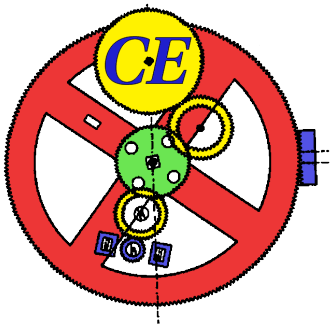


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2008

MSc THESIS

A predictable and composable front-end for system on chip memory controllers

Eelke Strooisma

CE-MS-2007-19

Abstract

Today, verification and integration dominate the cost of developing a System-on-chip. A front-end for a predictable and composable memory controller is proposed that has the purpose to reduce verification and integration effort. The front-end acts as a scheduler for shared external memory; a back-end is required to access the memory. A predictable memory controller guarantees maximum latency and a minimum net bandwidth at design time. This allows real-time requirements to be satisfied without simulation. Composability means that the service of a requestor is not affected by the behavior of other requestors. Hence, components can be verified in isolation and do not need to be reverified after integration. The behavior of a back-end is abstracted by memory accesses. A predictable and composable mapping from memory accesses to SDRAM commands is proposed. Analysis of the memory accesses shows that the access granularity must be increased for newer memory devices to maintain high efficiency. A modular design and strict separation of concerns is essential to simplify timing analysis. When composability is required, responses are delayed such that the behavior is not affected by interference from other requestors. The front-end is synthesized for CMOS090LP technology. The predictable front-end consumes $0.201mm^2$ for five requestors and when composability is enabled, $0.246mm^2$ is required. However, the buffers to delay the responses need $0.76mm^2$ additionally.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

A predictable and composable front-end for system on chip memory controllers

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Eelke Strooisma
born in Leeuwarden, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

A predictable and composable front-end for system on chip memory controllers

by Eelke Strooisma

Abstract

Today, verification and integration dominate the cost of developing a System-on-chip. A front-end for a predictable and composable memory controller is proposed that has the purpose to reduce verification and integration effort. The front-end acts as a scheduler for shared external memory; a back-end is required to access the memory. A predictable memory controller guarantees maximum latency and minimum net bandwidth at design time. This allows real-time requirements to be satisfied without simulation. Composability means that the service of a requestor is not affected by the behaviour of other requestors. Hence, components can be verified in isolation and do not need to be reverified after integration. The behaviour of a back-end is abstracted by memory accesses. A predictable and composable mapping from memory accesses to SDRAM commands is proposed. Analysis of the memory accesses shows that the access granularity must be increased for newer memory devices to maintain high efficiency. A modular design and strict separation of concerns is essential to simplify timing analysis. When composability is required, responses are delayed such that the behaviour is not affected by interference from other requestors. The front-end is synthesized for CMOS090LP technology. The predictable front-end consumes $0.201mm^2$ for five requestors and when composability is enabled, $0.246mm^2$ is required. However, the buffers to delay the responses need to store 26k bits in total, resulting in an additional area of $0.76mm^2$.

Laboratory : Computer Engineering
Codenumber : CE-MS-2007-19

Committee Members :

Advisor:	Kees Goossens, CE, TU Delft, NXP Semiconductors
Member:	Henk Sips, PDS, TU Delft
Member:	Ben Juurlink, CE, TU Delft
Member:	Said Hamdioui, CE, TU Delft

Contents

List of Figures

List of Tables

List of Terms

AHB	Advanced High-speed Bus
AMBA	Advanced Microcontroller Bus Architecture
AXI	Advanced eXtensible Interface
CCSP	Credit-Controller Static-Priority
CoMPSoC	Composable and Predictable Multi-Processor System on Chip
CAM	Composable memory access to pattern map
CPU	Central Procession Unit
DMA	Direct Memory Access
DDR	Double Data Rate
DDR2	Double Data Rate 2
DDR3	Double Data Rate 3
DTL	Device Transaction Level
FIFO	First In First Out
HD	High Definition
IP	Intellectual Property
LCD	Liquid Crystal Display
LPDDR	Low Power Double Data Rate
NoC	Network on Chip
PAM	Predictable memory access to pattern map
QoS	Quality of Service
RAM	Random Access Memory
SDRAM	Synchronous Dynamic RAM
SoC	System on Chip
TDM	Time Division Multiplexing
VHDL	Very high speed integrated Hardware Description Language

Acknowledgements

This thesis is the result of a project offered by NXP Semiconductors in cooperation with Delft University of Technology. First of all, I want to thank NXP Semiconductors for offering me the possibility to do an internship and performing research on this very interesting topic. Initially, I was working at IP & Architecture group of the department Corporate I&T. I would like to thank Ad Siereveld for helping me to understand the architecture and various issues of memory controllers. Later on, I moved to the SoC Architectures and Infrastructure (SAI) group of the Research department.

Both groups are located at the High Tech Campus in Eindhoven, The Netherlands. I liked doing my thesis in such an industrial environment, because it resulted in a practical project. Besides that, I learned a lot from all the people around me sharing their knowledge.

For supervising my activities and results I would like to thank Kees Goossens. During my internship I worked together with Benny Akesson who is doing a PhD at NXP Semiconductors about predictability and composability in external memory storage services. He helped me a lot with the analysis and concepts of my project.

Eelke Strooisma
Delft, The Netherlands
May 12, 2008

1 Introduction

1.1 Problem description

Due to the miniaturisation of transistors, complete systems are implemented on a single chip. Such chips are called System on Chip (SoC). SoCs consist of numerous Intellectual Property (IP) components. Compared to a system composed of several chips this has the advantage that it has higher performance and lower power consumption, because the distance between the IP components is shorter. Also the production is cheaper, since only one package has to be made.

The miniaturisation of transistors reduces the cost and allows the implementation of more and more functionality by a SoC. The drawback is that complexity of the contemporary SoCs is very high, and is probably getting higher in the future. When the same development methods are used, the time to develop such systems will increase. While time-to-market is getting more and more important, improvement of the development methods is critical.

The effort of design, integration and verification of the IP components of a SoC rapidly increases for larger components [1]. A way to reduce the size of the components, and therefore decreasing the complexity, is to make a hierarchy of components. Another method to decrease development time is to avoid as much design and verification as possible. The key is to create more universal components that can be reused for new applications. Most systems use common components like processors, audio/video decoders and memory. Reusable components that already have been designed and verified save a lot of time, thereby increasing design productivity [1]. Only new features of a SoC have to be developed, and ideally the rest can be dragged and dropped into the design. This idea is supported by the fact that new systems are often an improved version of an older system.

The first drawback is that the cost of universal components (like performance, power consumption and chip area) is always higher than that of specialized ones. Secondly, it is not easy to create and integrate such reusable components. Almost all of the SoCs have specific timing requirements. For these systems, the integration of components to form the system is difficult, because the timing can depend on the other components. For universal components this is even a bigger problem. At the time that they are being created, the environment is unknown. Structured methods for the design of IP components are developed to solve this problem.

1.2 Goal

At the SAI group of NXP Research, a platform template called CoMPSoC is being defined and developed to reduce the development effort of SoCs from an architectural point of

view [8]. CoMPSoC is a template for a SoC containing different kinds of IP components, resources and multiple processors, all connected by a Network on Chip (NoC). Currently, the focus is on predictable and composable design methodologies. They have the purpose to ease integration and verification of the IP components. Often, external memories shared by multiple IP components are used to reduce the cost. External memories offer high bandwidth and capacity at a low cost per pin. A memory controller is required to communicate with an external memory. The current prototype implementation of the platform template does not have a controller for external memory. The main goal of the graduation project is to create a memory controller that fits into the platform template and conforms to predictability and composability. The memory controller also has to be universal such that it can be reused more easily. An important aspect of the design and implementation process is to investigate the additional cost caused by the predictability and composability requirements.

1.3 Context

A SoC is used to run one or more applications. An application consist of a number of jobs that are started and stopped to perform different functions. The jobs are dedicated to a specific function and have no direct relation with other jobs. The tasks of a job perform the actual work. Tasks have dependencies between each other. Figure 1.1 shows this application model. The model only describes how the application should work and how the jobs and tasks are related. In what way the model is mapped to hardware and software depends on the constraints of the application. Common constraints are performance, power consumption and time-to-market.

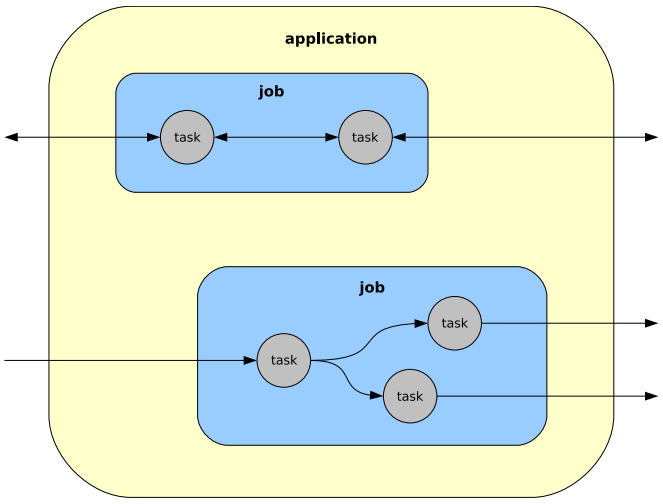


Figure 1.1: Application, jobs and tasks

Jobs that are independent have their own memory space. In an implementation, such jobs could have separate memories. To reduce the cost in terms of price, area, input and output pins, often an external memory is used, as mentioned earlier. Figure 1.2 shows an example of a SoC architecture that uses external memory. The application

with its jobs and tasks are mapped to the IP components. This architecture uses a NoC as interconnect. All IP components are connected to the NoC by means of network interfaces. Note that the memory controller is also an IP component. This infrastructure allows the IP components to communicate with the memory controller, and indirectly the memory. The memory controller forms the bridge between the IP components and the external memory. The IP components that send requests to the memory controller are called initiators. The memory is the target that responds to the request.

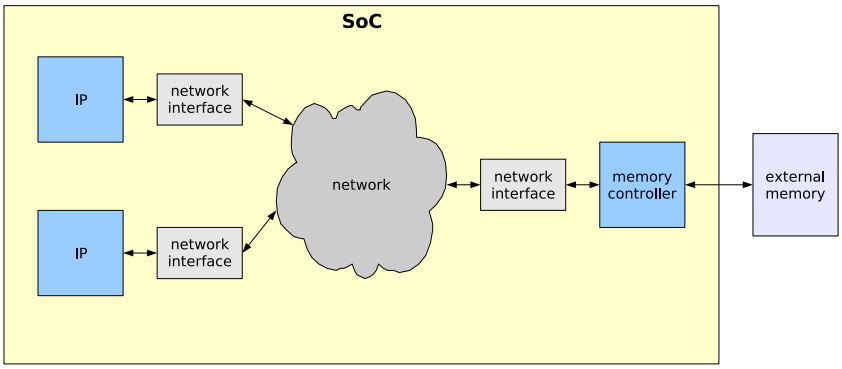


Figure 1.2: Memory controller inside a SoC

A more abstract representation of the environment of the memory controller is illustrated in Figure 1.3. The memory controller can handle requests and responses of multiple requestors to allow memory sharing. The requestors can use any kind of interconnect to communicate with the memory controller as long as that interconnect can provide the right interface for the ports of the memory controller. To access the memory, a requestor sends requests to its corresponding port at the memory controller. There are two types of requests for a memory: read and write request. A read request instructs the memory controller to get data from the memory. The read data is sent back as part of the response. When a requestor wants to write data to the memory, it issues a write request including the data. The response contains the result of the write request. An IP components that access the memory are called requestors. In practice this could be components like Direct Memory Access (DMA) controllers and processors. It is possible that a single IP component is connected to multiple ports. In this case the IP component corresponds to multiple requestors. Multiple IP components can be aggregated on one port, but they are regarded as a single requestor.

1.4 Structure

This thesis is intended to present the activities, design process, issues and results, such that it can be used to continue the work on the composable and predictable SoC. From the problem and goals, the requirements to build the memory controller are derived in Section 2. Section 3 discusses related research on the field of memory controllers, predictability and composability. Information about SDRAM is given in Section 4 to provide more understanding of the issues in the memory controller caused by the memory. The

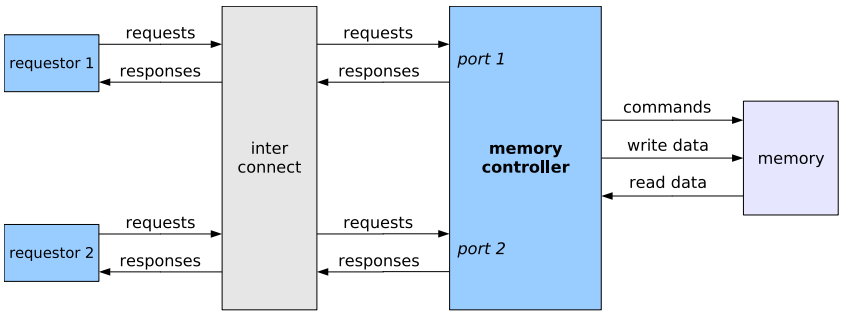


Figure 1.3: Abstract interface of a memory controller

basic functionality of memory controllers is explained in Section 5. Section 6 introduces and defines the memory command patterns, which are used to get a predictable and composable memory controller. At this point, there is enough background information to explain the design. Given the requirements, the design is proposed in Section 7. This design is implemented using VHDL, such that it can be synthesized. Section 8 discusses the architecture of the hardware implementation in detail. Using that implementation several experiments are done to verify and analyse the behaviour. The results of these experiments are shown and discussed in Section 9. Also synthesis results are presented in this section. Section 10 summarizes important aspects of the integration and verification problem and the proposed solution and implementation. Finally, Section 11 presents future work that could improve the proposed memory controller, or lead to better solutions. The remainder of the report consists of appendices which can be used as reference for details of the project.

2 Requirements

As discussed in Section 1, a major problem of current SoC development is the complexity of integration and verification. The goal is to design a memory controller that fits in a predictable and composable SoC architecture. This section introduces and motivates the requirements for such a memory controller. Section 2.1 explains what predictable and composable components are and why they simplify integration and verification of IP components. Section 2.2 discusses the secondary requirements, which have the objective to get a usable memory controller.

2.1 Verifiable

Verification of a system is the task of checking if the design is actually implementing the intended behaviour. One of the most difficult tasks of verification is to check if the real-time requirements are satisfied.

2.1.1 Verification of real-time systems

Systems with explicit timing requirements are called real-time systems. Besides correct functional behaviour, such systems have to guarantee timing behaviour to a certain extent. The system may not behave correctly or may even cause harm when the timing is violated.

Two categories of real-time requirements exists: hard and soft real-time requirements. Hard real-time requirements contain deadlines which may never be missed. Missing these deadlines results in a system that could cause harm or is useless. The property of soft real-time requirements is that missing a small number of soft deadlines does not result in an unusable system. To what extend misses may occur depends on the application.

Applications on a personal computer like a word processor do not have timing constraints. The time between typing and displaying the text on the screen is not bounded. The computer tries to make the delay as short as possible (best effort), but it may take a long time when the system is very busy. This is probably irritating for the user, but it does not affect the functionality. The timing behaviour for SoCs that are applied in embedded systems is often much more critical. An example of an application with hard real-time requirements is the ignition system for engines. It has strict timing bounds for certain actions (i.e. the time to ignite) to guarantee that the engine runs smoothly and is not damaged. Video or audio decoders often have soft real-time requirements. Deadline misses typically result in some distortion in the image or sound. A manufacturer of professional recording equipment has a lower tolerance of such effects than when the chip is used by a television for home use.

2.1.1.1 Dependencies

A dependency exists between two tasks when the behaviour of one task could be affected by the second task. Verification of a task in isolation is not possible when there are dependencies. Tasks with a lot of dependencies are hard to verify because all dependencies must be part of the verification process.

During the integration process, IP components are connected to each other and therefore (new) dependencies are created between tasks. At the time all dependencies exist, real-time requirements can be verified. A violated requirement could be fixed by reconfiguring or using a different IP component to execute the task for example. However, all dependent tasks have to be verified again because their timing behaviour may have been changed, possibly resulting in new violations.

