col* selects either the row or the column address and puts it onto the address bus, since the address bus is multiplexed in a DDR2 memory.

After the reset the memory gets initialized. This is done according to the JEDEC standard specified in [Ass04]. After the initialization phase the memory becomes **idle** and waits for bank accesses or a refresh indication provided by the main FSM. In case of a refresh indication a refresh command is sent to the memory and the time has to elapse until the memory is able to accept new commands. If no further access is indicated the command generation goes into **idle** mode. In case of a new bank access the command generation unit

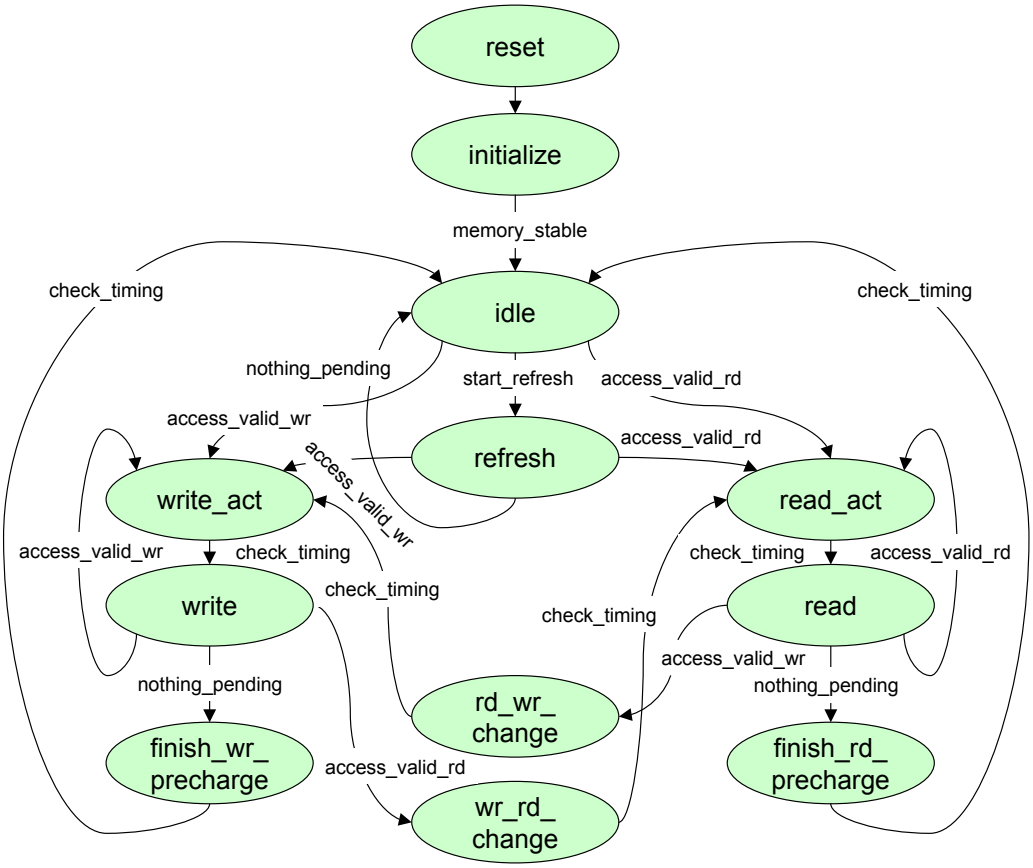


Figure 4.7: State diagram of the command generation unit

sends an activate command to the memory, and sends consecutively the access (either read or write) according to the timing requirement. If the next access has the same direction as the current access then the interleaved bank access pattern is proceeded. In case of a direction change the FSM waits for the required idle cycles to change the direction of the data bus (states: **rd_wr_change** or **wr_rd_change**). However, if no new access is waiting then the command generation unit gets back into the **idle** state with the intermediated state (**finish_wr_precharge** or **finish_rd_precharge**) waiting until the memory is able to accept new commands.

4.2 Arbitration mechanism

This section presents the used arbitration mechanism. For the design of a predictable memory controller it is mandatory to integrate an analyzable scheduling algorithm. Åkesson et. al. presents in [ASGR06] an analyzable arbiter with low latency bounds and which provides fine-grained requestor allocation. This arbitration algorithm is shown and analyzed in [ASGR06] and is summarized in the next sections. Furthermore, in Section 4.2.2 it is shown how the arbiter is implemented at RTL level.

4.2.1 Analysis of the arbitration

To provide a predictable memory controller it is mandatory to fulfill the goals identified in Section 3.3. Two points are listed below, and are considered as scheduling problems, and are solved by the chosen scheduler:

- provide requestor isolation
- predictable arbitration between multiple requestors

Focusing on this two points Åkesson et. al. suggests in [ASGR06] a credit-controlled static-priority scheduler, which provides requestor isolation and predictable arbitration between requestors. The credit-controlled arbiter, allows controlling the accesses of the requestor, corresponding to the requested bandwidth requirements. The proposed arbitration policy has a static-priority scheduler, which provides a service contract guaranteeing a minimal bandwidth and a maximum latency. The arbiter is composed of a credit-based rate-controller followed by a static-priority scheduler. See Figure 4.8 for an illustration. The regulator provides bandwidth controls and isolates requestors. The scheduler is responsible for the latency control of the requestors. The following paragraphs summarize the content of [ASGR06].

Requestor specification

The requirements of a requestor are specified with the required latency and the required bandwidth. The required bandwidth corresponds to the load ρ in the (σ, ρ) traffic model. The behavior of the requestor is characterized with the load ρ and the burstiness σ in the (σ, ρ) traffic model. The load ρ is a fraction of the required bandwidth divided by the available bandwidth in the system. To isolate requestors that behave according their traffic specification from those who misbehave (asking for more bandwidth than specified) a credit-based rate-regulator is used. The bandwidth ratio is specified with the two

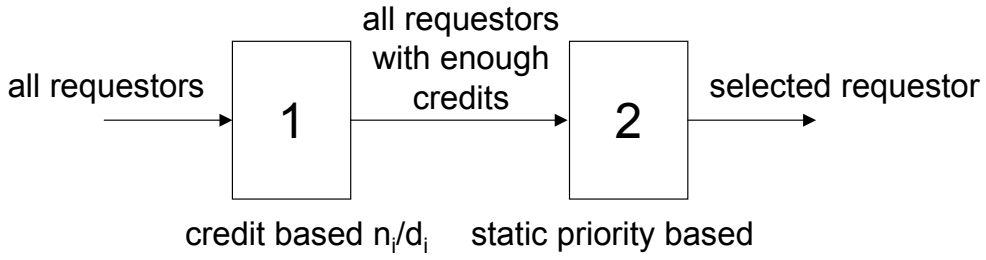


Figure 4.8: Components of the arbiter; credit-based rate-controller (1) and static-priority scheduler (2)

parameters: numerator and denominator. To allow an implementation of the scheduler, the parameter numerator and denominator are represented with a limited amount of bits. Equation (4.1) shows the relationship between the parameters numerator and denominator with the calculated bandwidth-ratio ρ_{calc}.

$$\rho_{calc} = \frac{numerator}{denominator} \tag{4.1}$$

Equation (4.2) specifies how credits are built up in the credit-based rate-regulator. At every arbitration cycle the credits are updated. A requestor is only allowed to be scheduled when enough credits are available, which means that the amount of credits has to be bigger than the value $denominator - numerator$ for a fixed request size of one. The time when enough credits are available is the time when the request is allowed to be scheduled.

$$credit(t) = \begin{cases} credit(t-1) + numerator - denominator & \text{if requestor is scheduled} \\ credit(t-1) + numerator & \text{if requestor is not scheduled} \end{cases} \tag{4.2}$$

However, this means that the scheduler is non-work-conserving, because even if there are requests pending the system is idle when no requestor has enough credits to be scheduled. By spending the slack a requestor is allowed to be scheduled even without credits thus achieving a better average latency.

To achieve a specified latency the requestors are assigned with a priority, corresponding to the latency requirements of other requestors and the own requestor [ASGR06]. [ASGR06] states the corresponding algorithms to derive the parameters numerator, denominator, and the priority for the scheduler for each requestor.

Results

The arbitration policy described in [ASGR06], provides maximum latency guarantees and minimal bandwidth guarantees to the requestors. The latency consists of the time required to build up credits and the interference time. The interference time is the worst-case time until an access is scheduled after the requestor has built up enough credits. This means that the requestor fulfills all requirements, but has to wait for other requestors with higher priority until it gets scheduled. This interference latency is defined with $\hat{\theta}_i^{>}$ [ASGR06].

4.2.2 Scheduler architecture

In the last section an arbitration mechanism is discussed, which has a credit-based arbitration with static priorities. The interface of the scheduler fits into the generic specification of the scheduler in Section 4.1.1. The interface of the scheduler is illustrated in Figure 4.1. One of the generic parameters is used to describe the number of requestors. Another generic parameter describes the used bits for the representation of the credits. By spending more bits for the credits the granularity of the allocation is finer, which results in less over-allocation and hence higher fairness of the scheduler. The trade-off with area is tunable since the number of bits is selected at design time.

The architecture proposed for the scheduler is shown in Figure 4.9. The required credits are stored in a register bank, which can be configured through a MMIO like interface. The credit registers used for every requestor in the design are the following:

- numerator
- denominator
- current credits
- maximum credits

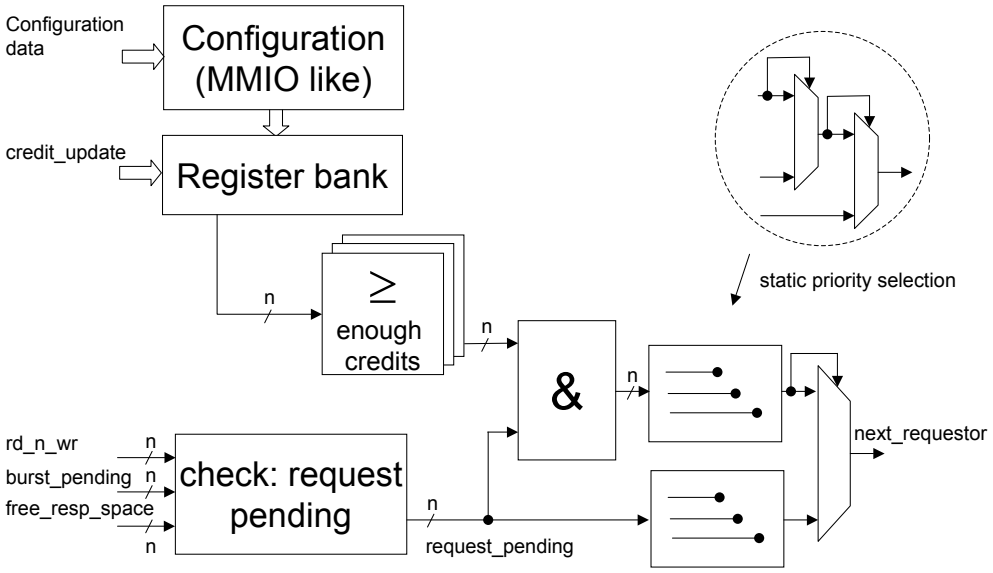


Figure 4.9: Scheduler architecture

Together *numerator* and *denominator* represent the specified bandwidth for one particular requestor. The current credits are stored in the register *current_credits*. In the arbitration mechanism a maximum bound on the number of credits is calculated, which is stored in the register *maximum_credits*. At every arbitration cycle the signal *credit_update* updates the credits according to Equation (4.2). The maximum credit bound is used to prevent requestors, which do not use their credits (not enough requests) to build up too many credits and violate the latency and bandwidth requirements of the

other requestors. This credit bound allows, furthermore, limiting the credit register to a certain number of bits.

With the input signals *rd_n_wr*, *burst_pending*, and *free_response_space* an internal signal (*request_schedulable*) is generated, which indicates if a request is schedulable, according to Section 4.1.1. However, the request is just scheduled (in the first consideration) if enough credits are built up. If a requestor has enough credits and the request is schedulable (this test is indicated with an ampersand in Figure 4.9) then the scheduler selects the requestor with the highest priority, which fulfills the requirements of enough credits and that it is schedulable.

The priority-based scheduler is designed as cascaded multiplexer tree, where the highest priority requestor is forwarded (and scheduled) if it has enough credits and it is schedulable. If the highest priority requestor does not fulfill these requirements then the requestor with the second highest priority is checked against the requestor with the third highest priority and so on.

However, if there is no schedulable requestor, which has enough credits then the memory controller would stall, even when there are schedulable requestors. To prevent from this situation work conservation is used and the highest priority requestor is scheduled, which is schedulable (but has not enough credits). This is indicated in the figure with the second (lower) static-priority selection box. The multiplexer at the end selects the highest priority schedulable requestor with enough credits if one found, otherwise it selects the highest priority schedulable requestor. This work conservation mechanism comes with an increased area efficiency trade-off.

Nevertheless, another way of implementing the selected scheduler is to design the algorithm in software running on a general-purpose processor. However, a problem of having the algorithm implemented in software is a speed issue. In hardware the credit check (if enough credits) is done for all requestors in parallel, but this cannot be done in software and therefore this comparisons have to be done after another. Furthermore, there is a different arbitration speed between the requestors. The scheduler has to check for high priority requestors if the requestor has enough credits and if no requestor with a higher priority is schedulable. However, for the lowest priority requestors all test for the other requestors have to be completed before the lowest priority requestor is analyzed. Since the scheduler is targeted for high speed arbitration (more than 200MHz) software is not a possible solution.

However, another way to implement the scheduling mechanism is to calculate the behavior of the scheduling algorithm beforehand (based on the specified bandwidth and latency requirements), and integrate this in a hardware look-up-table (LUT). The LUT selects among the schedulable requstors, affected by the current amount of credits, the best fitting requestor. For every use case this LUT has to be calculated during the (re)configuration phase thus making (re)configuration slower and more difficult.

4.2.3 Static-priority reordering

The arbitration mechanism is designed for static-priority allocation. However, to allow a reconfigurable design a mechanism is implemented, which allows changing the priorities of the requestor to be able to program the arbiter for a different set of requestors (for example a different use case). A naive way of doing this is by introducing a switch in front

of the arbiter. After the arbiter selected a requestor the look up table (LUT) translates the choice back so that the right requestor is selected by the memory controller and makes a priority buffer translation. Figure 4.10 illustrates the reconfiguration unit and the arbiter together with the request/response buffering of the memory controller.

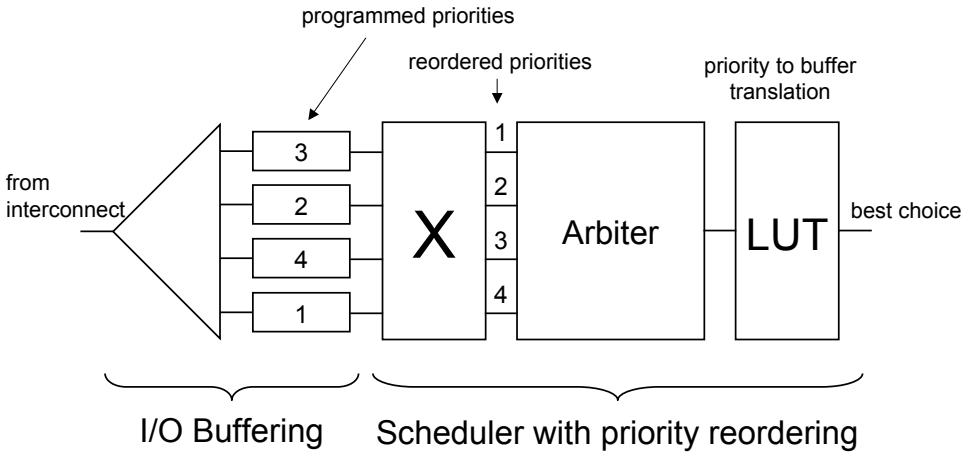


Figure 4.10: Priority reordering with a switch

The switch reorders the requestors and routes the signal *request_schedulable* according to the priority of the requestor to the arbiter. The buffers in this drawing show that the signal *request_schedulable* belongs to one request buffer. After a requestor is chosen the core memory controller fetches the data from this buffer.