To illustrate how dependencies are created and how they affect verification, consider the fictive application in Figure 2.1. The application has a job for image enhancement and one that processes an audio stream. The audio processing job consists of a task that decodes an audio stream and a task that applies some filter. The filter is dependent on the decoding task, since that one provides the input for the filter. Dependencies in the application model are called *explicit*.

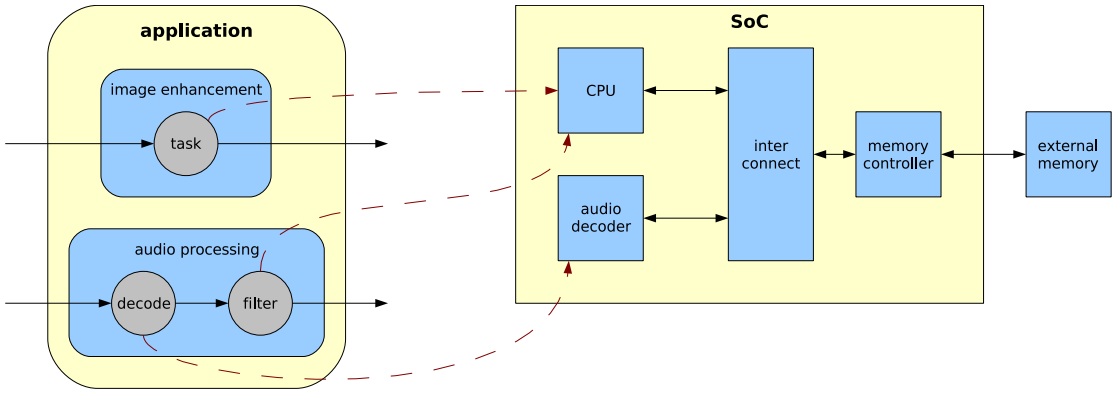


Figure 2.1: Mapping of an application to a SoC

As mentioned in Section 1.3, an application with its tasks and jobs is mapped to hardware and software. Often, there is no one to one relation between jobs, tasks and the IP components of a SoC. It could be the case that a task needs multiple IP components and multiple tasks share the same IP component. Figure 2.1 shows how the application is mapped to a SoC. The dashed arrows indicate which IP components execute the tasks. The CPU is used to execute the task of the image enhancement job, but also for filtering the audio stream. The external memory is used as input and output of the image and audio stream. Three shared resources can be identified: the CPU, interconnect and the memory. Due to the mapping, all introduce additional dependencies. First, the CPU has to execute tasks of different jobs, which do not have a dependency in the application model. Secondly, the memory and interconnect are shared by the CPU as well as the audio decoder. These dependencies are called *implicit*.

Some applications allow starting and stopping jobs at run-time. This causes depen-

dencies to change at run-time. Assume that a third job is started at run time, which contains a task that must be executed by the CPU. Now there exists implicit dependencies between all three jobs of the application. To determine the timing behaviour of a single job, all three jobs have to be analysed.

Like the example, the requestors of a shared memory (or any other shared resource) usually do not belong to a single job. The service that a requestor receives depends on the behaviour of other requestors and therefore causes implicit dependencies. When one requestor needs less service than allocated, the other requestors can get more.

2.1.1.2 Timing analysis

To check if all real-time requirements are satisfied, the timing behaviour must be analysed. In general, there are two methods for timing analysis:

- *Static timing analysis*: Analysing the source of the system to determine the timing behaviour.
- *Dynamic timing analysis*: Simulate or run the system using different kinds of input and measure the timing behaviour

Using static timing analysis, the timing behaviour of an system can be modelled and real-time requirements can be formally verified. A correct model gives conservative results for verification of real-time requirements (i.e. worst case execution time of a task is too high, best case execution time is too low). A more exact model gives a better estimation, but may be more difficult to derive and analyse. Guaranteeing real-time requirements is not possible for tasks with unbounded timing behaviour. An example of such a task is one that waits for user input. Since the user could never give input, it could take forever before the task finishes. An advantage of static timing analysis is that it can be done at design time and allows structured and hierarchical verification.

Dynamic timing analysis gives optimistic results (like underestimation of the worst case execution time of a task). It is infeasible to test all combinations of inputs and states of the system [1]. The worst case or best case might never happen during dynamic timing analysis, because the whole system cannot be covered. Hard real-time requirements cannot be guaranteed using this method, making it unsuitable for systems which have those requirements. Another disadvantage is that the complete system or an accurate simulation model must be ready before it can be analysed. However, dynamic timing analysis is relatively easy to perform, since it consist of feeding input patterns to the design and measuring the behaviour.

2.1.2 Predictable

As mentioned in Section 2.1.1.2, real-time requirements of the tasks (and jobs) of an application can only be verified when the model of the task produces bounded results. These tasks are predictable according to Definition 2.1. Note that a job or task is not predictable when they could execute unpredictable tasks or subtasks. Definition 2.2 defines the relation between predictable tasks and predictable components. An application that is composed of predictable jobs and tasks has the advantage that hard real-time

requirements can be satisfied because static timing analysis can guarantee that a hard deadline is never missed. In addition, real-time requirements can be verified during design time. However, a predictable component has more design restrictions. It must be designed in a such way that its behaviour can be analysed and bounded by available methods.

Definition 2.1 *A task is* **predictable** *when its behaviour can be bounded at design time.*

Definition 2.2 *A* **predictable component** *is a component that only executes predictable tasks.*

The tasks that map to the memory controller are read and write operations. A read or write operation is initiated by a request. After the operation finished, the response is sent back. By Definition 2.2, a predictable memory controller requires that all tasks are predictable. This means that the following behaviour must be bounded at design time:

- *Latency*: The time between request and response
- *Net bandwidth*: The amount of data that can be read or written in a certain time interval

The bounds are defined as maximum latency and minimum net bandwidth. Net bandwidth corresponds to the actual amount of useful data that can be accessed by a requestor in a fixed time interval. In contrary, gross bandwidth refers to the maximum amount of data that could be accessed in a fixed time interval. Section 4.6 discusses the relation between gross and net bandwidth.

It is obvious that the memory affects the behaviour of a request. Therefore, a predictable memory controller requires a predictable memory. Section 6 shows that an SDRAM device is always predictable, but not necessarily efficient.

2.1.3 Composable

Having a system built from predictable components is a major advantage for verification. No simulation is needed to verify the timing behaviour, because the bounds are already known. To further reduce the verification complexity, the composability methodology is introduced. In general, components of a composable system can be verified and integrated easily. According to Kopetz [15], the key properties of the components of a composable system are:

- *Independence:* The component should be independent, assuring that it can be designed and verified in isolation of other components. A proper interface that describes time and value domain is vital.
- *Invariance:* The behaviour of the component specified by the interface description must not change after integration. When the component does not behave as described by its interface, verification using that interface may produce invalid results.

- *Growth:* When inserting a component in the system, it may not influence the behaviour of the existing components.

Ideally, all components of a composable system comply with these properties in any situation. In practice, these requirements are only met under certain conditions. The behaviour of the system can only be guaranteed when all conditions are met. The verification process must check if one or more conditions are not met and fail when the behaviour is not guaranteed.

The architecture of a system - how components are defined and communicating - is crucial for the independence property. It reduces the complexity of development, because verification of a single component is easier than multiple components simultaneously. The independency property also enables more parallelism in the development process.

The advantages of reusable components can be exploited by the invariance property. It reduces the development time of a SoC in two ways. First, verification can be performed before the system is designed, thereby removing this process from the critical path of development of a system. Secondly, the component only needs to be verified once. Besides the development, there is an advantage for maintenance. The majority of current systems contain software which can be updated to implement new features or to solve problems. The invariance property assures that only the modified component is affected.

Adding components one by one to a system is a common method to compose a system. The growth property guarantees that the existing components do not need to be reverified, since their behaviour does not change. This principle is important for the service guarantees of a shared resource. The resource must guarantee that all requestors still get their allocated service, even when all requestors ask their maximum share simultaneously. Note that this is closely related with the invariance principle, because minimum service guarantees are part of the interface description of a resource.

Based on the principles of a composable system, we introduce composable jobs in Definition 2.3. Jobs of an application are a natural choice to be composable because jobs do not have explicit dependencies. Since jobs are mapped to components, a component must still assure that the composable jobs are not affected by others (by Definition 2.4).

Composable components are harder to create because they have to comply with more requirements. Being independent of others also means that a component cannot take advantage of the behaviour of other components. When a component requests less service than it has allocated, it is not allowed that others use the remaining service. Hence, composable components may have lower average performance than non-composable components. Memory controllers are often the bottleneck of a system [11], therefore this is a major disadvantage for systems that have very different average and worst-case performance.

Definition 2.3 *A job is* **composable** *when its behaviour is not implictly affected by other jobs.*

Definition 2.4 *A* **component** *is composable when the tasks that it executes for a specific job does not implictly affect the behaviour of other jobs that use the component .*

A job that make use of the memory controller is mapped to one or more requestors. According to Definition 2.4, the behaviour of a task may not affect tasks of other jobs. Using the behaviour as described by Section 2.1.2, the requirements for a composable memory controller are the following:

- The latency of a memory access may not be affected by memory accesses of other requestors
- The net bandwidth of a requestor may not be affected by memory accesses of other requestors

2.2 Secondary requirements

Besides the predictability and composability requirements, there are secondary requirements, mainly related to the practical use and flexibility.

2.2.1 Configurable

To reduce the cost of a chip, one can save on hardware by sharing. A lot of tasks are never executed at the same time, but have common hardware demands (like cpu time, memory service or arithmetic units). Tasks that never run concurrently could potentially use the same hardware. However, each task has its own service requirements. The memory controller must have a configurable service per requestor to satisfy a wide range of requirements. Most requirements can be satisfied when minimum net bandwidth and maximum latency of a request can be guaranteed. It is possible that service requirements of a requestor change during run time. Service for the most demanding requestor must be allocated when the service is only programmable at design time. Such over-allocation is avoided when service can be (re)configured at run time. The memory controller can be reconfigured at the time a requestor requires less service, thereby allowing others to receive more service.

As explained in Section 2.1.2, all tasks of a predictable system are predictable, and therefore need a predictable memory controller. Configurable predictability is not necessary. On the other hand, not all requestors require a composable memory controller, because a system can consist of composable and non-composable jobs. To improve performance and reduce hardware, it must be possible to disable composability at design-time.

2.2.2 Performance

Currently, memories are often the bottleneck of embedded systems [7]. To get a usable and cost effective memory controller, it should be able to use the memory as efficiently as possible. Basically, this means that it should offer high net bandwidth and low latency. However, performance may be sacrificed for allowing a predictable and composable solution.

2.2.3 Reusable

The last requirement is that the memory controller must be easy to reuse. The primary target for the memory controller is the CoMPSoC platform template, which uses a NoC as interconnect. To support a wider range of SoCs, the requestors interface must also support buses (i.e. AXI, AHB). In addition, the memory interface must be universal enough to support the common external memories (i.e. SDRAM, DDR, DDR2,DDR3, LPDDR). Besides reusability, a universal interface makes it possible to draw more general conclusions, because it covers more SoC architectures.

2.3 Conclusions

The primary requirements of the memory controller are predictability and composability. A predictable component has bounded behaviour. This allows real-time requirements to be verified at design time. For the memory controller, the latency of a request must be bounded and a minimum net bandwidth must be guaranteed for some interval. A composable system helps reducing the integration effort by eliminating the need of verifying components for every integration step. It requires composable components, because they do not implicitly affect each others behaviour. The requirement for the memory controller is that the behaviour towards a requestor may not change because of some other requestor. Besides good performance, the memory controller must have enough functionality and flexibility, such that it can be used for a wide range of applications.

3 Related work

Section 1 and 2 motivated why a predictable and composable memory controller is desired. This section gives an overview of research that is related to this topic and motivates why a new design is necessary.

Memory controllers that are aware of Quality of Service (QoS) exist, but often have no proof that they can guarantee a minimum net bandwidth or maximum latency of a request [9, 10, 16, 5, 18, 27, 17]. These guarantees are necessary to be predictable and guarantee real-time requirements. A step towards a predictable system is the memory controller in [18]. This memory controller uses a private virtual time memory system (VTMS) for every requestor. SDRAM timing details can be abstracted by this method such that a simpler model can be used to analyse the timing behaviour of the memory controller. The front-end proposed in this report maps abstract service time and cycles to memory accesses for the same purpose (Section 7.2.3).

Typically, memory dependent information is used by a scheduler to improve efficiency [9, 10, 16, 6]. The disadvantage is that the memory controller is tightly bound to a specific memory technology, making it more difficult to use for different memories (i.e. SRAM).

The basic functionality and environment of Sonics MemMax memory scheduler [27] is similar to the front-end proposed by this report (Figure 7.1). Requests enter the memory scheduler (front-end) from a predictable interconnect. The memory scheduler decides which request can be sent to the DRAM controller (back-end) based on efficiency and real-time requirements. Requests are scheduled rather than SDRAM commands to avoid being dependent on the memory technology. However, it is not known if it is predictable since the architecture of MemMax is not available.

The Embedded Hardware Manager (EHM) of [17] does not have a specific memory interface, such that it also can be used for other resources like interconnects and CPU. The resource controllers (like an SDRAM memory controller) are embedded in the EHM as separate blocks. The EHM can only guarantee gross bandwidth, such that hard real-time requirements cannot be verified.

The two memory controllers Columbus [24] and Spider [21] use fixed SDRAM command patterns for accessing the memory such that the timing characteristics of an access is known and bounded at design time. Furthermore, a requestor is granted access to the memory by a static scheduling scheme. The disadvantage of these static schedulers is that maximum latency is coupled to the allocated bandwidth, allocation granularity is coupled to latency and a large number of schedules need to be stored if there are many use cases.

A more flexible memory controller is Predator [4, 19]. It uses the Credit-Controller Static-Priority (CCSP) arbiter [2] to allow dynamic scheduling of requests. Like Columbus and Spider, it uses fixed command patterns for a memory access and guarantees minimum net bandwidth and maximum latency for each requestor. However, Predator

is not composable.

During the literature study, no memory controller has been found that claims to be composable or uses a different methodology to reduce verification and integration effort.

3.1 Conclusions

Current proposals for memory controller architectures focus primarily on (average) performance, often in spite of predictability. Only a few can guarantee minimum net bandwidth and maximum latency, which is crucial for real-time systems. Most memory controllers try to use the memory efficiently by taking advantage of memory dependent information. However, this restricts the use of the memory controller to specific types of memory. No memory controller has been found that has the goal to reduce verification and integration effort.

4 SDRAM

The target memory of the memory controller is Synchronous Dynamic RAM (SDRAM). This type of memory is cheap and relatively fast. This section discusses the architecture, how the memory is used, and shows an efficiency model.

4.1 History

In the first computer systems, most data storage was done by systems based on magnetic properties of materials, often in the form of tapes, discs or drums. A common problem is that the read/write head should be positioned at the right location of the material (or the material moved to the head) before the location can be read or written. Those data storage systems have the property that the time to access different locations is not constant and could depend on the previous access. For example, an access at the beginning of a tape followed by an access to the end of the tape costs more time than two accesses at the beginning. With the use of transistors, Random Access Memory (RAM) was developed. Random access means that the time of an access does not depend on the location or previous access. This memory is perfectly predictable, since the access time does not depend on the traffic. Current SDRAM devices are not truly random access anymore, resulting in non-constant access times. However, the execution time of a single access can still be bounded but is high compared to the best case. Section 6 discusses this issue in more detail.

4.2 Architecture

The data of an SDRAM device is stored in a collection of memory cells. Each cell contains as many bits as the width of the data bus. They are organized in banks, rows and columns. Figure 4.1 shows the banks of an SDRAM device. A bank can be seen as a table of rows and columns (Figure 4.2). Besides the memory cell array, each bank also has a row buffer (sense amplifier). The interface of an SDRAM consists of a command bus and a bidirectional data bus. Different operations can be initiated by sending the commands. The data bus is used to write to and read data from the memory cells.

4.3 Accessing memory

The memory cells cannot be accessed directly. It is only possible to read from or write to the row buffer. In addition, it is not possible to access a single cell. The memory is accessed by a *burst* of reads or write operations. The burst length determines the minimum amount of cells that are accessed in the row buffer. It can be programmed in

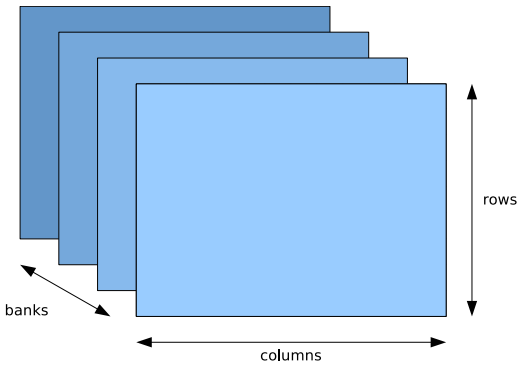


Figure 4.1: SDRAM architecture

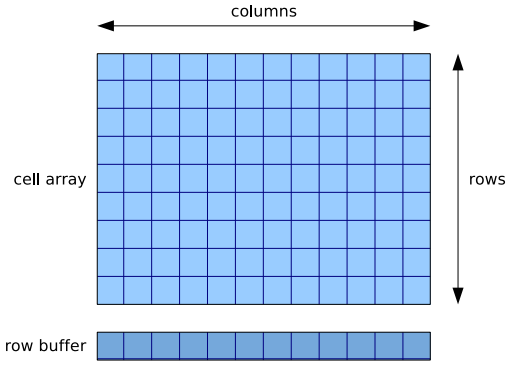


Figure 4.2: Bank architecture

a register of the memory device. DDR2 supports a burst length of 4 or 8 words [12], DDR3 only supports 8 word bursts [13]. The following steps have to be executed for a single burst:

- *Activate row*: First, the row containing the cells to read or write has to be activated. The memory copies the values of that row to the row buffer. This action basically destroys the values in the cells.

- *Read/write column*: After the activation of the row, the value of the column can be read from the row buffer and transferred on the data bus. In case of a write operation, the value in the buffer that corresponds to the column to write to is replaced by the value on the data bus.

- *Precharge row*: The final step is to precharge the row. The complete row in the row buffer is copied to the right position in the cell array. Since the contents of the activated row of the cell array has been destroyed, precharging is necessary for both read and write operations.

No data can be transferred during the activation and precharging of a row. To improve the performance, the fast page mode has been introduced. When one performs more accesses on the same row it is only needed to activate the row for the first access and precharge after the last access for that row. Multiple banks can also be used to improve performance. Because each bank has its own row buffer, the time to activate a row of a certain bank can be (partially) hidden by reading or writing from another bank. This is illustrated in Figure 4.3. First, bank 1 is activated to read some data. At the time the read data is transferred (the read period), bank 3 is activated. Likewise, bank 2 and 4 are used to hide activate and precharge periods. When both techniques are applied, a significantly higher bandwidth and lower latency can be achieved [20].

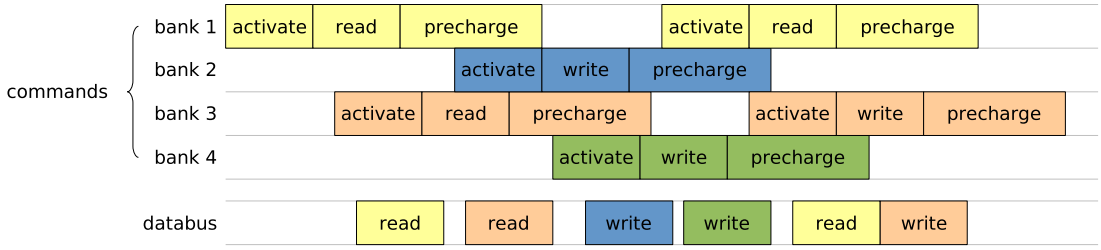


Figure 4.3: Bank interleaving

The time for performing a burst is not necessarily constant, because rows are not always precharged and activated before the burst can be executed as mentioned in Section 4.3. However, the timing behaviour can be bounded at design time, since the timing of all SDRAM commands is constant. According to Definition 2.2, this means that an SDRAM device is a predictable component.

4.4 Refreshing memory

Besides accessing the memory, another vital function is refreshing the memory. Each memory cell of an SDRAM consist of a capacitor that 'remembers' the value of one bit. However, the charge of the capacitors decreases due to leakage. To assure the data is retained, they must be recharged once in a while. This action is called a refresh. The following steps are needed to perform a refresh:

- *Precharge banks*: The activated rows of all the banks must be precharged to write any possible change back to the memory cells. The refresh command can now use the row buffer without destroying data.

- *Refresh*: Now that all banks are idle, the refresh command can be called. The memory device contains a mechanism that determines what row to refresh, such that all capacitors are recharged before they are depleted.

4.5 Commands

To perform a function, different commands can be sent to the memory. The most important ones are listed below.

- *ACT*: Activates a row of a bank.
- *RD*: Performs one burst of reads starting at a certain column. Optionally, the row of the burst can be precharged automatically.
- *WR*: Performs one burst of writes starting at a certain column. Optionally, the row of the burst can be precharged automatically.
- *PRE*: Precharges the active row of a bank.
- *PREA*: Precharges the active row of all banks.
- *REF*: Refreshes some rows of the memory.

For SDRAM, a command can be issued every cycle. However, most commands have certain timing constraints, such that they cannot be issued at any time. These timing constraints depend on the type of SDRAM. To avoid being dependent on the type of memory device, we introduce abstract timing constraints in Table 4.1. This table only contains the constraints that are used to construct sequences of commands. It does not include rules to prevent illegal actions like issuing a REF command when not all banks of the memory have been precharged.

Figure 4.4 shows some constraints for a sequence of commands. The number appended to the command denotes the bank to target. Appendix A contains timing constraints for a DDR2-400, DDR2-800, DDR3-800 and DDR3-1600 memory device.

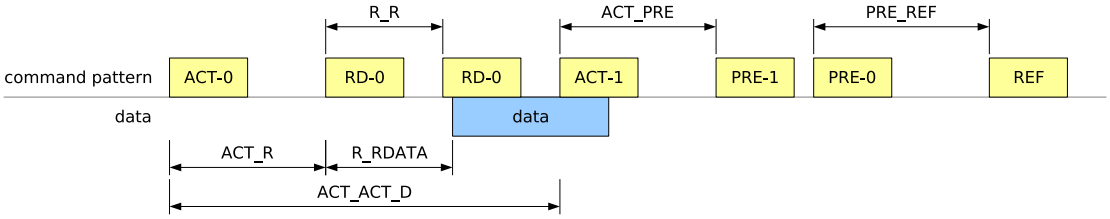


Figure 4.4: Timing constraints of SDRAM commands

4.6 Efficiency

In this section, the efficiency model of Woltjer [28] is presented. More details can be found in the thesis of Woltjer. Section 6 uses this model to evaluate suitable access methods for a predictable memory controller.

Memory efficiency is a measure that indicates the fraction of time that there is data on the data bus. The total number of cycles the memory has run can be split up into data cycles and lost cycles. Data cycles represent the cycles where useful data is read

Table 4.1: Command timing constraints

identifier	*description*
ACT_R	minimum time between start of ACT command and start of first RD command (same bank)
ACT_W	minimum time between start of ACT command and start of first WR command (same bank)
4ACT_ACT	period containing maximal four activate commands
ACT_ACT	minimum time between start of ACT command and the start of the next ACT command (same bank)
ACT_ACT_D	minimum time between start of ACT command of a bank and start of ACT command (different bank)
ACT_PRE	minimum time between start of ACT command and start of PRE command (same bank)
R_PRE	minimum time between start of a RD command and start of PRE command (same bank)
W_PRE	minimum time between start of a WR command and start of PRE command (same bank)
PRE_ACT	minimum time between start of a PRE command and start of an ACT command (same bank)
R_ACT	minimum time between start of a RD command and start of ACT command (same bank)
W_ACT	minimum time between start of a WR command and start of ACT command (same bank)
PRE_REF	minimum time between start of a PRE command and start of a REF command
REF_ACT	minimum time between start of a REF command and start of a ACT command
R_R	minimum time between start of RD command and start of a next RD command (any bank combination)
W_W	minimum time between start of WR command and start of a next WR command (any bank combination)
R_W	minimum time between start of RD command and start of a WR command (any bank combination)
W_R	minimum time between start of WR command and start of a RD command (any bank combination)
R_RDATA	time between start of RD command and start of first data cycle of that command on the bus
W_WDATA	time between start of WR command and start of first data cycle of that command on the bus

from, or written to the memory (Figure 4.3). All other activities, such as precharging and activating, are part of the lost cycles. The definition of memory efficiency is then

written as:

$$\eta = \frac{dataCycles}{totalCycles} = \frac{totalCycles - lostCycles}{totalCycles}$$

The gross bandwidth is the bandwidth when there are no lost cycles. For a DDR2-800 device using an 8 bit data bus this is 800 MB/s. However, this requires conditions which are never possible in practice. The actual available bandwidth is the net bandwidth, defined as:

$$netBandwidth = \eta \cdot grossBandwidth$$

Woltjers model distinguishes between the following efficiencies: refresh efficiency, data efficiency, bank efficiency, read/write efficiency and command efficiency, which are explained in the next sections.

4.6.1 Refresh efficiency

During a refresh operation, no data can be read or written which results in lost cycles. The maximum time between the refresh operations depends on the memory device and operation temperature. The average refresh period is specified as t_{REFI} and the time needed for a refresh is t_{RFC}. The maximum time between two refresh commands is $9 \cdot t_{REFI}$ for DDR, DDR2 and DDR3 memory devices. After waiting the maximum time, 8 consecutive refresh commands have to be issued to comply with the average period. Compared to a fixed refresh period the execution time of refresh commands can be altered somewhat to increase the efficiency. The time to perform a complete refresh is: $t_{RFC} + t_{prechargeAll}$, where $t_{prechargeAll}$ is the time to get all banks precharged. The formula to compute the refresh efficiency is shown below:

$$\eta_{refresh} = 1 - \frac{1}{t_{REFI}} \cdot (t_{RFC} + t_{prechargeAll})$$

4.6.2 Data efficiency

The data efficiency is a measure for the relation between required data and actual transferred data. It is not possible to get an arbitrary amount of data from any location without transferring extra data. There are two factors that determine this efficiency:

- *Minimum burst size of the memory*: The burst length and word size of SDRAM devices determines the smallest amount of data that can be accessed.
- *Size and alignment of the request*: A burst must be aligned to the minimum burst size.

Efficiency drops when the size of the request is not a multiple of the minimum burst size, or if the a request is not aligned with the burst size. A small burst size is desirable for high data efficiency. Data efficiency is considered to be the responsibility of the requestor. However, the access method of the memory controller can make it hard for requestors to maintain high data efficiency (explained in Section 6.3.5.3, page 39).

4.6.3 Bank efficiency

Due to the organization of a memory and fast page mode, not every cell can be accessed in the same amount of time. Based on the previous access, there are three different accesses possible:

- *Access a cell on the same row*: The cell can be accessed immediately, since the right row is already activated
- *Access a cell on a different bank*: The cell can be accessed immediately only when the row is activated on that bank. While accessing a bank, the unused banks can activate the row for the next access. There are no lost cycles when the precharging and activating is finished before a bank is used again. However, the bank and row of the next access must be known to be able to activate it.
- *Access a cell on a different row in the same bank*: The current row must be precharged and the next row be activated. After those operations, the cell can be accessed.

The last event is called a *bank conflict* and must be avoided to maintain high efficiency. The worst-case efficiency can be calculated by assuming that every access causes a bank conflict. An example of this is shown in Section 6.1.

4.6.4 Read/write switching efficiency

The memory controller is connected by a bidirectional data bus to the memory. This bus is used for reading and writing. When the memory controller sends a read command followed by a write or vice versa, the bus has to change its direction resulting in a *read/write conflict*. Changing the direction of the bus costs time and therefore results in lost cycles; no data can be transferred. The read/write efficiency is a measure for this loss of cycles.

4.6.5 Command efficiency

The memory receives commands from the memory controller on the command bus. This bus includes lines for the column, row and command type. Only one command per cycle can be represented by this bus. A *command conflict* is a situation where more than one command should be posted to prevent loss of cycles. This can happen when for example a read command on bank 1 is issued and an activate on bank 2 should be issued to get that bank ready for the next command. The burst length determines how many words are read or written for one command. Let the burst length be 8 words and assume that the efficiency is 100% meaning that the full bandwidth is used. Since two words can be transferred per clock cycle, each 4 cycles a read or write command must be sent to the memory. The other 3 cycles can be used to activate and precharge commands for the other banks. However, when the burst length is decreased to 2 words, the memory controller has to send a command each cycle. No room for other commands is left. As explained in Section 4.6.2, a smaller burst size increases the data efficiency. But as noted above, the command efficiency decreases. Also the other efficiencies are correlated, which makes it hard to optimize the performance of a memory controller.

4.7 Conclusions

This section introduced the basic architecture of SDRAM and explained how they are accessed and refreshed. Commands are used to control the memory and have specific timing constraints. The efficiency of the memory depends on which commands are sent to the memory and can be modelled by the presented efficiency model. All this background information is used to obtain an efficient way to access the memory.

5 Memory controllers

An external memory is shared by multiple IP components to reduce the cost of a SoC. A shared memory also enables transferring large amounts of data between IP components (like the system design case in [25]). A memory controller is necessary to share the memory, since it cannot serve multiple IP components by itself. In addition, the memory controller provides a simple interface for the IP components, such that they do not have to be aware of memory details. Section 5.1 explains which tasks a memory controller has to perform and Section 5.2 shows how the tasks fit into a general architecture that is used as a model for the design of the memory controller.

5.1 Tasks

The basic tasks of a memory controller are request scheduling, memory mapping, command generation and memory management. This section briefly discusses these tasks.

5.1.1 Request scheduling

Except for some special types of memory like Dual Port RAM (DPRAM), the memory only allows one access at a time. The memory controller schedules one request at a time, such that it can be executed by the memory. The way requests are scheduled depends on the goals of the memory controller. Common goals are: satisfying service requirements, high memory efficiency, and low power consumption. All kinds of information is used to schedule a request: properties of the request, history of a requestor, and the state of the memory. Examples of information that can be used are:

- *Different requestor*: It is important to know which requestors have waiting requests to satisfy service requirements. Latency and bandwidth bounds may not be violated.
- *Read and write requests*: Memory efficiency can be improved by scheduling groups of reads and writes. This reduces the number of read/write switches.
- *Physical location*: Reducing the number of bank activations, or exploiting bank interleaving is used to increase memory efficiency, but also reduces power consumption. Requests can be scheduled based on the physical address. Grouping requests that want to access the same row is more efficient. Selecting a request with a different bank than the previous access can hide bank activation and precharging.

Schedulers that execute a fixed schedule, determined at design-time, are referred to as static schedulers. An example is Time Division Multiplexing (TDM). Time is divided into slots of a fixed length. The slots are distributed over the requestors at design

time. More slots are assigned to requestors that have higher bandwidth requirements. The advantage is that the timing behaviour of such scheduler can easily be determined and is known at design time, which makes it predictable. Dynamic schedulers do not schedule at design-time. Whereas unused slots of a static scheduler are lost, a dynamic scheduler can assign unused slots to other requestors at run-time to improve bandwidth and latency.

5.1.2 Memory mapping

Requestors are not aware of the architecture of the memory. From their perspective, the memory can be a one dimensional array of elements (not to be confused with memory cells). The size of an element depends on the protocol of the requestor. A request that is issued by a requestor contains a *logical address* to specify the location of the first element. The *size* of the request denotes the number of elements to access. Figure 5.1 shows an example of a request that accesses a memory of 32 elements.