However, implementing the switch affects the area negatively. The area of the area of the switch is expected to grow quadratically with the number of requestors and thus does not allow a scalable scheduler.

The proposed way to overcome this problem is to remove the switch, but still allow reordering the priorities. A switch is saved if the buffers are organized according their priorities. Figure 4.11 illustrates the reordering of the buffers, which removes the switch and the LUT. This solution allows a scalable scheduler and is therefore used in the design.

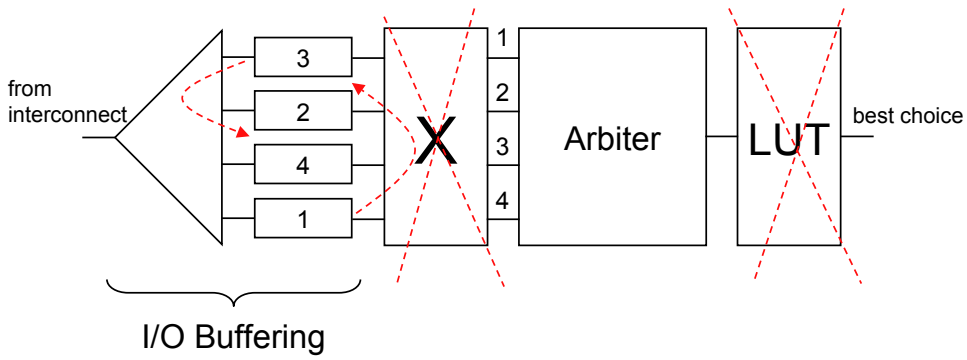


Figure 4.11: Priority reordering by interchanging the buffers

When using Æthereal as interconntect this solution has another advantage. The packets coming from the network are sorted into the different buffers according their connection ID. By assigning connections to (physical) buffers according the priorities of the requestors the reordering of the buffers comes for free without the expense of introducing a switch.

4.3 Worst-case latency of a memory request

The memory controller together with the proposed scheduler is designed to be analyzable. This allows calculating the maximum latency required for an access to the memory. The worst-case latency is calculated for a read access in Equation 4.4, and for a write access in Equation 4.3. The latency is measured as the time how long it takes when a request gets schedulable to the point in time when the last word is written to the memory (write access) or the point in time where the last word is placed in the response buffers (read access).

$$\begin{aligned} wc\_latency_{write} &= 18 * \hat{\theta}_i^{>} + write\_access + precharge_{write} + refresh \\ &= 18 * \hat{\theta}_i^{>} + 25 + 10 + 20 \\ &= 18 * \hat{\theta}_i^{>} + 55 \end{aligned} \tag{4.3}$$

$$\begin{aligned} wc\_latency_{read} &= 18 * \hat{\theta}_i^{>} + read\_access + precharge_{write} + refresh \\ &= 18 * \hat{\theta}_i^{>} + 27 + 10 + 20 \\ &= 18 * \hat{\theta}_i^{>} + 57 \end{aligned} \tag{4.4}$$

Both Equations look similar and contain the interference latency ($\hat{\theta}_i^{>}$) of the arbitration mechanism. The interference latency is given in arbitration cycles. However, one arbitration cycle is the average time for one access. The average time for one access is 18 clock cycles. This is calculated for a DDR2-400 device and the case that the requests change the direction every time. The *read_access* time starts when the request is picked and ends when the last word is written to the response buffers. The *write_access* time starts when the request is picked and stops when the last word is written to the memory. For the worst-case of both latencies the time of a refresh has to be added. However, the refresh requires that all banks are precharged. The precharge time is longer if a write access was the last access before the refresh compared to a read access. Therefore, the precharge time for a write access is added in the worst-case calculation. Comparing the two Equations results that the worst-case latency for a read access is higher than for a write access. The minimum bandwidth of a requestor is guaranteed with the credit mechanism.

4.4 Design Flow

The memory controller is implemented according to the specification in Section 4.1 and is implemented in VHDL. However, a memory controller is only generated if a memory is specified as component in the application requirements of the Æthereal design flow.

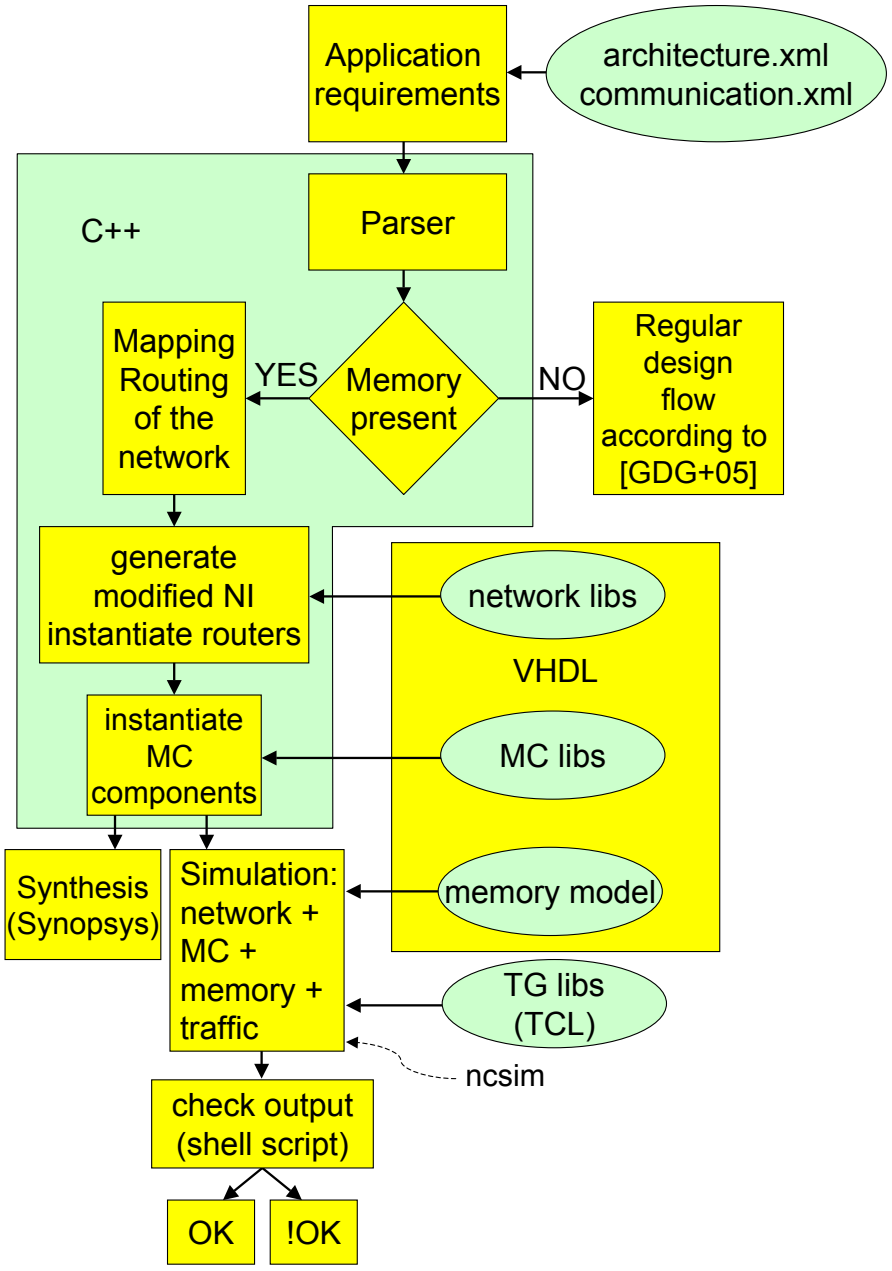


Figure 4.12: Modified design flow for generating a memory controller

Figure 4.12 illustrates the process from specifying the components to generating the system depending on the presence of a memory.

The application requirements are stated in the XML files architecture.xml and communication.xml and specify the behavior of the components (requestors), like the minimum bandwidth and maximum latency of a requestor [GDG+05]. The design flow parses the XML files and extracts the information if a memory is present in the system. If not then the regular design flow according to [GDG+05] is invoked. If a memory is present then the design flow generates the network infrastructure by instantiating components from the network libraries designed in VHDL. The network interfaces for the connected memory controller are modified according to the buffering requirements shown in Section 3.2.2. The routers are instantiated in the same way as described in [GDG+05]. As next step the memory controller components are instantiated from the memory controller library. The memory controller is designed in VHDL as a generic library thus allowing a memory controller for an arbitrary number of requestors. Furthermore, an interface to a DDR2 memory is created, which allows to connect this type of memory. The above described parts (shown in the figure as shadowed box) are designed in C++.