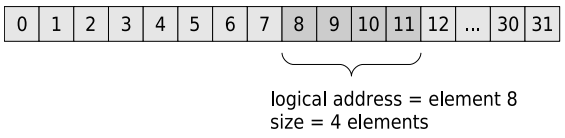


Figure 5.1: The memory as perceived by a requestor

The cells of a memory are identified by the *physical address*. For SDRAM devices this address consists of the *bank*, *row* and *column*. Memory controllers that are connected to multiple memories, extend the physical address by a *rank* to identify the memory device. The purpose of memory mapping is to translate the logical address and size to the physical address and number of cells respectively. There are different kinds of memory mapping schemes with different properties. Figures 5.2 and 5.3 show two memory maps for an example memory that has 4 columns per row, 2 rows per bank and 4 banks. No rank is needed, because only one memory device is used. For simplicity, the element of a request is assumed to have the same size as a memory cell. Figures 5.2(a) and 5.3(a) show how the physical address can be derived from the logical address. A representation of the memory cell arrays are shown by Figures 5.2(b) and 5.3(b). The number in the memory cells refers to the logical address. Consider the sequential memory map in Figure 5.2. The example request in Figure 5.1 that accesses elements 8 to 11 is mapped to columns 0 to 3 of row 0 in bank 1. This mapping has the advantage that a minimum amount of rows needs to be accessed, effectively reducing the number of row activations and precharges.

The difference between the sequential and bank interleaving memory map is that the bank index has been moved to the least significant bits of the logical address. The elements of the example request are mapped to memory cells of all four banks. While accessing row 0 in bank 0 for the first cell, the memory controller can already activate bank 1 for the next cell to hide the activation time of the second access. When the size of all requests is a multiple of the number of banks (in terms of memory cells), bank

interleaving can always be applied. This access method is used by the proposed memory controller (in Section 6.3).

Until now, we assumed that the requestor interprets the memory as a one dimensional array of elements. However, applications like image or video processors could profit from a two dimensional interpretation of the memory cells. More details about other memory maps can be found in [28].

5.1.3 Command generation

When the logical address of a request has been mapped to the physical address, the SDRAM commands can be generated. In addition, refresh commands are issued once in a while to assure that the data in the memory is retained. The command generator has to be aware of the state of the memory for two reasons. First, the commands may not violate command timing constraints in Table 4.1. Secondly, the state of the memory is essential for optimization techniques. When a request wants to access a row that already has been activated, it is not necessary to precharge and reactivate. The command generator has to shadow the state of the memory, since SDRAM devices do not provide an interface to obtain the state. The commands that are sent to the memory determine the state. As mentioned in Section 5.1.1, the request scheduler also may need access to the shadowed memory state.

5.1.4 Memory management

To guarantee proper behaviour of the memory, certain management tasks have to be carried out. For SDRAM devices these include:

- *Initialization*: A power-up and initialization sequence is necessary before an SDRAM device can be used.
- *Refreshing*: The memory needs to be refreshed periodically to retain the data
- *Configuration*: The memory has registers that must be programmed (like the burst length of a read or write command)
- *Powering down*: When the memory is not used it can be powered down to save energy

Only memory refreshes are regarded in this report. The other tasks are not the primary target for this research, are memory specific, and do not occur during normal operation.

5.2 Architecture

Figure 5.4 shows the general structure of a memory controller. This is an abstract architecture that represents a wide range of memory controllers. It does not fix the sequence of common tasks like memory mapping, arbitration and command generation.

The requestors are connected to the memory by means of an interconnect like direct connections, a bus, or a network. The figure shows a single memory device, however, some memory controllers are connected to multiple memories to increase parallelism. Communication from the interconnect, through the memory controller to the memory is done by requests and responses. The format and information of a request or response varies.

A front-end and back-end can be identified within a memory controller. The front-end performs the memory independent tasks. The back-end is responsible for memory dependent tasks. The arbiter can be part of the front-end, back-end, or both. Arbiters that schedule SDRAM commands are mainly part of the back-end [20]. The two stage scheduler in [9] is an example of an arbiter that is part of both.

5.2.1 Requestor interfaces

All incoming requests of a requestor arrive at its requestor interface. Requestors can use different interfaces and protocols. The request contains the type of access (read or write), the amount of data to read or write, the address of the memory cells to access and write data (for a write request). A requestor interface ensures that the requests of the requestors are translated to a single usable format for the arbiter. This allows the use of a protocol independent arbiter. The response received from the arbiter is sent back by the interconnect to the requestor. It represents the result of the request, containing read data (for a read request), and optionally an indication whether the operation succeeded or not.

5.2.2 Arbiter

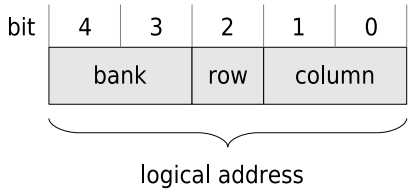
The streams of requests coming from the requestor interfaces are used by the arbiter for scheduling. It selects a request and sends it to the memory interface. The responses returned by the memory interface are converted to the right format and routed back to the corresponding requestor interface. The main task of the arbiter is request scheduling as discussed in Section 5.1.1. However, memory mapping and command generation could also be necessary when this information is used to schedule the requests. Besides sending requests to the memory interface, the arbiter can also instruct the memory interface to perform a refresh.

5.2.3 Memory interface

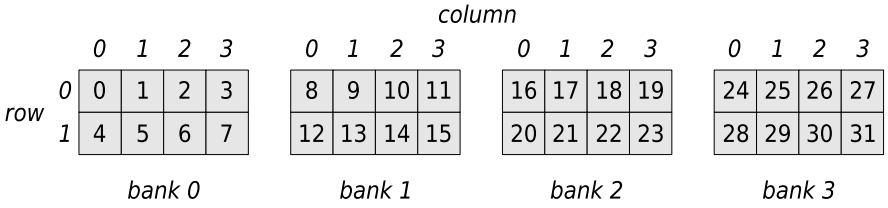
Communication to the memory is performed by sending appropriate SDRAM commands, sending write data and receiving read data. For every memory device connected to the memory controller, there has to be a memory interface. This block accepts a single stream of requests from the arbiter. The read data, and possibly parts of the request, are packed into a response and sent back to the arbiter. Memory mapping and command generation are only performed when it has not already be done by the arbiter. Finally, memory management is the responsibility of the memory interface.

5.3 Conclusions

The common tasks of a memory controller are request scheduling, memory mapping, command generation and memory management. These tasks are mapped to the architecture of the memory controller. The general architecture that is proposed in this section, is formed by requestor interfaces, an arbiter, and a memory interface. Incoming requests are decoded by the requestor interfaces and passed to the arbiter that schedules the requests. The memory interface sends the request (SDRAM commands and data) to the memory device. Data returned by the memory travels the backward path until it reaches the corresponding requestor.

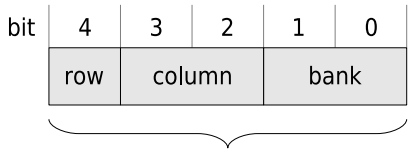


(a) Address translation

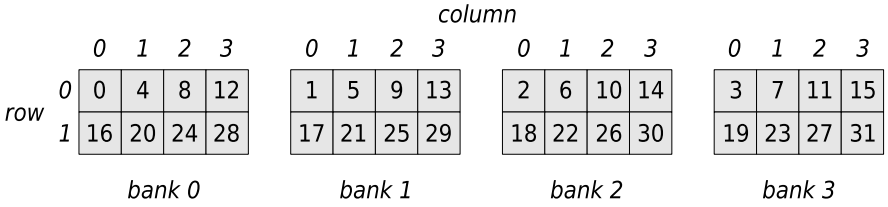


(b) Mapping of a logical address to the memory cells

Figure 5.2: Sequential memory map



(a) Address translation



(b) Mapping of a logical address to the memory cells

Figure 5.3: Bank interleaving memory map

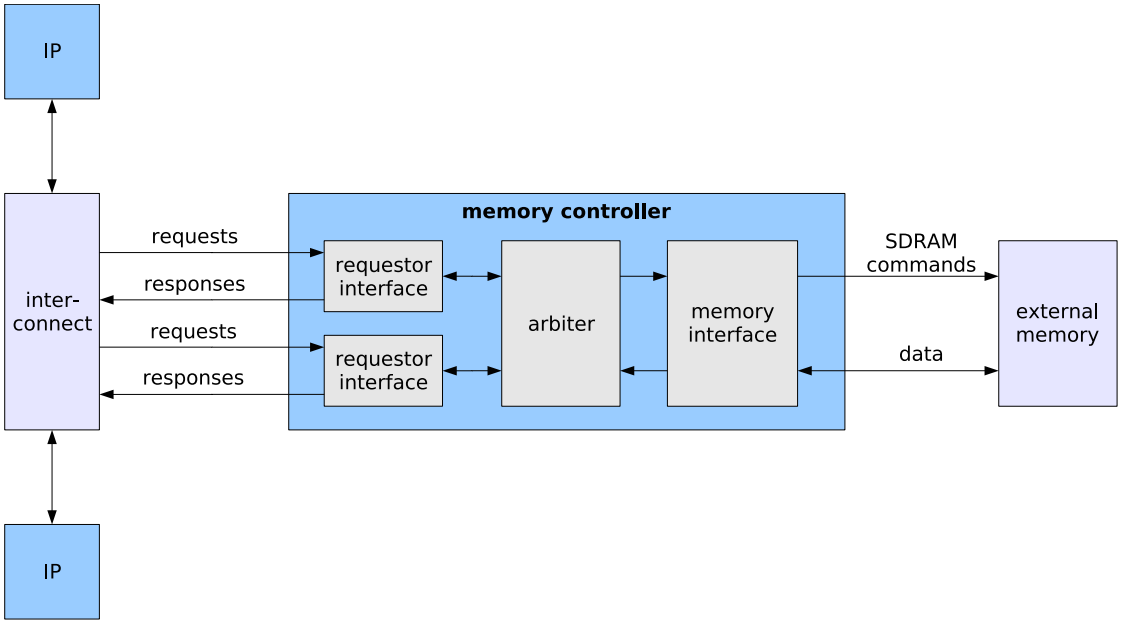


Figure 5.4: General structure of a memory controller

6 Memory command patterns

This section presents an abstract interface between the front-end and back-end. This interface only accepts read, write and idle operations. In addition it is composable and predictable. This abstract interface is essential for a simple but predictable and composable front-end. Multiple levels of abstractions are introduced for the data and time domain. We advice to use Appendix E as a reference for an overview of all abstractions.

First, Section 6.1 discusses the problem of bounding the behaviour of the memory. This leads to the introduction of memory accesses in Section 6.2. They represent the basic operations of the proposed interface. Section 6.3 discusses memory command patterns that have the purpose to group SDRAM commands. A predictable mapping from the basic operations of the interface to the memory command patterns are explained in Section 6.4.

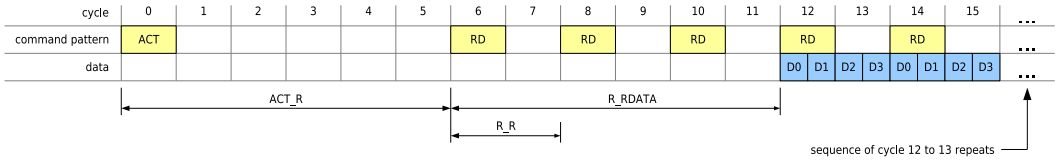
6.1 Predictable memory

As mentioned in Section 2.1.2, a predictable memory is required for a predictable memory controller. By nature, SDRAM devices are predictable, since all commands have bounded behaviour (Section 4). However, there is a big difference between the worst-case and best-case execution time of a single burst. Consider a DDR2-800 device with the properties and timing constraints listed in Table 6.1. For simplicity, we assume that the memory is read-only and does not need to be refreshed. The gross bandwidth for this device is 800 MB/s. The best case for this device occurs when all bursts target the same row (Figure 6.1(a)). A row has to be activated for the first read burst. For the consecutive bursts, there is no need to activate or precharge banks, because the row they want to access already has been activated. The command sequence for the worst case is illustrated in Figure 6.1(b). In this case, all bursts target a different row in the same bank. After every burst, the current bank must be precharged and the next bank activated. In the worst-case there are 20 data cycles for the interval $[0, 239]$. This results in an efficiency of $100\% \cdot \frac{20}{240} = 8.3\%$ or a net bandwidth of 67 MB/s. The best-case has 228 data cycles for the same interval and therefore a much higher efficiency: $100\% \cdot \frac{228}{240} = 95\%$ (760 MB/s). Unfortunately, the worst-case net bandwidth determines the minimum guaranteed bandwidth. For real-time systems with high bandwidth requirements, a memory with a very high gross bandwidth is needed, because the worst-case efficiency of SDRAM devices is very low. An over-dimensioned memory (i.e. higher frequency, wider data bus) has a higher cost in terms of power and area. In addition, the design and verification effort increase, because timing constraints are tighter for higher frequencies.

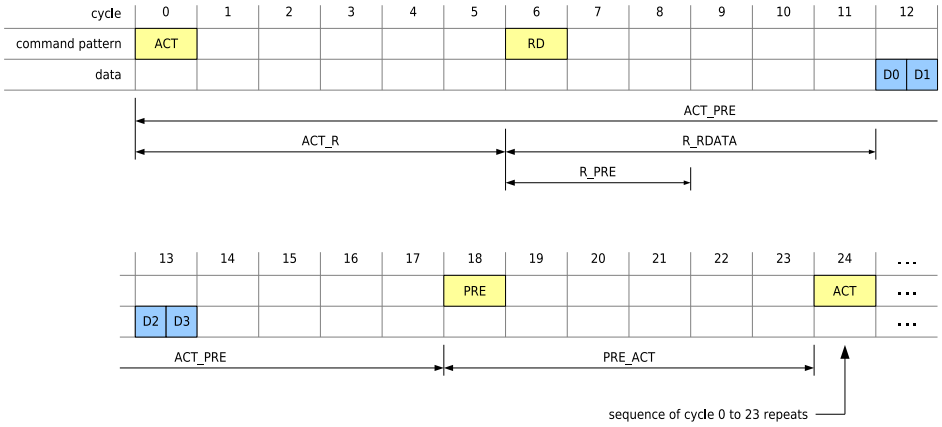
Most modern memory controllers, exploit run-time information to improve efficiency and latency. The memory access scheduler of Rixner [20] is an example of such an architecture. This scheduler has a precharge manager and row arbiter for every bank

Table 6.1: Properties of a DDR2-800 memory device

property	*value*
Clock frequency	400 MHz
Width of data bus	8 bits
Burst length	4
ACT_ACT	24 cycles
ACT_PRE	18 cycles
ACT_RD	6 cycles
R_PRE	3 cycles
R_R	2 cycles
PRE_ACT	6 cycles
R_RDATA	6 cycles



(a) Best-case



(b) Worst-case

Figure 6.1: Sequences for executing read bursts

and a single column and address arbiter. Rixner proposes several policies that can be used by the managers and arbiters. The precharge manager could use the open policy for example. In this case, a bank is only precharged if there are waiting bursts that want to access another row of the bank, and there is no request for the current row anymore. This policy is suitable when there is a high probability that a burst accesses the same row as the preceding one. The sequential memory map (Figure 5.2) is useful to exploit this. However, such techniques have two disadvantages for a predictable memory controller. First, only the average case execution time and latency is improved. Secondly, the policies

make use of the traffic and state of the memory. Without pessimistic assumptions (i.e. always a bank conflict), it is hard or impossible to perform static timing analysis and derive bounds at design time.

6.2 Memory access

The smallest data unit of a memory is the memory cell. As explained earlier, they can only be accessed by a burst. The length of a burst defines the access granularity of an SDRAM device (Section 4). It is the smallest amount of data that can be accessed. The predictable memory controller does not execute single bursts, because this is not efficient (Figure 6.1(b)). A group of bursts is executed for a single read or write operation instead. These groups are called *memory accesses* as defined by Definition 6.1.