The next paragraph shows a code snippet from the architecture.xml file where the memory controller is specified. The connected requestors are characterized as ports, where the information about the protocol and the data width is specified.

```
<ip id="memory" type="MemoryController">
  <port id="p1" type="Target">
    <portparam>
      <protocol protocol="MMBD"></protocol>
      <width width="64bit"></width>
    </portparam>
  </port>
  ...
</ip>
```

The memory is described as an IP block with the ID memory and of type MemoryController. Type MemoryController is used to point out, that the memory controller is instantiated by the flow and that the interface to a DDR2 memory is generated. Every requestor is connected to the memory controller via a port. The ports are of type target, since the requestors always invoke the communication between requestor and memory. With the protocol parameters (portparam) the protocol is specified. In this example the protocol is Memory Mapped Block Data (MMBD) and a data width of 64-bits is used.

For the synthesis Synopsys is used and the design is synthesized for a CMOS12 process. The synthesis results are discussed in Section 4.5.

The design is simulated with NCsim from Cadence. A memory model is used to verify the timing of the memory controller. Different test strategies are used to verify the behavior of the system. One test strategy is to simulate the network together with the memory controller, and the memory model. During the simulation traffic generators (TG) generate random traffic for the network and stimulate the memory controller. A shell script compares the simulation output with the expected output and reports as result if the simulation was OK or FAILED (!OK). The test strategies are described in more detail in Section 4.6.

4.5 Synthesis results

This section presents the synthesis results in terms of area and speed. The area of the scheduler depends on the number of connected requestors and on the number of bits used for the credit registers. Therefore, a comparison of area and the number of ports (where the requestors are connected to) and the number of bits used for the credit registers is given. In the scheduler design the switch was identifies as limiting factor for a scalable system. A solution without the switch has been described in Section 4.2.3 thus allowing a scalable scheduler implementation.

4.5.1 Memory Controller

The memory controller is synthesized for CMOS12 and a target speed of 200 MHz. The memory controller utilizes an area of $0.042mm^2$, when synthesized for 6 requestors. The scheduler consumes almost half of the area. When varying the number of requestors, the main influence (over 40% for 8 connected requestors) in terms of area comes from the scheduler. Buffering is not included in the area numbers, as this is part of the NIs. The main FSM, command generation, and the address mapping have a constant size. As shown in Figure 4.9 more registers and a larger multiplexer tree is requires as the number of requestors increase. Therefore, in the next section the scheduler is analyzed in more detail.

4.5.2 Scheduler

This Section describes the synthesized area required for the two scheduler designs proposed in Section 4.2.2. The area of the scheduler depends on the number of requestors. This section presents a comparison of area in respect to the number of requestors connected. Furthermore, this section presents a solution to a scalable implementation of the proposed scheduler. Two implementations of the scheduler exist. The first implementation includes a switch matrix, which allows reconfiguration of the priorities in the scheduler. The second implementation requires that the priorities are given 1-1 with the desired connection. This means that connecting a requestor to Port 1 of the memory controller results in the highest priority for this requestor. A requestor connected to Port 2 gets the next highest priority and so on. Priority reordering is discussed in more detail in Section 4.2.3. The width of all credit registers is constant with 6 bits for the synthesis, if not indicated otherwise.

Scheduler with priority-switch

This scheduler implementation (see Figure 4.10 for the architecture) allows to reorder the priorities of the requestors during run time. To allow this a NxN switch matrix is implemented in the scheduler. The complexity of the switch is $O(n) = n^2$ and determines the area demand of the switch. It is expected that the area grows exponential. Figure 4.13 shows the synthesized area of the scheduler with respect to the number of input ports, which correspond to the requestors.

This figure shows that with an increasing amount of ports the area of the scheduler grows quadratically. This means that the scheduler is not scalable as required. The next paragraph shows the implementation of a scalable scheduler.

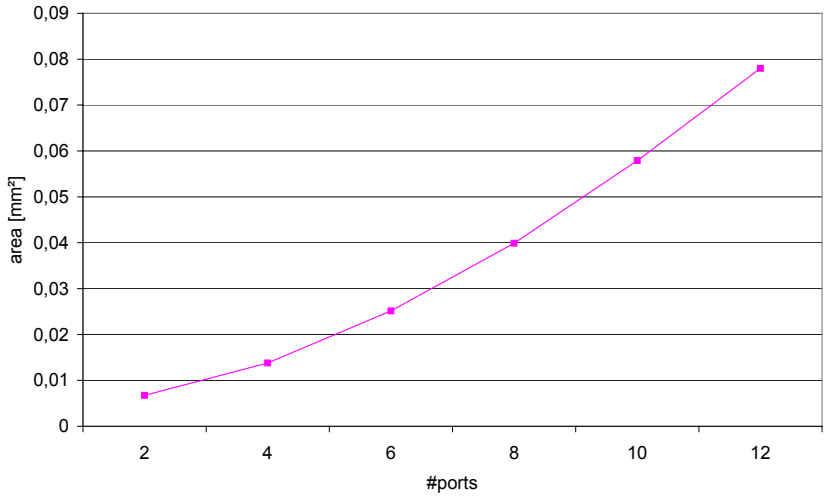


Figure 4.13: Area of the scheduler with priority reordering

Scheduler without priority-switch

This scheduler implementation (see Figure 4.11 for the architecture) assumes that the requestors are connected according to their priority. Therefore, no switch is necessary to reorder the priorities. However, as described in Section 4.2.3 priority-reordering is done at the system level by interchanging the queues. Figure 4.14 shows the synthesized area of the scheduler in respect to the number of input ports.

The diagram shows that the area of the scheduler grows linearly with the number of requestors. The curve is not exactly linear because at some points extra bits need to be introduced to address the different amount of ports. For example, four ports are addressed with two bits. In the next step one extra bit is introduced to address six ports. However, to address eight ports no extra address bit has to be introduced and causes a bend in the curve. The same reasoning is valid for the step from ten ports to 12 ports and further to 14 ports.

The number of bits used in the credit registers is another parameter, which is generic and can be varied in the instantiation of the scheduler. The more bits used the finner the granularity of the bandwidth allocation is. This causes that with fewer bits used overallocation is introduced to represent the required bandwidth from a requestor [ASGR06]. However, having a finer granularity means that hardly any overallocation is made but results in a bigger area demand on the chip. Figure 4.15 shows the effect of different amounts of credit bits used. For this experiment the number of ports is constant with 6 ports. This figure shows a linear behavior between area and the credit-register-bits.

Varying the two parameters number of ports and number of credit-register-bits shows a linear relationship between the area and each parameter. This allows to scale the scheduler

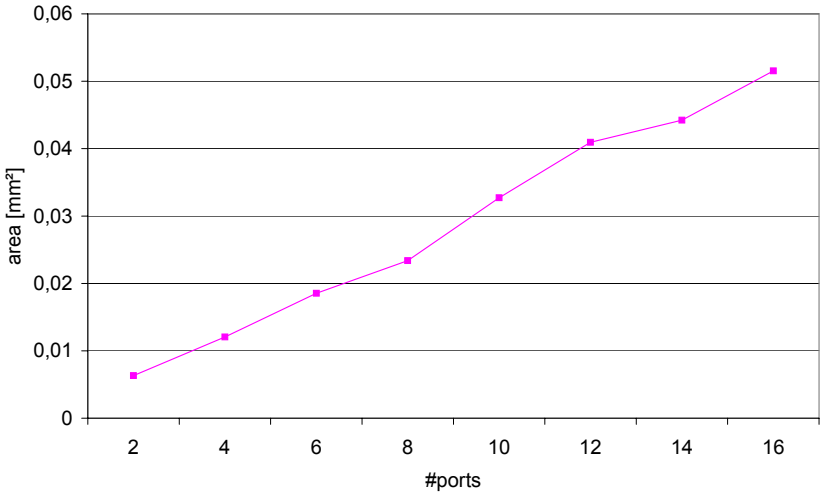


Figure 4.14: Area of the scheduler without priority reordering

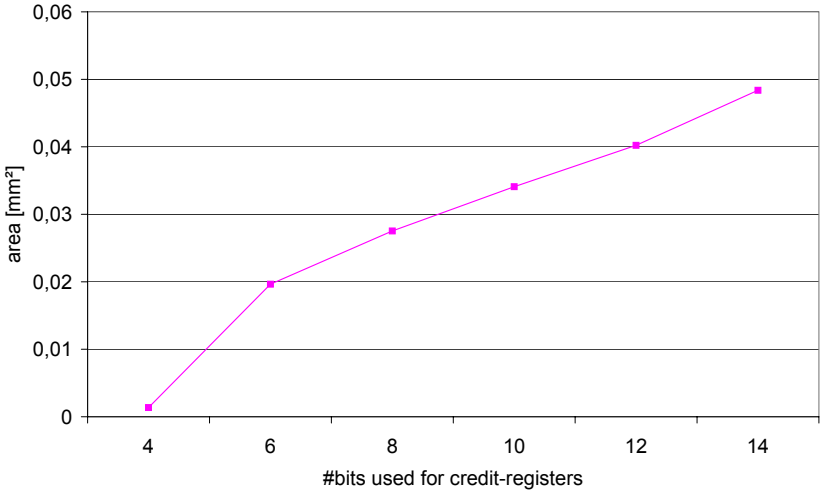


Figure 4.15: Area of the scheduler in respect to the bits used in credit registers

and allows the system architect to reason about the expected area of the used scheduler at design time.

The scheduler was synthesized with 6 ports and 6 bits for the credit registers for a maximum speed test of 260MHz in CMOS12. This allows using the scheduler with a memory controller in a memory system where the data is transferred at 260MHz at both clock edges.

Comparison of the two implementations

As shown in the previous sections in the case of the first implementation (with the switch) the area grows quadratical with the growing number of ports. In the second implementation (without the switch) the area grows linearly. Figure 4.16 illustrates the area consumed by the switch in percent when splitting the scheduler into switch and an arbitration unit.

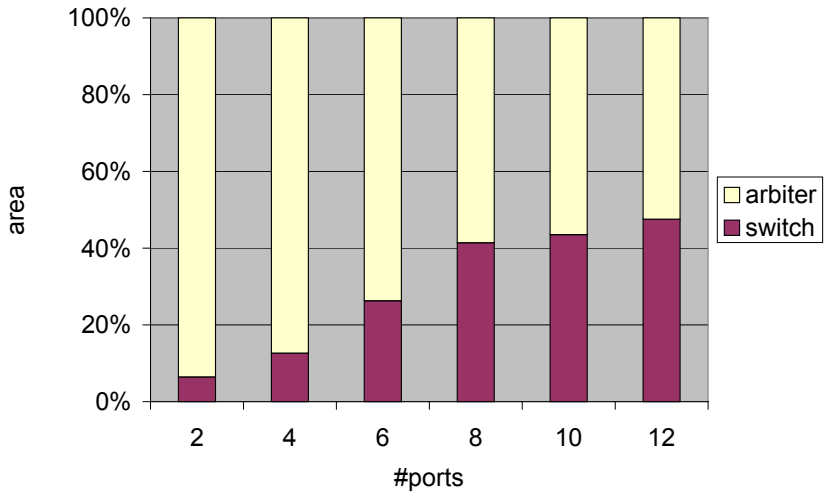


Figure 4.16: Influence on the area of the scheduler from the switch

The diagram shows that with the increasing number of ports the area influence of the switch becomes significantly bigger. For the memory controller design this means that the switch becomes the restricting part when scaling the number of requestors. Therefore, it is necessary to reason about another way of programming the priorities in the memory controller. It is identified in Section 4.2.3 that by ordering the requestors at system level the proposed scheduler allows a scalable design. The reordering at system level is achieved with interchanging the queues, and thus does not require to implement a switch for reordering the priorities. No additional costs are added to the system allowing priority reconfiguration.

4.6 Test strategy

The functionality of the memory controller is tested in three ways. These tests are organized as follows:

1. Verification of the interface (interaction) between the memory controller and the DDR2 memory
2. Verification of the memory controller included in a SoC
3. Functional verification of the scheduler from the memory controller

The first test checks the timing of the memory controller. The timing is verified with a behavioral model of a DDR2 memory from Micron[1]. This test verifies the interface and the interaction between the memory controller and the DDR2 memory. The second test simulates the memory controller together with the behavioral memory model, the network, and the requestors. The behavior of the requestors is modeled with traffic generators (TC). The traffic generators emulate the traffic behavior of the requestors specified in the file communication.xml. The first two tests provide a functional test of the memory controller, without considering the implementation of the scheduler. These tests are described in more detail in the next section. The third test simulates the behavior of the scheduler in isolation. The output of the designed RTL model is verified with the expected scheduler output from a high level model from [ASGR06]. This test is described in Section 4.6.2. Simulations and experiments of the entire system (including TC, NoC, memory controller, and behavioral memory model) are discussed in Section 5 by means of a set-top box use case.

4.6.1 Core Memory Controller

The core memory controller as a whole is tested in two ways at RTL level. In the first case the timing behavior is verified with a behavioral model of a DDR2 memory from Micron. In the second case the memory controller is tested together with Æthereal. The first test is used to verify the behavior of the memory controller at an early stage, assuring that the timing parameters of the DRAM are not violated. Therefore, the behavioral model of a dynamic memory and the memory controller are stimulated with memory requests that are scheduled and converted to DRAM commands. The memory model is taken from Micron (DDR2-400-16bit)[1]. The memory controller and the memory model are instantiated by the testbench. The testbench emulates a requestor that is permanently sending read and write requests to the memory controller. The data that is written is compared with the data, which is read back from the memory. Furthermore, the timing of the memory controller is checked from the memory model, which stops the simulation if the timing is violated. This test essentially confirms the correctness of the basic read, write, and refresh groups. See Figure 4.17 for an illustration.

In the second case the behavior of the total system is verified. Several requestors and a memory are interconnected with Æthereal. To simulate the behavior of the requestors, traffic generators are automatically connected to the network interface. These traffic

[1]Micron has free dynamic memory models accessible on their website www.micron.com

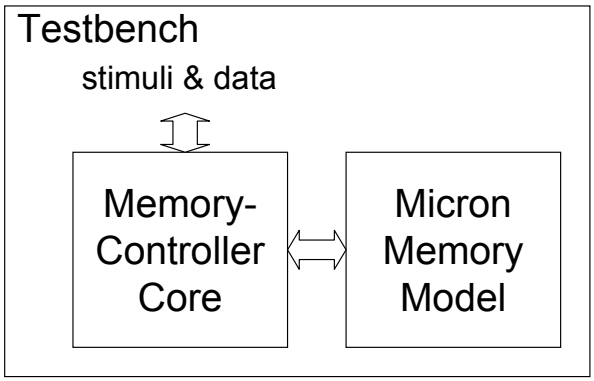


Figure 4.17: Timing test of the memory controller

generators stimulate the memory controller according the traffic specified in the application specification in the Æthereal design flow. The traffic generators send according their specified bandwidths, write accesses to the memory and afterwards read the data from the memory. The data is traced at the data bus between the network interface and the traffic generators and is written to a file. This file is parsed after the simulation and is checked if the data from the write request is the same as the data from the read request. If it is the same then the memory controller combined with the network and the memory works as expected. Figure 4.18 illustrates this test concept. This test checks only the functional correctness of the memory controller, and does not test if the arbitration policy takes care of the latency and the bandwidth requirements. The test concept of the scheduler is described in the next section. Further experiments with set-top box use case are discussed in Section 5.

4.6.2 Scheduler

The arbitration mechanism used in the scheduler is proposed in [ASGR06] and is summarized in Section 4.2. This arbitration mechanism is developed as high-level model in Phyton and as RTL model in VHDL. The high level model is used to develop and to analyze the algorithm. Furthermore, the high level model generates test patterns, which are compared to the RTL model to assure the same behavior. Both models expect as input the specification of the different requestors, with their credit specification (numerator, denominator, maximum credits) and their priority level. At every arbitration cycle the best fitting requestor is selected, and the credits are updated accordingly. For every arbitration cycle the chosen requestor and the current credits are written to an output file. The RTL model of the scheduler is stimulated and verified with a testbench. The test-bench reads the specification of the requestor (the same as for the high-level model) and configures the scheduler via the MMIO like interface.