Definition 6.1 *A* **memory access** *is the smallest subpartition of a read or write operation of a memory controller.*

The amount of data that is read or written by a memory access must be aligned with, and a multiple of, a memory burst. Figure 6.2 shows the partitioning of the memory space in memory cells, bursts and memory accesses. The length of a burst is two cells and a memory access consists of two bursts. Requests are mapped on top of the memory accesses. One or more memory accesses has to be executed for a request. It is not possible to perform a part of a memory access.

The access granularity of the memory controller is therefore the size of a memory access. The size is constant and determined at design time. A memory access consisting of many bursts results in a coarse granularity. A fine granularity requires a memory access of a small amount of bursts. Unnecessary data is accessed when the address of a request is not aligned with the memory accesses (Figure 6.2(b)) or the size of a request is not a multiple of a memory access (Figure 6.2(c)).

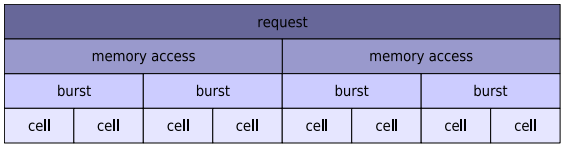
A memory access does not only represent an amount of data, but also refers to the operation that accesses that data. We distinguish between the following operations:

- *Read access*: The operation that reads from the memory
- *Write access*: The operation that writes to the memory
- *Idle access*: The operation that does not access any data

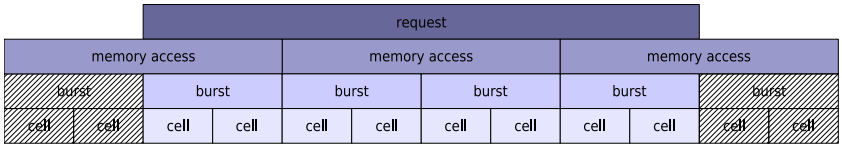
The idle access is used to define a period where no data has to be accessed such that every single point in time can be mapped to an memory access. More details are discussed in Section 6.4.

6.3 Memory command patterns

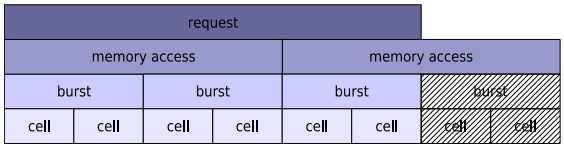
To hide details of SDRAM commands, we use *memory command patterns* to group a sequence of SDRAM commands according to a more abstract operation (Definition 6.2). The proposed memory controller uses memory command patterns that are fixed at run time. This greatly reduces the effort of timing analysis as will be explained in Section 7.



(a) Aligned request



(b) Unaligned request



(c) Request that is not a multiple of memory accesses

Figure 6.2: Memory partitioning and request alignment

Definition 6.2 *A* **memory command pattern** *is a sequence of pairs containing an SDRAM command and the execution time of the command relative to the start of the pattern. The length of a memory command pattern defines the total execution time.*

There are two restrictions for the use of memory command patterns. First, they must be issued back to back and cannot overlap. Every SDRAM command that a memory has to execute must be part of a memory command pattern. Note that it is possible that dependencies between patterns are created, because the timing constraints of a command can affect the next pattern(s). The second restriction is that the patterns cannot be interrupted before they end. Such situations must be modelled by separate patterns.

An example of a memory command pattern that reads from the memory is $< (0, ACT), (3, RD), (5, PRE) >; length = 6$, illustrated by Figure 6.3. Unless mentioned otherwise, the unit of time for a memory command pattern is clock cycles. The cycles without commands are assumed to be NOP commands. Note that the bank, row and column address must be specified before the pattern can be executed by the memory. In the remainder of this report, the suffix of a command specifies the bank (i.e. ACT-2 denotes an activate command for bank 2). Timing constraints have to be satisfied for patterns considered to be valid. Besides reading, patterns can be constructed for operations like writing and refreshing the memory. A sequence of x NOP commands is described by an empty sequence with length of x cycles: $<>, length = x$

The sequence of commands in Figure 6.1(b) suffers primarily from a low bank efficiency resulting in a low bound on net bandwidth. For a realistic read/write memory that must be refreshed, the bound is even lower because read/write switch efficiency and refresh efficiency cannot be assumed to be 100%. No advantage can be made of the opti-

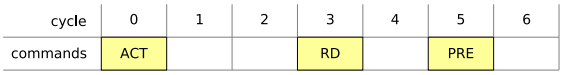


Figure 6.3: Memory command pattern that performs a read operation

mization techniques (i.e. bank interleaving and fast page mode), because at the time the system needs to be verified, the traffic is not known. To improve the guaranteed service, more information about the traffic must be known either by analysis or restrictions. An example is that the traffic always accesses the same row. Using this information, the memory controller never activates and precharges a bank for a single burst.

The Predator memory controller proposed in [19, 4] uses basic groups. A basic group is a memory command pattern that performs a read or write operation. Predator guarantees a higher net bandwidth (82.6% for a DDR2-400 device) than the naïve approach of Section 6.1, because bank interleaving can applied for every memory access. First, it is assumed that the number of bursts in a memory access equals the number of banks. Secondly, a bank interleaving memory map is used, such that each burst targets a different bank.

The next subsections show the patterns that are used by our memory controller. The patterns are derived from the concept of basic groups. Subsection 6.3.4 presents the memory map that is required for these patterns.

6.3.1 Access patterns

The basic groups can be mapped to access patterns which are more general. They are used to read or write all the bursts of a memory access. This pattern is always necessary to perform a memory access. However, more patterns could be required for a memory access as discussed in Section 6.4. The access patterns have three different types: a read pattern, a write pattern and an idle pattern. The idle pattern only consists of NOP commands and is used when no data needs to be accessed. The read and write patterns can be characterized by the following properties:

- *burstLength*: Number of memory cells that are read or written sequentially per burst.
- *interleavedBankCount*: Number of banks that are used in the pattern. This ranges from one to all available banks. When the pattern does not use all banks, the banks are divided in groups. The address of the request determines which group is used.
- *burstCount*: Number of bursts per bank.

The properties and the operations performed by the read and write patterns are illustrated by Figure 6.4(a). As can be seen from the figure, almost all activation and precharge periods are hidden because another bank is executing a read or write burst at that time. The basic groups of Predator are access patterns with a burst length of 4 or 8 and interleave across all banks. One burst is executed per bank. However, these

patterns do not give the highest efficiency for all devices and request sizes. Section 6.3.5 shows efficiency and latency for different devices and patterns.

The templates for the read, write and idle patterns are $< readPatternSequence >; t_{read}$, $< writePatternSequence >; t_{write}$ and $<>; t_{idle}$ respectively. Symbols for these patterns are defined in Table 6.2.

The command sequences are shown in Figures 6.4(b) and 6.4(c). The figure only shows the order of the commands for a bank and the command types correctly. The exact timing of the commands depend on the memory device, because they have different timing constraints for the commands. Appendix B shows the patterns for different devices.

For two reasons, auto precharge is always be applied for the last read or write command of each bank. First, it reduces the amount of commands on the command bus, because no PRE command has to be issued. Hence, it potentially decreases the number of command conflicts. Secondly, it reduces dependencies with successive patterns. This is illustrated by Figure 6.4(a) where the next pattern already started before the last bank (bank i+4) could be precharged. Without auto precharge, the next pattern has to issue the PRE command for that bank.

6.3.2 Switching pattern

Timing constraints between successive access patterns of the same type are satisfied by bank interleaving. However, a read to write switch pattern must be inserted between a read and write pattern. This assures that there is enough time to change the direction of the data bus. When a write is followed by a read, a write to read switch pattern must be inserted. A switching pattern contains only NOP commands. The templates for the read to write and write to read switching patterns are $<>; t_{rtw}$ and $<>; t_{wtr}$, respectively. Recall that NOP commands are not listed in the sequence for clarity. The length of the patterns are be computed by Equations (6.1) and (6.2). It is possible that a combination of device and access pattern has switching patterns of zero length, because the access pattern are that long that timing constraints are always met.

$$t_{rtw} = \max(0, R\_W - (t_{read} - t_{last\_read\_cmd} + t_{first\_write\_cmd})) \tag{6.1}$$

$$t_{wtr} = \max(0, W\_R - (t_{write} - t_{last\_write\_cmd} + t_{first\_read\_cmd})) \tag{6.2}$$

6.3.3 Refresh pattern

The refresh pattern is issued periodically to refresh the memory. It only contains the refresh command. The time before the refresh command is needed to wait until all banks are precharged, and the time after the refresh is needed for performing the actual refresh. Three versions exist, since the time to precharge the bank depends on the previous access pattern. The template for the refresh pattern that is issued after a read pattern is $< (readRefPos, REF) >; t_{read\_ref}$, where $readRefPos$ is computed by Equation (6.4). If the preceding pattern is a write pattern, the template is: $< (writeRefTime, REF) >; t_{write\_ref}$. Equation (6.5) is used to compute $writeRefTime$. When the refresh pattern

Table 6.2: Symbols of the memory command patterns

t_{read}	Length of the read pattern
t_{write}	Length of the write pattern
t_{idle}	Length of the idle pattern
t_{access}	Length of an access pattern (when all have the same length)
t_{rtw}	Length of the read to write switch pattern
t_{wtr}	Length of the write to read switch pattern
t_{read_ref}	Length of the refresh pattern after a read pattern
t_{write_ref}	Length of the refresh pattern after a write pattern
t_{idle_ref}	Length of the refresh pattern after an idle pattern
t_{ref}	Length of the refresh pattern (when all have the same length)
$t_{first\_read\_cmd}$	Time of the first read command of the read pattern
$t_{last\_read\_cmd}$	Time of the last read command of the read pattern
$t_{first\_write\_cmd}$	Time of the first write command of the write pattern
$t_{last\_write\_cmd}$	Time of the last write command of the write pattern

is issued after an idle access, the template is $< (idleRefTime, REF) >; t_{idle\_ref}$. The time of the REF command is computed by Equation (6.6). The length of all patterns are computed by (6.3), where $refTime$ refers to the time that the REF command is issued.

$$t_{ref} = refTime + REF\_ACT \tag{6.3}$$

$$readRefTime = R\_PRE + PRE\_REF - t_{read} + t_{last\_read\_cmd} \tag{6.4}$$

$$writeRefTime = W\_PRE + PRE\_REF - t_{write} + t_{last\_write\_cmd} \tag{6.5}$$

$$idleRefTime = 0 \tag{6.6}$$

6.3.4 Memory map

As explained in Section 5.1.2, the logical address of a request has to be translated to bank, row and column addresses. The access patterns require that bursts access different banks. This can be enforced by using the memory map of Figure 6.5. This memory map is slightly different from the memory map proposed in [4] because it splits the bank address into a low and high part. This is necessary to support access pattern that do not use all banks. The logical address is assumed to be expressed in bytes. The width of the fields are defined in Table 6.3.

The access granularity of the memory controller is determined by this memory map, since it is equal to the size of a memory access (Section 6.2). The granularity and size of a memory access is calculated by Equation (6.7).

$$width_{ra} = interleavedBankCount \cdot burstLength \cdot burstCount \cdot dataWidth \tag{6.7}$$

Table 6.3: Width of the physical addresses (bits)

byte	$log_2(dataWidth/8)$
column low	$log_2(burstCount \cdot burstLength)$
bank low	$log_2(interleavedBankCount)$
column high	$log_2(columnCount) - log_2(burstCount \cdot burstLength)$
bank high	$log_2(bankCount) - log_2(interleavedBankCount)$
row	$log_2(rowCount)$

where:

$bankCount$	Number of banks of the memory
$rowCount$	Number of rows in a bank
$columnCount$	Number of columns in a row
$dataWidth$	Width of the data bus in bits

The memory access address is the part of the logical address that identifies the memory access. The remaining bits identify the burst, memory cell and byte.

6.3.5 Results

In this section, the timing behaviour of the read and write patterns is analysed for DDR2-400, DDR2-800, DDR3-800 and DDR3-1600 devices. The timing constraints can be found in Appendix A. A subset of the patterns are listed in Appendix B.

The memory devices have a data width of 16 bits and a page size of 2KB, unless mentioned otherwise. This page size results in the worst results, because some commands have tighter timing constraints than a device with a page size of 1KB. Request size denotes the amount of data the requestor wants to read or write.

6.3.5.1 Bank efficiency

The effect of bank efficiency on the net bandwidth is shown by Figure 6.6. The most efficient access patterns that we found are shown in the figure. The net bandwidth only comprises bank efficiency of the access pattern. Read/write, command, refresh and data efficiency are assumed to be 100%.

The request size is expected to grow for some applications (like video decoders). However, the size of the requests of CPU's is not likely to change in the future, such that data efficiency may decrease. For an access granularity of 32 bytes (Figure 6.6(a)), the bank efficiency of the access patterns is so low for new devices that they do not provide a higher net bandwidth. The access patterns for a DDR2-400 device in Figure 6.6(b) have a bank efficiency of 100% (these are used by Predator). This figure also shows a rather big difference between the read and write pattern. Write patterns are longer because WR_PRE is in general larger than RD_PRE (Table A.1 on page 137). The minimum guaranteed efficiency cannot be improved because the request type is only known at run time. The DDR3-800 device has a slightly better bank efficiency than its DDR2-800 counterpart. The DDR2-800 and DDR3-800 manage to have a 100% bank efficiency

when a 128 bytes access granularity is used by the memory controller (Figure 6.6(c)). The DDR3-1600 needs at least an access granularity of 256 bytes to get 100% bank efficiency according to Figure 6.6(d). The bank efficiency of the other devices remain 100%, because two memory accesses of 128 bytes (having 100% bank efficiency) can be used to form a memory access of 256 bytes.

6.3.5.2 Granularity trend

The results of Figure 6.6 show that newer devices (or with a higher clock frequency) require a coarse access granularity to get a high bank efficiency. Figure 6.7 illustrates the need to increase the access granularity to maintain bank efficiency. Only read patterns are regarded in this figure. Write patterns have a similar, but a bit more pessimistic trend. The access granularity is an interpolation of the results from Figure 6.6. Furthermore, the results of a virtual DDR4-3200 device are extrapolated. The figure shows that an exponential increase in access granularity is necessary to maintain efficiency for newer devices. Note that this figure does not show the effect of data efficiency. Only when the request granularity has the same trend as the access granularity, the efficiency can be maintained.

6.3.5.3 Data efficiency

From Figure 6.6, it is concluded that a high bandwidth always can be offered by increasing the access granularity. However, it is likely that data efficiency for a memory controller with a coarse access granularity decreases, because too much data is accessed for small requests (Figure 6.2). The actual data efficiency depends on the size and address of the requests (request granularity). Data efficiency is regarded as the responsibility of the requestor, but the memory controller is not usable when request always suffer from very low data efficiency. Note that a low data efficiency is not only bad for the net bandwidth, but also for power consumption, because much work is not used.

Figure 6.8 shows the effect of data efficiency for different combinations of request sizes and access granularities. Requests are assumed to be aligned with the access granularity which means that the presented data efficiency is an upper bound for the requests. Only results for read patterns are shown. The efficiency of those figures includes bank efficiency and data efficiency. All other efficiencies are assumed to be 100%.

In general, the efficiency of the write patterns is equal, or slightly lower, than the read counterpart because they have lower bank efficiency.

For the DDR3-1600 device (Figure 6.8(d)), requests must have at least a size of 128 bytes to get an efficiency above 45%. In fact, the DDR2-400 device of Figure 6.8(a) performs best for smaller requests. Figures 6.8(b) and 6.8(c) show that there is little difference between a DDR2 and DDR3 device running at the same frequency, although the DDR3-800 device has a slightly higher efficiency. From the figures can be seen that there is no read pattern for any memory that can deliver an efficiency of 100% for a 32 bytes request. Read patterns that can reach 100% efficiency have a coarse access granularity and need large requests. However, for all devices, these patterns perform worse for small requests than the patterns that have a tailored access granularity. This

means that the best read pattern are not the ones with the highest bank efficiency by definition. It depends on the request granularity of the requestors.

6.3.5.4 Data latency

Besides efficiency and bandwidth, the latency of the memory is an important timing characteristic of the memory. We define two types of access pattern latency.

- *First data latency*: The time between the start of the access pattern and the cycle that an uninterrupted stream of data can be guaranteed (all cycles must have data)
- *Last data latency*: The time between the start of the access pattern and the cycle after the last data

First and last data latencies of DDR2 and DDR3 devices are shown by Figure 6.9. Latency is only shown for read patterns with the highest bank efficiency. The only difference between the write and read pattern are the R_RDATA and W_WDATA timing constraints, because the write and read commands are issued at the same time. According to Appendix A, the latency of the write patterns is one or two cycles less than the read pattern.

From Figure 6.9(a) can be concluded that the first data latency is not strongly affected by the access granularity. The latency of the DDR2-800 and DDR3-800 are identical. The first data latency does not scale with the clock period of the memory device. The clock period is four times shorter than the DDR2-400, but the latency is only improved by a factor of 1.2. The DDR2-800 and DDR3-800 devices do not perform better than the DDR2-400 device.

Figure 6.9(b) shows that last data latency does improve for newer memories. The last data latency consists of two parts: the latency of the first word and the time between the last and first word. An uninterrupted stream of data is provided after the first data latency according to the definition. This means that the time between the last and first word only depends on the gross bandwidth of the memory. Newer memories have a lower last data latency for larger requests since they have a higher gross bandwidth and the first data latency is almost constant.

6.4 Memory access to pattern map

A *memory access to pattern map* is used to simplify the interface of the memory (Definition 6.3). Only memory accesses can be executed: the *read access*, *write access* and *idle access*. An idle access is executed when no data needs to be read or written. The mapping from memory accesses to patterns depends on the state and access to execute.

Figure 6.10 illustrates that this model has a hierarchy of three levels. The highest level are the memory accesses. Memory accesses are composed of memory command patterns. According to the mapping, the first memory access of Figure 6.10 consist of a read pattern (RD), the second is composed of a read to write switch (RTW), and write pattern (WR). The lowest level are the SDRAM commands that are executed by the memory. The figure shows that the patterns for a read access can be different at run time. Note that all executed patterns are part of a memory access.

Definition 6.3 *A* **memory access to pattern map** *defines which memory command patterns are executed for a memory access (read, write or idle).*

Example 1 *Assume a memory that does not need to be refreshed and has the properties listed in Table 6.1. The following table shows the patterns that are used.*

pattern	length	sequence
RD	24	$< (0, ACT), (6, RD), (18, PRE) >$
WR	24	$< (0, ACT), (6, WR), (18, PRE) >$
SW	6	$<>$
$IDLE$	24	$<>$

The read and write patterns (RD, WR) are constructed in such a way that two successive read patterns or write patterns can be executed back to back without violating timing constraints. A switching pattern (SW) is included to demonstrate that a memory access can consist of multiple patterns. A very simple mapping based on the patterns is shown in the next table:

previous pattern	memory access	patterns
not WR	*read*	*RD*
WR	*read*	*SW RD*
not RD	*write*	*WR*
RD	*write*	*SW WR*
-	*idle*	*IDLE*

The patterns to execute for a memory access depend on the previously issued pattern and the type of memory access to execute. According to this mapping, there are five different sequences of patterns for a memory access. The switch pattern is executed when the type of the previous pattern differs (except for the idle pattern). An idle pattern is executed for an idle access, independent of the previous pattern.

6.4.1 Predictable memory access to pattern map

The *predictable memory access to pattern map* (PAM) is a map that has the purpose to assure predictable behaviour (Definition 6.4). Example 2 discusses a simple PAM.

Definition 6.4 *A* **predictable memory access to pattern map** *is a memory access to pattern map that guarantees bounded behaviour at design time. The behaviour is defined as:*

- Rate*: number of read and write accesses for a bounded real time interval. Idle accesses are not included.*
- Last data latency*: the real time between the start of a memory access and the last data of that access on the data bus*

Example 2 *The minimum rate of Example 1 can be determined by assuming that read and write accesses are interleaved. According to the mapping, the switch pattern is always executed. Hence, the total length of a read or write access is 30 cycles and the rate is bounded by 1/30 memory accesses per cycle. As can be seen from Figure 6.1(b), the last data latency of the read pattern is 13 cycles. To simplify the example, we assume that the write pattern has a shorter latency. The last data latency of a memory access is 19 cycles because a switch pattern of 6 cycles could be inserted before the read pattern. From the bounded rate and last data latency, we conclude that this is a PAM.*

The PAM that is used in the remainder of this report is constructed from the patterns discussed in Sections 6.3.1, 6.3.2, and 6.3.3. To simplify timing analysis, we assume that the length of all access patterns and refresh patterns are equal:

$$t_{access} = t_{read} = t_{write} = t_{idle}$$

$$t_{ref} = t_{read\_ref} = t_{write\_ref} = t_{idle\_ref}$$

The longest access and refresh patterns are used to satisfy the timing constraints. Table 6.4 shows the truth table of the PAM. Figure 6.11 illustrates a sequence of patterns according to this PAM and lists the abbreviations of the patterns. The mapping uses a refresh timer (c), the previous pattern and the type of memory access to determine the patterns. The refresh timer starts at the refresh period ($t_{refPeriod}$) and decrements until the refresh pattern is executed. At this time, the timer is incremented by $t_{refPeriod}$ again. This ensures that the average time between refresh patterns equals $t_{refPeriod}$. The mapping can be slightly optimized, because a switch pattern before or after a refresh pattern is not necessary. However, it makes analysis more difficult. In addition, it has a minimal impact on the rate, since refreshes are not executed that often and switching patterns are fairly small.

6.4.1.1 Net bandwidth and rate

The bounds on rate and the net bandwidth are closely related. The notation that is used here and in the remainder of the report is listed in Table 6.5. Note that latency, execution time and rate can be different for every request.

We determine the upper bound on the execution time of a sequence of n_{access} memory accesses to compute the minimum guaranteed rate. Definition 6.5 shows the relation between the number of memory accesses and requests ($n_{request}$). The execution time of all the memory accesses is now equal to the execution time of the requests as defined by Definition 6.6.

Definition 6.5 (Relation between requests and the number of memory accesses)

$$n_{access} = \sum_{i=0}^{n_{request}-1} size_{ra}(i)$$

Table 6.4: Truth table of the PAM

refresh timer	*previous pattern*	*memory access*	*patterns*
$c > 0$	not WR	read	RD
$c \leq 0$	not WR	read	REF RD
$c > t_{wtr}$	WR	read	WTR RD
$c \leq 0$	WR	read	REF WTR RD
$0 < c \leq t_{wtr}$	WR	read	WTR REF RD
$c > 0$	not RD	write	WR
$c \leq 0$	not RD	write	REF WR
$c > t_{rtw}$	RD	write	RTW WR
$c \leq 0$	RD	write	REF RTW WR
$0 < c \leq t_{rtw}$	RD	write	RTW REF WR
$c > 0$	-	idle	IDLE
$c \leq 0$	-	idle	REF IDLE

where:

RD	Read pattern
WR	Write pattern
IDLE	Idle pattern
RTW	Read to write switch pattern
WTR	Write to read switch pattern
REF	Refresh pattern

Table 6.5: Notation

$\hat{x}$	The upper bound of x
$\check{x}$	The lower bound of x
$\Theta(i)$	The latency of request i; defined as the time between the start of the request and the time that an uninterrupted stream of data can be provided
$\rho(i)$	The rate of request i; defined as the rate that the request can be served
$\mathcal{E}(i)$	The execution time of request i; defined as the inverse rate: $\mathcal{E}(i) = 1/\rho(i)$; it is also known as the data introduction interval
$size_{ra}(i)$	The number of memory accesses that are executed for request i
$type(i)$	The type of memory accesses that are executed for request i

Definition 6.6 (Execution time of $n_{request}$ requests)

$$\mathcal{E}_{total} = \sum_{i=0}^{n_{request}-1} \mathcal{E}(i)$$

The total execution time of all memory accesses depends on the number and length of

the memory command patterns as shown in Equation (6.8).

$$\mathcal{E}_{total} = n_{ref} \cdot t_{ref} + n_{access} \cdot t_{access} + n_{rtw} \cdot t_{rtw} + n_{wtr} \cdot t_{wtr} \tag{6.8}$$

To compute the worst case execution time of the memory accesses ($\hat{\mathcal{E}}_{total}$), the number of refreshes and switch patterns must be determined. First, we calculate the worst case number of refresh patterns ($\hat{n}_{ref}$). The time between two successive refresh patterns cannot be constant, since it has to wait until an active pattern is finished (Table 6.4). The time that the refresh pattern has to wait is referred to as *blocking*. According to Equation (6.9), the maximum blocking of a refresh pattern is based on the longest pattern, except the refresh pattern itself:

$$\hat{t}_{refreshBlocking} = \max(t_{rtw}, t_{wtr}, t_{access}) - t_{clk} \tag{6.9}$$

where: t_{clk} Clock period of the memory memory

The worst case number of refreshes in a certain period can be calculated by Equation (6.10).

$$\hat{n}_{ref} = \left\lceil \frac{t_{period} + \hat{t}_{refreshBlocking}}{t_{refPeriod}} \right\rceil \tag{6.10}$$

where: $t_{refPeriod}$ Average refresh period of the memory
t_{period} The period that refreshes can take place

However, this requires $t_{period} = \mathcal{E}_{total}$, which is not yet known. A possible method is to create an algorithm that uses an initial execution time and performs multiple iterations to determine the number of refreshes. The maximum number of switching patterns ($\hat{n}_{rtw}$ and $\hat{n}_{wtr}$) are calculated by Equations (6.11) and (6.12). For the worst case must be assumed that the data bus has to be switched for every access pattern because the type of accesses are not known at design time. We assume that the write to read switch pattern is equal or longer than its counterpart ($t_{wtr} \geq t_{rtw}$).

$$\hat{n}_{rtw} = \left\lfloor \frac{n_{access}}{2} \right\rfloor \tag{6.11}$$

$$\hat{n}_{wtr} = \left\lceil \frac{n_{access}}{2} \right\rceil \tag{6.12}$$

Finally, the worst case execution time of the memory accesses can be determined by Equation (6.8). The minimum rate for n_{access} memory accesses is defined by Equation (6.13).

$$\check{\rho} = \frac{n_{access}}{\hat{\mathcal{E}}_{total}} \tag{6.13}$$

The minimum rate is used to compute the net bandwidth according to Equation (6.14). This is the net bandwidth for an interval of $\hat{\mathcal{E}}_{total}$ amount of time and assuming that the data efficiency is 100%. The size of a memory access ($width_{ra}$) can be computed by Equation (6.7).

$$netBandwidth = width_{ra} \cdot \check{\rho} \tag{6.14}$$

6.4.1.2 Latency

Section 6.3.5.4 introduced the first and last data latency of an access pattern. These definitions can be extended to requests. The first and last data latency for a request are:

- *First data latency*: The time between the start of the first pattern of the request and the cycle that an uninterrupted stream of data can be guaranteed (all cycles must have data)
- *Last data latency*: The time between the start of the first pattern of the request and the cycle after the last data

We use the worst-case execution time of a request (Equation (6.15)) to determine the bounds on first and last data latency. One switch pattern is included since all memory accesses of the requests have the same type. The worst case number of refresh patterns can be calculated by Equation (6.10) and using $t_{period} = \hat{\mathcal{E}}_(i)$.

$$\hat{\mathcal{E}}_(i) = \hat{n}_{ref} \cdot t_{ref} + size_{ra}(i) \cdot t_{access} + t_{wtr} \tag{6.15}$$

The worst-case last data latency for request i is calculated using Equations (6.16) and (6.17).

$$\hat{\Theta}_{last-burst}(i) = \begin{cases} \hat{\mathcal{E}}_(i) - t_{access} + t_{last\_read\_cmd} & if\ type(i) = read \\ \hat{\mathcal{E}}_(i) - t_{access} + t_{last\_write\_cmd} & if\ type(i) = write \end{cases} \tag{6.16}$$

$$\hat{\Theta}_{last-data}(i) = \begin{cases} \hat{\Theta}_{last-burst} + R\_RDATA + \frac{burstLength}{\rho_{memory}} & if\ type(i) = read \\ \hat{\Theta}_{last-burst} + W\_WDATA + \frac{burstLength}{\rho_{memory}} & if\ type(i) = write \end{cases} \tag{6.17}$$

where: ρ_{memory} Data rate of the memory

From the last data latency, the upper bound on first data latency can be derived according to Equation (6.18). The size of the request is only equal to $size_{ra}(i) \cdot width_{ra}$, if the request is aligned to memory accesses. Figure 6.12 illustrates the timing of an unaligned read request based on Figure 6.2(b).

$$\hat{\Theta}_{first-data}(i) = \hat{\Theta}_{last-data}(i) - \frac{size(i)}{\rho_{memory}} \tag{6.18}$$

where: $size(i)$ Size of the request in terms of data

However, the upper bounds depend on the address, size and type of the request ($size_{ra}(i)$, $size(i)$ and $type(i)$). To derive an upper bound for any read or write request with a maximum size of $\hat{size}$, the maximum number of memory accesses is calculated by Equation (6.19). In the worst case, two additional memory accesses are executed for an unaligned request. In Section 7.2.1 is shown that only the latency of read requests is required.

$$\hat{size}_{ma} = \begin{cases} \frac{\hat{size}}{width_{ra}} & \text{only aligned requests} \\ \left\lceil \frac{\hat{size}}{width_{ra}} \right\rceil + 2 & \text{otherwise} \end{cases} \tag{6.19}$$

We conclude that this memory access to pattern mapping is predictable according to Definition 6.4, since bounds on rate and last data latency can be derived without depending on run-time information.

6.4.2 Composable memory access to pattern map

Besides predictability, the memory controller requires composability. According to Definition 2.4, the behaviour of all memory accesses executed by a job, may not depend on the memory accesses of other jobs. However, the memory does not know which job issued a memory access. This means that the rate and last data latency must be independent of the type and address of the memory access. According Definition 6.7, this is a *composable memory access to pattern map* (CAM).

Definition 6.7 *A* **composable memory access to pattern map** *is a memory access to pattern map that guarantees that the* rate *is independent of the type and address of the executed memory accesses.* Last data latency *does not depend on other memory accesses.*

The CAM for the proposed memory controller uses the same patterns as the PAM proposed in Section 6.4.1. However, they are mapped differently such that it conforms to Definition 6.7. The rate of the PAM is not composable, because the total length of a memory access is not independent of other accesses. This can be seen from Table 6.4, where the patterns for a memory access depends on the last pattern of the previous access (column *previous pattern*).

For the same reason is the last data latency dependent on the previous access. The latency of a read or write pattern is constant, but a switch pattern increases the latency. The insertion of a switch pattern depends on the previous access.

To create a CAM, six new patterns are defined which are composed of the patterns of the PAM (illustrated in Figure 6.13). We distinguish between long and short access patterns. Long patterns have the same length and also the length of all short patterns are equal. We assume that the write to read switch pattern is equal or longer than its counterpart ($t_{wtr} \geq t_{rtw}$). This is a valid assumption for the patterns that we use (Appendix B, Table B.5). A switching pattern only consists of NOP commands. Therefore, the length (i.e. the number of NOP commands) is the only difference between the

RTW and WTR patterns. Except for two successive short patterns, the long and short patterns can be issued back to back. The R_W and W_R timing constraints are satisfied regardless of the access type, because the switch patterns (part of the short and long patterns) assure that there is sufficient time between read, idle, and write patterns.

Table 6.6 shows the truth table of the CAM. The first memory access assumes that the previous pattern is a short pattern. Therefore, the first memory access uses a long pattern (according to the truth table). The second memory access uses a short one, and so on. Short and long patterns are issued in an interleaved way. Hence, the total length of a memory access is not affected by others. Since the execution time of a memory access does not change at run time, the rate is not affected by the type and address of memory accesses. For the same reason, the last data is also not dependent on other memory accesses. From these properties and Definition 6.7 can be concluded that this is a composable memory access to pattern map.

Figure 6.14 illustrates that long and short patterns are interleaved. The second run has different type of memory accesses, but this does not affect the length.