The testbench starts the arbitration, by invoking a new arbitration cycle. After every arbitration cycle the scheduler provides the best fitting requestor. The testbench reads the selected requestor and writes this information together with the current credit values into the output file. After the simulation the output file of the high level simulation and the output file of the RTL simulation are compared. If the output is equal then the behavior of

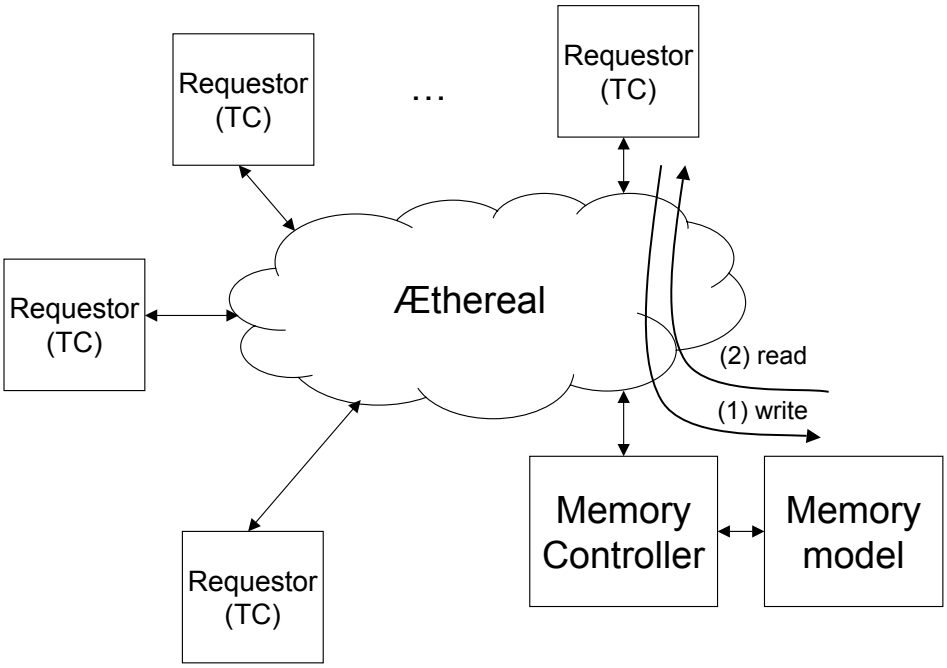


Figure 4.18: The requestor - simulated with traffic generators (TC) - access the memory with write and read accesses

both schedulers is the same. This is simulated for different sets of test cases, with different credit configuration and different number of requestors which gives the same results for both models thus allowing to apply the analysis of the scheduler proposed in [ASGR06] to the RTL model. Figure 4.19 shows an illustration of the described test setup.

4.7 Conclusion

The memory controller is implemented as RTL model in VHDL. The test concept splits the test into a separate test for the scheduler, which is verified against a high level model, and a test for the core memory controller.

However, the memory controller is synthesized for CMOS12 and results in a small memory controller ($0.042mm^2$ for 6 requestors), which is scalable. The area is dominated by the scheduler, as it consumes almost half of the area, since the core memory controller has no data buffers implemented. Furthermore, because of the interleaved access pattern it is not necessary to trace the current state of the memory banks and so it is not necessary to implement registers for tracking the state and the timing of the banks. However, the scheduler is scalable due to the proposed priority reordering with interchanging the connected queues at system level.

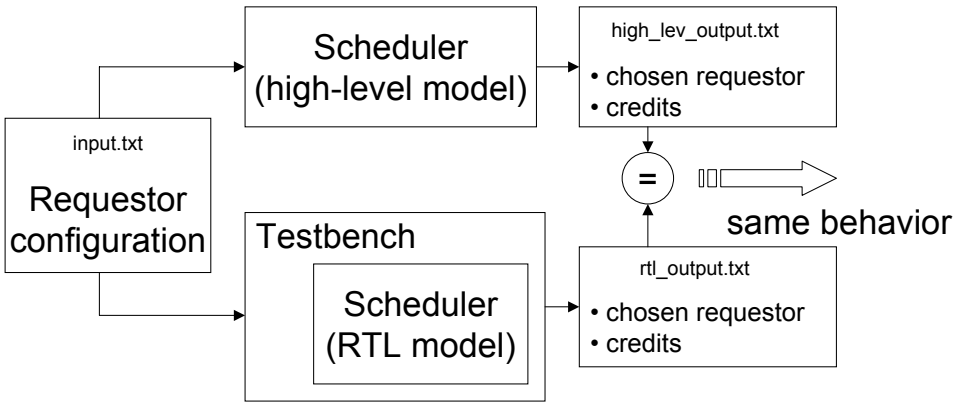


Figure 4.19: The test strategy of the scheduler

Chapter 5

Experiments

This chapter shows by means of a Philips Mid-End multimedia SoC use-case the simulated performance of the memory controller. The example is taken from the specification of a set-top box. Section 5.1 describes the specification of the use-case and how the experiment is carried out. In Section 5.2 the results of the experiments are discussed and are sub-categorized into latency results and efficiency results. The latency results show that they are in the targeted region of the specified use-case. However, the use-case is defined for a burst-size of 32 bytes but the memory controller requires a burst size of 64 bytes. Therefore, the burst size is adapted to 64 bytes. The simulated memory efficiency is above 80%, which does not contradict the calculated gross-to-net bandwidth translation.

5.1 Experiment setup

The memory controller is simulated with a Philips SoC specification for a mid-end SoC. The specification is given in Table 5.1.

Requestor	*Read*		*Write*	
	bandwidth [MB/s]	*max. latency [ns]*	*bandwidth [MB/s]*	*max. latency [ns]*
ARM/MIPS	50	480	50	480
DSP1	25	480	25	480
DSP2	25	480	25	480
Audio-in	0	unspecified	2	unspecified
Audio-out	2	unspecified	0	unspecified
Video-in	0	unspecified	50	unspecified
Video-out	100	unspecified	0	unspecified
Video-proc2d-1	60	360	60	360
Video-dec2d-1	60	360	20	360
Misc	50	800	25	800

Table 5.1: Use-Case definition

The use-case further specifies that all requestors have a burst size of 32 bytes, except requestor *Video-out* with a burst size of 64 bytes. However, a constraint of the predictable

memory controller is connecting requestors with a burst size of 64 bytes. This allows an interleaving access pattern, which leads to an analyzable memory model. Adapting the use-case to a burst size of 64 bytes allows simulating the use-case with the designed memory controller. The sum of the read and write bandwidth gives an overall bandwidth of 629 MB/s. Considering a gross-to-net bandwidth translation factor of 80% translates to an overall gross bandwidth requirement of 786.25 MB/s. A DDR2-400 device with a word width of 16 bits has a gross bandwidth of 800 MB/s and is therefore used for this experiment.

The next step is to translate the bandwidth and latency requirements into the required parameters used by scheduler, which are: Priority, Numerator, Denominator. The scheduler-parameters are automatically derived according to the algorithm specified in [ASGR06] and are shown in Table 5.2.

	Bandwidth-Ratio		*Scheduler-parameters*		
Requestor	*Decimal*	*Fraction*	*Priority*	*Numerator*	*Denominator*
ARM/MIPS	0.156	5/32	3	5	32
DSP1	0.078	5/64	4	5	64
DSP2	0.078	5/64	5	5	64
Audio-in	0.004	1/256	7	1	256
Audio-out	0.004	1/256	8	1	256
Video-in	0.078	5/64	9	5	64
Video-out	0.156	5/32	10	5	32
Video-proc2d-1	0.188	46/245	1	46	245
Video-dec2d-1	0.125	1/8	2	1	8
Misc	0.117	17/145	6	17	145

Table 5.2: Scheduler parameters

Programming the memory controller with the above mentioned parameters, allows an RTL simulation of the design. Since the behavior of the scheduler implemented in RTL is equal to the behavior of the scheduler in the high level, it is possible to measure the latency results in the high level model. The results to this simulation are presented in the next section.

5.2 Results

The next sections show the latency results derived from the high level simulation. They, furthermore, show efficiency results from the RTL simulation, where the whole network including the memory controller and a memory model is simulated according the defined use-case.

5.2.1 Latency

The latency results are analytically calculated and measured in the high level model. The latency is accounted as specified in Section 5.1 The latency results are given in Table 5.3.

Requestor		*Analytical worst-case*		*Measured worst-case*	
	Specified Latency [ns]	*Scheduler Latency [cycles]*	*Access Latency [ns]*	*Scheduler Latency [cycles]*	*Access Latency [ns]*
ARM/MIPS	480	3	555	2	465
DSP1	480	7	915	3	555
DSP2	480	11	1275	4	645
Audio-in	unspec.	27	2715	10	1185
Audio-out	unspec.	34	3345	12	1365
Video-in	unspec.	37	3615	19	1995
Video-out	unspec.	75	7035	23	2355
Video-proc2d-1	360	1	375	0	285
Video-dec2d-1	360	2	465	1	375
Misc	800	13	1455	5	735

Table 5.3: Latency results

The access latency is the overall time needed for one access. This is the time that an access stays at the head of the input queue until the access is finished. For a read request, the time is stopped when the last data word is placed in the output queue. In case of a write access the time is stopped when the last word is written to the memory. The access latency consists of the latency needed by the arbitration, the memory controller, and the memory, as specified in Equation (4.3) and in Equation (4.4).

5.2.2 Efficiency

As the memory controller allows to guarantee a gross-to-net bandwidth translation a minimum efficiency of the memory is always achieved. Figure 5.1 shows the efficiency of the memory from start up. The efficiency is specified in Equation (2.1) as the fraction of *data_cycles* over *total_cycles*. The *total_cycles* are only counted if at least one requestor is pending. Which means the efficiency goes down in case no data is transfered and there are pending requests, for example when a memory refresh is performed. The simulation results are derived from an RTL simulation with the use-case defined in Table 5.1.

At the start-up the memory is initialized by the memory controller. In the start-up phase no efficiency is measured. After the first full request arrives over the network the memory controller starts to process the request. The efficiency goes up at clock cycle 1069. However, the system is not backlogged yet and no data is pending so the efficiency goes down again. In this case the efficiency is below the promised 80% due to the fact that the system ist not stable yet. Which means that when serving just one request and afterwards returning to the idle state causes more idle states on the bus than in the case when the system is stable. This start-up phase is described in Section 3.3.1. After clock cycle 1247 other requests arrive and the efficiency reaches a maximum. After clock cycle 1603 the refresh causes the efficiency to go down, but after that the system is stable at an efficiency above 80%. The second and the third refresh period show another dropping of the effiency at clock cycle 3205 and 4629. As expected the efficiency of the memory controller is above 80% when the system is stable.

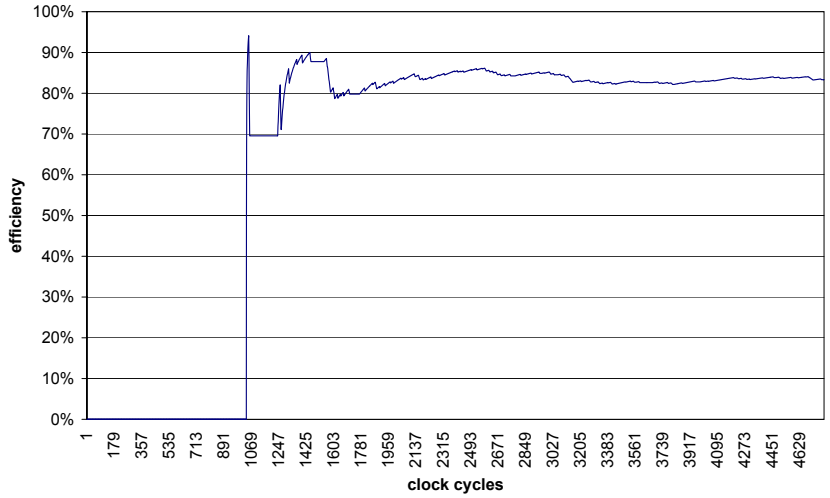


Figure 5.1: Measured accumulated efficiency

In Figure 5.2 the current efficiency of the memory is shown. This allows visualizing the impact of the refresh cycles on the efficiency better. The efficiency is calculated according to Equation (2.1) but only the last 50 clock cycles are considered.

At the start-up the memory controller initializes the memory. In this figure the efficiency is allowed to drop below 80% as it integrates over 50 cycles. Refresh for instance causes a huge drop that sticks around for a while due to integration. Another drop is caused by read/write switches and vice versa. Note that in 50 cycles three read/write switches can happen, which is visible in a decreasing efficiency.

The first request is received at clock cycle 1069. Since the system is not backlogged the efficiency goes down. New request arrive at clock cycle 1247 and the efficiency goes above 80%. The first three refresh cycles are shown at clock cycle 1603, 3205, and 4629, where the efficiency gets really low. The other points where the efficiency goes under 80% indicate that there are a read/write or a write/read switches. Nevertheless, the efficiency staying below 70% after clock cycle 3561 indicates that the system is not backlogged. The necessity of precharging the open banks before entering the idle state causes the system efficiency going below 80%. In the long run the overall efficiency is never below 80%, and does not contradict the proposed and calculated gross-to-net bandwidth translation.

5.3 Conclusion

Starting from a use-case, which defines a Philips SoC for mid-end requirements, like a set-top box, experiments are carried out and efficiency and latency results are derived. The specification of the use-case specifies the requestor with a burst size of 32 bytes. However,

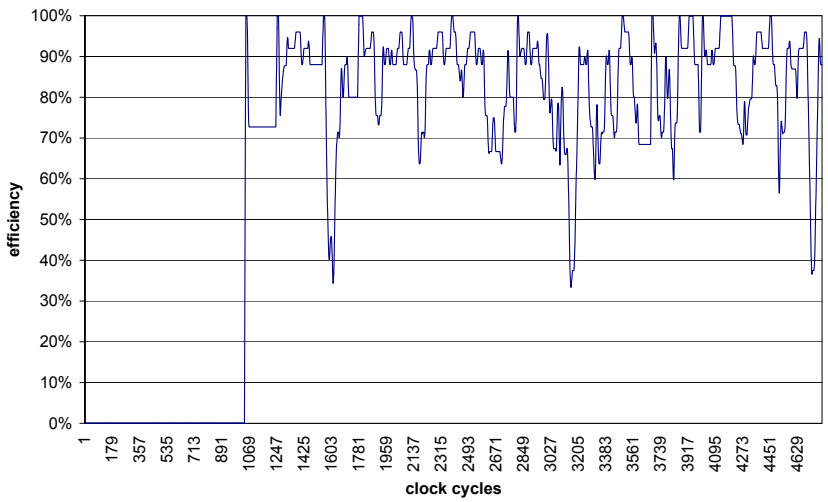


Figure 5.2: Measured smoothened (over 50 cycles) instantaneous efficiency

the memory controller is simulated with a burst size of 64 bytes. Therefore, the latency results are not completely comparable to the latency specified. Nevertheless, the latency results are in the region of the specified latency, which makes the design of the memory controller promising for further generations, where the burst-size is expected to become bigger. The measured efficiency results show that a gross-to-net bandwidth translation is provided. The efficiency stays after the start-up phase always above 80%.

Chapter 6

Final Remarks and Outline

This thesis describes the design and implementation of a memory controller for real-time systems. This chapter concludes the thesis in Section 6.1 and gives an outline of future work in Section 6.2.

6.1 Conclusion

A memory controller is proposed, which is predictable and thus can be used for real-time systems. The memory controller is co-designed for the use with Æthereal, but can be used with in any system where a predictable memory controller is required.

The predictability of the system is reached with an interleaved access pattern and a predictable scheduler. This allows a gross-to-net bandwidth translation of the memory. Furthermore, a high efficiency is gained with this access pattern, where the efficiency stays above 80%. The maximum latency for an access is calculated analytically at design time and is for highest priority requestor in the worst-case $375ns$.