The distinction between long and short patterns allows a higher rate than when patterns of a constant length would be used, as the length must be equal to the long pattern to satisfy the timing constraints.

Table 6.6: Truth table for the CAM

refresh timer	*previous pattern*	*memory access*	*patterns*
$c > 0$	short	read	L-RD
$c > 0$	long	read	S-RD
$c \leq 0$	short	read	REF L-RD
$c \leq 0$	long	read	REF S-RD
$c > 0$	short	write	L-WR
$c > 0$	long	write	S-WR
$c \leq 0$	short	write	REF L-WR
$c \leq 0$	long	write	REF S-WR
$c > 0$	short	idle	L-IDLE
$c > 0$	long	idle	S-IDLE
$c \leq 0$	short	idle	REF L-IDLE
$c \leq 0$	long	idle	REF S-IDLE

where:

short	S-RD, S-WR or S-IDLE pattern
long	L-RD, L-WR or L-IDLE pattern
S-RD	Short read pattern
L-RD	Long read pattern
S-WR	Short write pattern
L-WR	Long write pattern
S-IDLE	Short idle pattern
L-IDLE	Long idle pattern
REF	Refresh pattern

The CAM guarantees the same rate and net bandwidth as the PAM, because the rules of the CAM enforces the worst-case of the PAM. Therefore, the rate and net bandwidth are not only bounded but also constant.

However, the data latency is longer because switches are also executed between memory accesses of the same type. As explained earlier, this is necessary for a constant rate at run time. Equations (6.20), (6.21), and (6.22) are used to compute the execution time of a request. The remaining equations of the PAM can be used to derive bounds on data latency.

$$\hat{\mathcal{E}}_(i) = \hat{n}_{ref} \cdot t_{ref} + size_{ra}(i) \cdot t_{access} + \hat{n}_{rtw} \cdot t_{rtw} + \hat{n}_{wtr} \cdot t_{wtr} \tag{6.20}$$

$$\hat{n}_{rtw} = \left\lfloor \frac{size_{ra}(i)}{2} \right\rfloor \tag{6.21}$$

$$\hat{n}_{wtr} = \left\lceil \frac{size_{ra}(i)}{2} \right\rceil \tag{6.22}$$

A bound for rate and last data latency can be calculated without depending on run time information. Therefore, this memory access to pattern map is not only composable but also predictable.

6.4.3 Results

Based on the proposed PAM and CAM, this section shows the guarantees on maximum latency and minimum bandwidth for a DDR2-400, DDR2-800, DDR3-800 and DDR3-1600 memory device. The patterns for the PAM and CAM are listed in Appendix B.

6.4.3.1 Guaranteed maximum latency

Tables 6.7 and 6.8 present the guaranteed upper bound on first and last data latency for different memory devices. As mentioned earlier, the first data latency does not represent the time when the first word arrives, but when an uninterrupted stream can be guaranteed. Therefore, the first data latency depends on the last data latency. Patterns with different access granularities have been selected to illustrate the effect of the properties of the access patterns.

The latencies are calculated by Equations (6.17) and (6.18). The maximum number of memory accesses is computed according to Equation (6.19). Only latency of read requests is listed. The maximum latency of a write request is slightly less for two reasons: First, write commands of the write pattern are executed at the same cycle as the read pattern. Secondly, the time between the write command and the write data is less than for a read command. For all devices, the maximum size of a request is exactly two memory accesses. Therefore, unaligned requests can only occur when the address is not aligned with a memory access. Latency for unaligned requests are listed in Table 6.8.

First of all, there is not much difference between the maximum latency of the CAM and PAM. There is no difference at all for the DDR2-800 and DDR3-1600 devices,

because their switching patterns have a zero length ($t_{wtr} = t_{rtw} = 0$). Hence, the PAM and CAM have the same worst-case execution time of a request.

From the tables can be concluded that the memory controller guarantees a lower latency for aligned requests. The main reason is that two more memory accesses are required in the worst case. The DDR2-800 device has the lowest latency for aligned requests. It outperforms the DDR2-400 device, because it has a higher clock frequency. The DDR3-800 has the worst first data latency, due to the large memory accesses and maximum request size. However, it can guarantee high net bandwidth as shown by Figure 6.17. Although the DDR3-1600 device has the lowest latency for unaligned requests, it is not efficient because the clock frequency is two or four times higher than the other memory devices.

Table 6.7: Data latency, assuming aligned requests

device	$width_{ra}$	$\hat{size}$	$\hat{\Theta}_{first-data}$		$\hat{\Theta}_{last-data}$	
			PAM	*CAM*	*PAM*	*CAM*
DDR2-400	64 bytes	128 bytes	180 ns	190 ns	340 ns	350 ns
DDR2-800	64 bytes	128 bytes	162.5 ns	162.5 ns	242.5 ns	242.5 ns
DDR3-800	128 bytes	256 bytes	200 ns	207.5 ns	360 ns	367.5 ns
DDR3-1600	32 bytes	64 bytes	170 ns	170 ns	190 ns	190 ns

Table 6.8: Data latency, assuming unaligned requests

device	$width_{ra}$	$\hat{size}$	$\hat{\Theta}_{first-data}$		$\hat{\Theta}_{last-data}$	
			PAM	*CAM*	*PAM*	*CAM*
DDR2-400	64 bytes	128 bytes	340 ns	380 ns	500 ns	540 ns
DDR2-800	64 bytes	128 bytes	297.5 ns	297.5 ns	377.5 ns	377.5 ns
DDR3-800	128 bytes	256 bytes	360 ns	397.5 ns	520 ns	557.5 ns
DDR3-1600	32 bytes	64 bytes	280 ns	280 ns	300 ns	300 ns

6.4.3.2 Guaranteed net bandwidth

The minimum guaranteed rate of the PAM is a bound for the number of memory accesses in a certain period (Equation (6.13)). The net bandwidth can be derived from the rate according to Equation (6.14). As mentioned earlier, the net bandwidth of the proposed CAM is equal to the bound of the PAM.

Figures 6.15 and 6.17 show the net bandwidth for the access patterns that have the highest efficiency. Data efficiency is assumed to be 100% in the figures. For the Figures 6.16 and 6.18 access patterns with a finer access granularity are chosen. The patterns for these devices can be found in Appendix B.

The curves indicate the minimum net bandwidth for intervals with an increasing length. No bandwidth can be guaranteed for very small intervals, because this interval is too small to guarantee that one full memory access is executed. For intervals longer than 1400 ns, all devices can guarantee at least 90% of the bandwidth for an infinite interval. Around multiples of the refresh period (7800 ns), an additional refresh must be

executed. Therefore, the curves are flat for a while until a longer interval can guarantee a higher bandwidth again. Table 6.9 lists the net bandwidth for the devices for a time interval of 188 us. The table confirms that the DDR3-1600 and DDR2-800 devices have a low efficiency, because they have a fine access granularity ($width_{ra}$). The DDR2-800 and DDR3-800 have an efficiency of more than 82% because they have a high bank efficiency. Even more bandwidth can be guaranteed when they use larger memory accesses, since less switching patterns are necessary for the same amount of data.

Table 6.9: Minimum guaranteed net bandwidth for an interval of approximately 188us

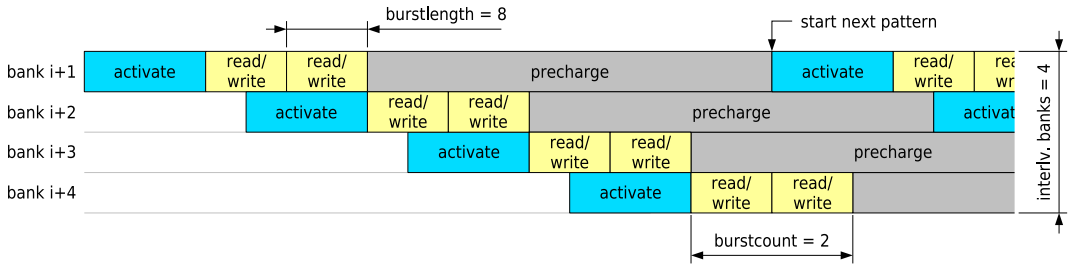
device	$width_{ra}$	*gross bandwidth*	*net bandwidth*	*efficiency*
DDR2-400	64 bytes	800 MB/s	662 MB/s	82.8%
DDR2-800	64 bytes	1600 MB/s	934 MB/s	58.4%
DDR3-800	128 bytes	1600 MB/s	1320 MB/s	82.6%
DDR3-1600	32 bytes	3200 MB/s	574 MB/s	17.9%

6.5 Conclusions

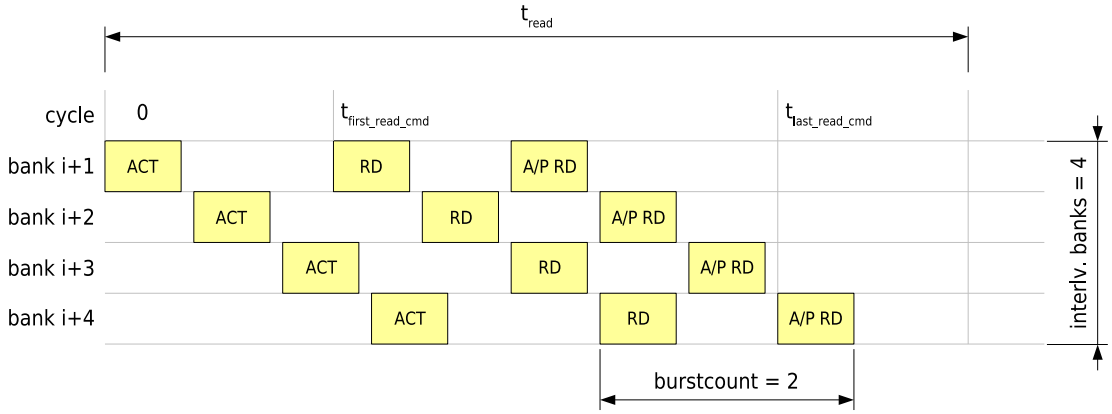
The worst-case execution time of a single burst is very high compared to the best case. A memory access defines the smallest amount of data that the memory controller accesses. A fine granularity (i.e. a memory access consists of one burst) results in poor net bandwidth guarantees such that an expensive memory must be used to satisfy real-time requirements. Better results can be obtained by increasing the access granularity.

Memory access to pattern maps are used to translate memory accesses to patterns. A PAM guarantees that the rate and latency of memory accesses are bounded at design time. A CAM guarantees that the rate and latency are not affected by other memory accesses. Communication between memory controller and memory uses the concept of memory accesses as basic operation. The PAM and CAM assure that the behaviour can be implemented and is predictable and composable, respectively.

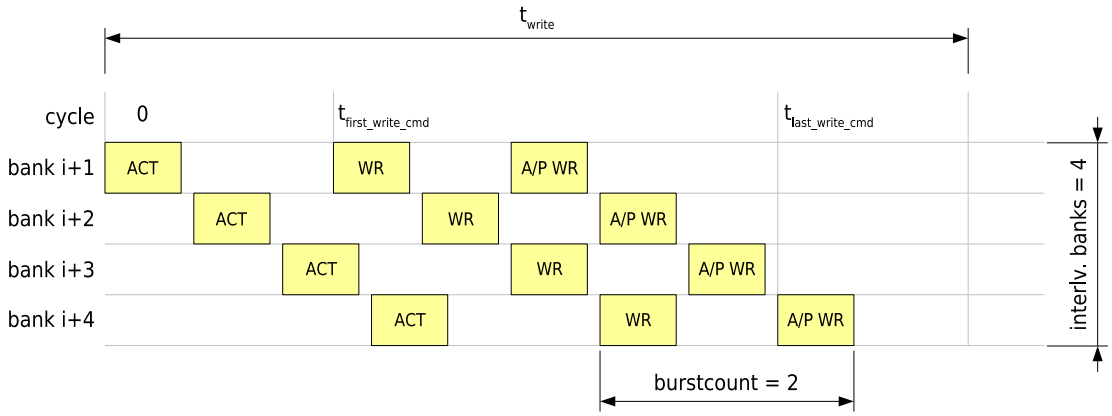
Analysis of access patterns of the CAM and PAM shows that 'faster' memory devices are not able to guarantee higher net bandwidth for a fine access granularity. Higher net bandwidth can only be provided for coarser access granularities. We expect that the size of a memory access has to grow significantly to maintain efficiency. The guaranteed maximum latency of data is strongly affected by the access granularity, the maximum size, and alignment of a request. In spite of the high frequency of DDR3-1600 memory, the guaranteed latency does not improve much. The minimum net bandwidth of the PAM and CAM depend on the interval. For intervals longer than 1400 ns, 90% of the bandwidth for an infinite interval can be guaranteed. A DDR2-400 device guarantees a minimum net bandwidth of 662 MB/s for an access granularity of 32 bytes. A DDR3-800 device can guarantee at least 1320 MB/s but requires an access granularity of 128 bytes.



(a) Operations executed by the memory for a read or write pattern



(b) SDRAM commands of a read pattern



(c) SDRAM commands of a write pattern

Figure 6.4: Read and write pattern

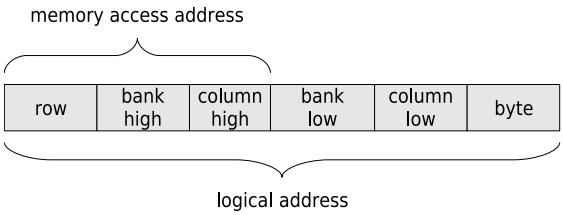
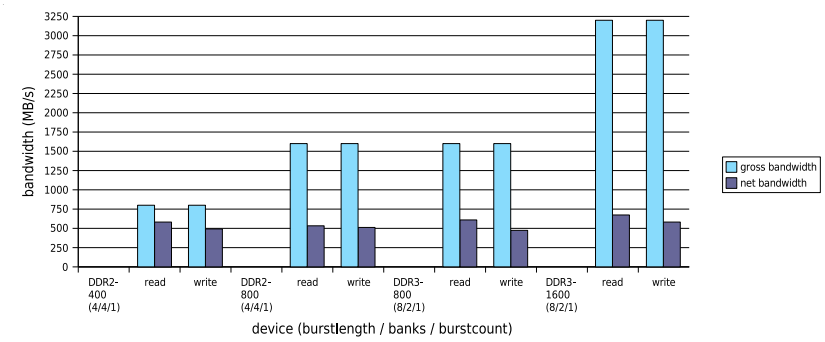
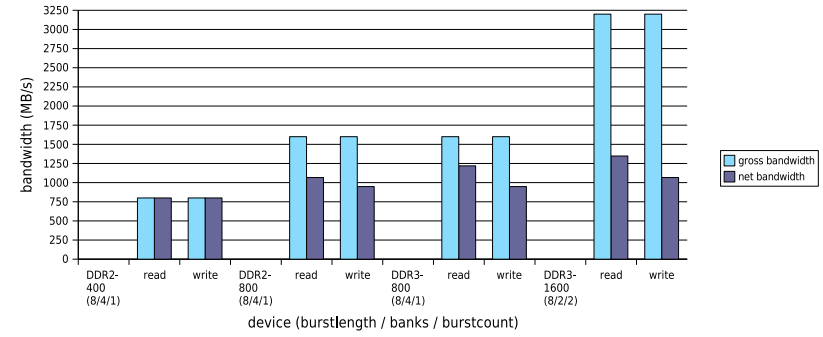


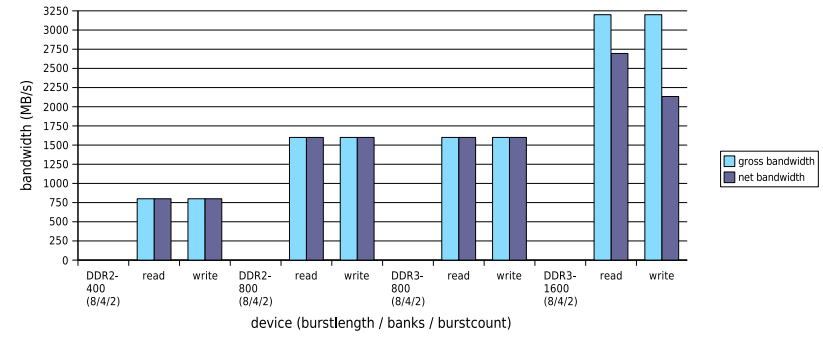
Figure 6.5: Memory map for generalized basic groups



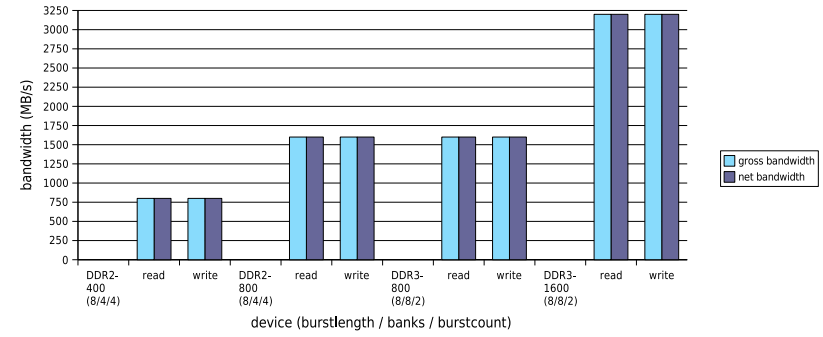
(a) Access granularity of 32 bytes



(b) Access granularity of 64 bytes



(c) Access granularity of 128 bytes



(d) Access granularity of 256 bytes

Figure 6.6: Bank efficiency for read patterns, other efficiencies are assumed to be 100%.