The memory controller is synthesized with Synopsis for CMOS12 and results in a small memory controller ($0.042mm^2$ for 6 requestors), which is scalable. The scheduler consumes almost half of the area, since the core memory controller is small in size because it has no data buffers implemented. Furthermore, because of the interleaved access pattern it is not necessary to trace the current state of the memory banks and thus area for registers which track the state and the timing of the banks is saved. However, the scheduler is scalable due to the proposed priority reordering with interchanging the connected queues at system level.

6.2 Outline

Further work involves analyzing the latency of the proposed system to optimize for lower latency. A possible way to do so is to look for smaller access granularity, since the latency is influenced by the packet size of a request and therefore can be lowered with the access granularity. However, lowering the latency introduces a lower efficiency too. Data reordering at the memory controller could be a way to gain higher efficiency, but with the drawback that the requests depend on other requestors. A further optimization would be that currently the scheduler awaits a full request in the input buffers, until the request is

allowed to be scheduled. A prediction of the arrival process could be a way to save time before the request is fully present at the buffers, but has already enough credits and can be scheduled.

Appendix A

Glossary

A.1 Used acronyms

TG	Traffic Generator
IP	Intellectual Property
CPU	Central Processing Unit
CA	Communication Assist
DSP	Digital Signal Processor
RAM	Random Access Memory
DDR	Double Data Rate
DDR2	Double Data Rate version 2
SDRAM	Synchronous Dynamic RAM
RTL	Register Transfer Level
QoS	Quality-of-Service
CRC	Cyclic Redundancy Check
MC	Memory Controller
TCL	Tool Command Language
VHDL	Very High Speed Integrated Circuit Hardware Description Language
MMBD	Memory Mapped Block Data

Bibliography

[Å05] Benny Åkesson. *An analytical model for a memory controller offering hard-real-time guarantees*. Lund's Institute of Technology, MSc thesis, May 2005.

[ARM04] ARM. *PrimeCell Synchronous Static Memory Controller*, 2004.

[ASGR06] Benny Åkesson, Lisbeth Steffens, Kees Goossens, and Markus Ringhofer. *Credit-Controlled Static Priority Arbitration - Analysis and Implementation*. unpublished note, submitted to DATE07, September 2006.

[Ass04] JEDEC Solid State Technology Association. *DDR2 SDRAM Specification, jesd79-2a ed*. EDEC Solid State Technology Association 2004, 2500 Wilson Boulevard, Arlington, VA 22201-3834, January 2004.

[BB04] D. Bertozzi and L. Benini. Xpipes: A Network-on-Chip Architecture for Gigascale Systems-on-Chip. *IEEE Circuits and Systems Magazine*, 4(2):18–31, 2004.

[BM02] Luca Benini and Giovanni De Micheli. Networks on chip: A new paradigm for system on chip design. In *IEEE Computers*, pages 35(1):70–80, 2002.

[CS99] D.E. Culler and J.P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers Inc., San Francisco, 1999.

[DRGR03] John Dielissen, Andrei Rădulescu, Kees Goossens, and Edwin Rijpkema. Concepts and Implementation of the Philips Network-on-Chip. In *IP-Based SOC Design*, November 2003.

[GDG+05] Kees Goossens, John Dielissen, Om Prakash Gangwal, Santiago González Pestana, Andrei Rădulescu, and Edwin Rijpkema. A Design Flow for Application-Specific Networks on Chip with Guaranteed Performance to Accelerate SOC Design and Verification. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1182–1187, March 2005.

[GDR05] Kees Goossens, John Dielissen, and Andrei Rădulescu. The Æthereal Network on Chip: Concepts, Architectures, and Implementations. *IEEE Design and Test of Computers*, 22(5), Sept-Oct 2005.

[GGRN04] Kees Goossens, Om Prakash Gangwal, Jens Röver, and A. P. Niranjan. Interconnect and Memory Organization in SOCs for advanced Set-Top Boxes and TV — Evolution, Analysis, and Trends. In Jari Nurmi, Hannu Tenhunen, Jouni Isoaho, and Axel Jantsch, editors, *Interconnect-Centric Design for Advanced SoC and NoC*, chapter 15, pages 399–423. Kluwer, April 2004.

[GRG+05] Om Prakash Gangwal, Andrei Rădulescu, Kees Goossens, Santiago González Pestana, and Edwin Rijpkema. Building Predictable System On Chip: An Analysis Of Guaranteed Communication In The Æthereal Network On Chip. In Pieter van der Stok, editor, *Philips Research Book Series Volumne 3*, chapter 1, pages 1–36. Springer, 2005.

[HdCLE03] S. Heithecker, A. do Carmo Lucas, and R. Ernst. A mixed QoS SDRAM controller for FPGA-based high-end image processing. In *Signal Processing Systems, 2003. SIPS 2003. IEEE Workshop on*, pages 322–327, August 2003.

[HP03] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2003.

[Lap93] Philip A. Laplante. *Real-time Systems Design and Analysis - An Engineer's Handbook*. IEEE Computer Society Press, New York, 1993.

[LB03] Andy S. Lee and Neil W. Bergmann. On-chip interconnect schemes for reconfigurable system-on-chip. December 2003.

[LLJ05] Kun-Bin Lee, Tzu-Chieh Lin, and Chein-Wei Jen. An efficient quality-aware memory controller for multimedia platform SoC. In *Circuits and Systems for Video Technology, IEEE Transactions on*, volume 15, page 5pp., New York, NY, USA, May 2005. ACM Press.

[MVF02] Jens Muttersbach, Thomas Villiger, and Wolfgang Fichtner. Practical design of globally-asynchronous locally-synchronous systems. In *Proc. Int'l Symposium on Asynchronous Circuits and Systems*, 2002.

[OHM05] Umit Y. Ogras, Jingcao Hu, and Radu Marculescu. Key research problems in NoC design: a holistic perspective. In *CODES+ISSS '05: Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 69–74, New York, NY, USA, 2005. ACM Press.

[RDGP+05] Andrei Rădulescu, John Dielissen, Santiago González Pestana, Om Prakash Gangwal, Edwin Rijpkema, Paul Wielage, and Kees Goossens. An Efficient On-Chip Network Interface Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Programming. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 24(1):4–17, January 2005.

[RDK+00] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. Memory access scheduling. *SIGARCH Comput. Archit. News*, 28(2):128–138, 2000.

[RGAR+03] E. Rijpkema, K. Goossens, J. Dielissen A. Rădulescu, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade Offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip. *IEE Proceedings: Computers and Digital Technique*, 150(5):294–302, September 2003.

[RSM01] K. Ryu, E. Shin, and V. Mooney. A Comparison of Five Different Multiprocessor SoC Bus Architectures. In *EUROMICRO Symposium on Digital Systems Design*, pages 202–209, September 2001.

[SLKH02] E. Salminen, V. Lahtinen, K. Kuusilinna, and T. Hämäläinen. Overview of bus-based system-on-chip interconnections. In *IEEE Int'l Conf. Circuits and Systems*, pages II–372–II–375 vol. 2, 2002.

[SSM+01] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vincentelli. Addressing the System-on-a-Chip Interconnect Woes through Communication-Based Design. In *Proc. Design Automation Conference (DAC)*, pages 667–672, June 2001.

[SSTN02] I. Saastamoinen, D. Siguenza-Tortosa, and J. Nurmi. Interconnect IP node for future system-on-chip designs. In *The First IEEE International Workshop on Electronic Design, Test and Applications*, pages 116–120, 2002.

[vEHJ+05] Jos van Eijndhoven, Jan Hoogerbrugge, M. N. Jayram, Paul Stravers, and Andrei Terechko. Cache-Coherent Heterogeneous Multiprocessing as Basis for Streaming Applications. In Peter van der Stok, editor, *Dynamic and Robust Streaming In And Between Connected Consumer-Electronics Devices*, volume 3 of *Philips Research Book Series*, chapter 3, pages 61–80. Springer, 2005.

[Web01] W.-D. Weber. *Efficient Shared DRAM Subsystems for SOCs.* Sonics, Inc, 2001. White paper, 2001.

[Wol05] Luud Woltjer. *Optimal DDR controller.* University of Twente, MSc thesis, 2005.