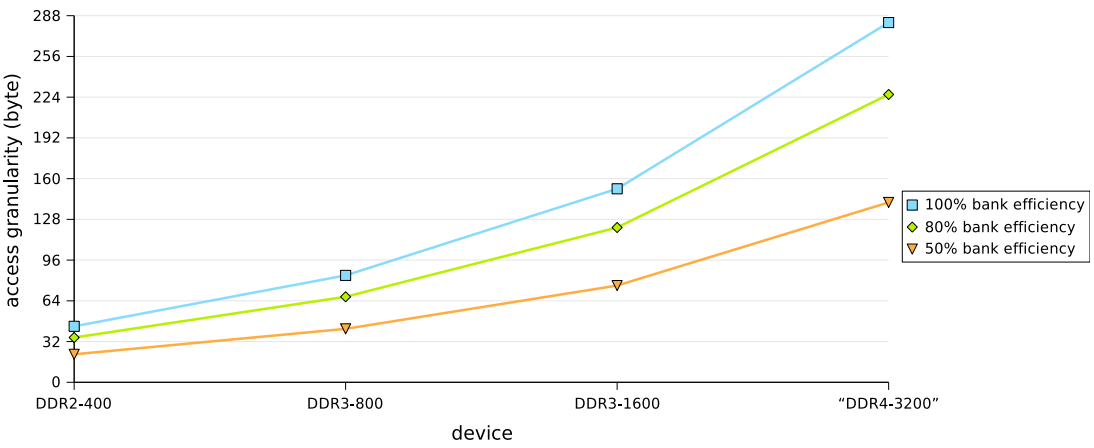
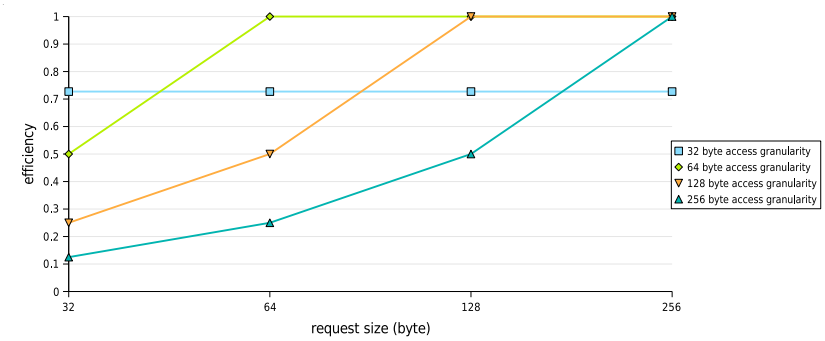
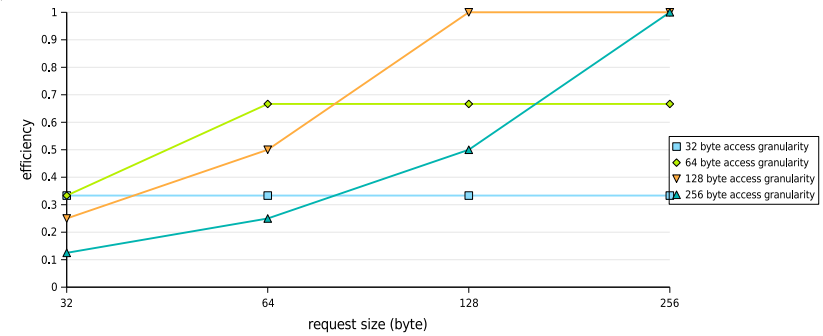


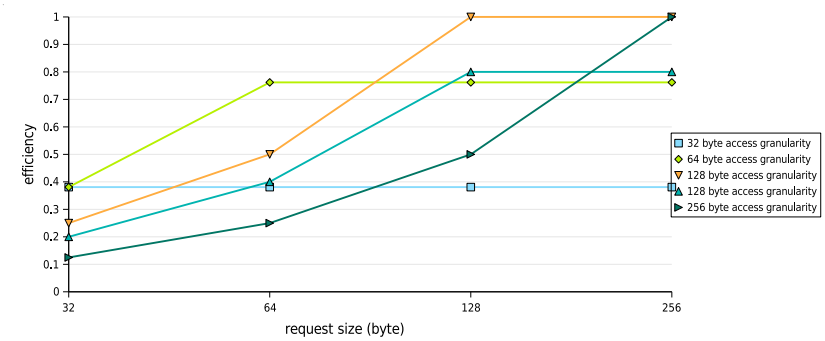
Figure 6.7: Granularity trend of read patterns



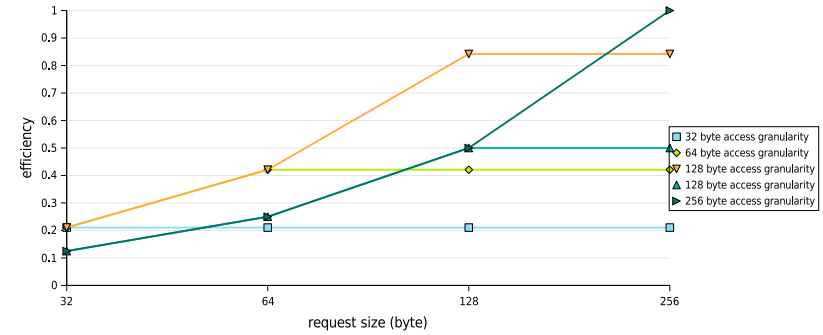
(a) DDR2-400 device



(b) DDR2-800 device

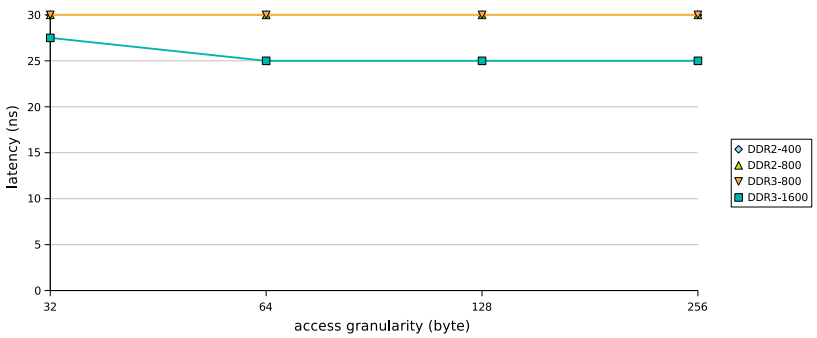


(c) DDR3-800 device

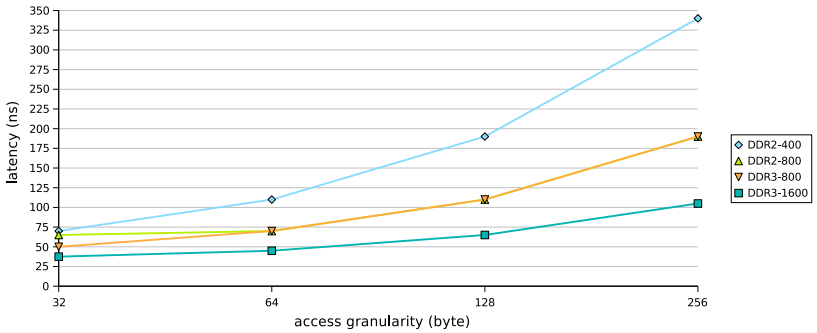


(d) DDR3-1600 device

Figure 6.8: Bank and data efficiency for read patterns, other efficiencies are assumed to be 100%.



(a) First data latency



(b) Last data latency

Figure 6.9: Data latency of read patterns

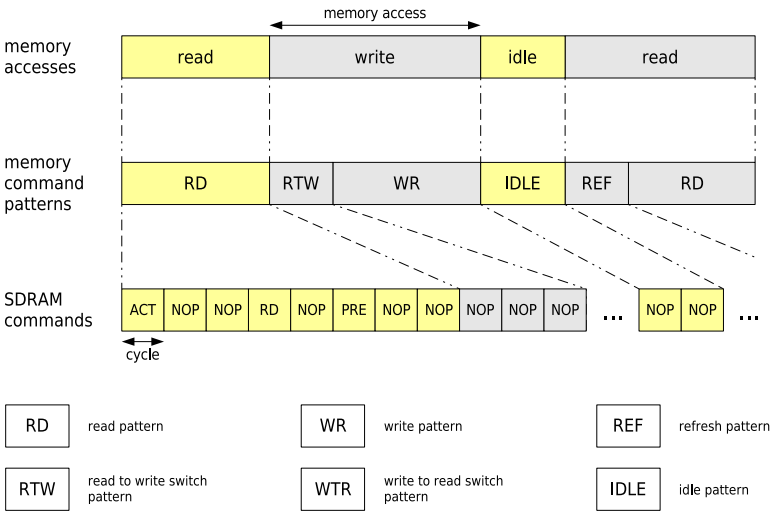


Figure 6.10: Relation between memory access, memory command patterns and SDRAM commands

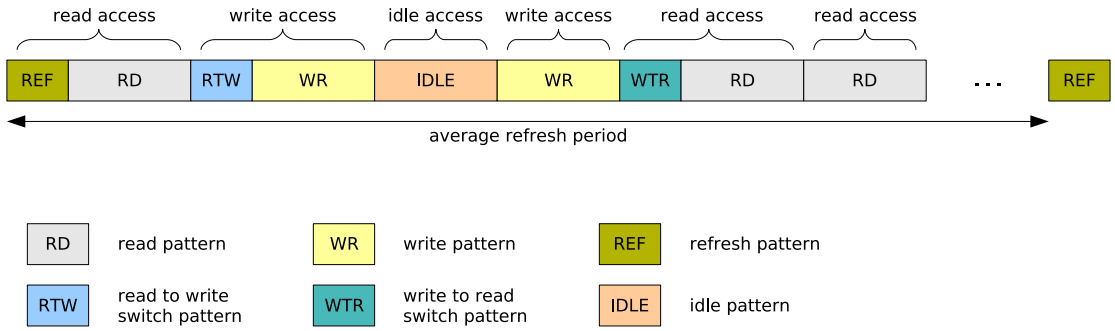


Figure 6.11: Sequence of the patterns according to the PAM

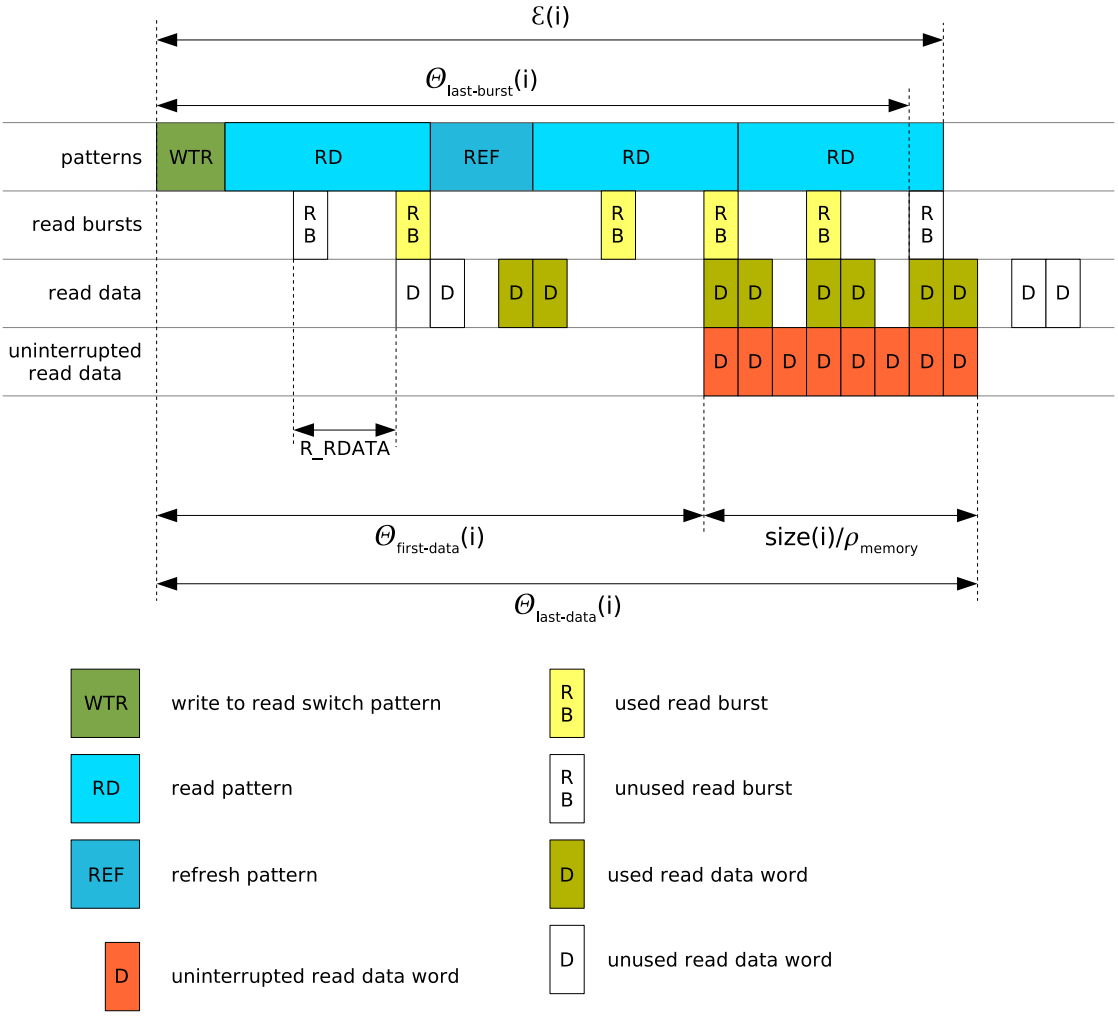


Figure 6.12: Timing details of a read request. The read pattern has two read bursts with a length of two words.

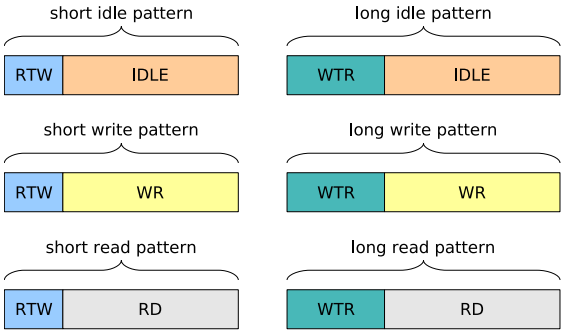


Figure 6.13: The composition of the patterns of the CAM

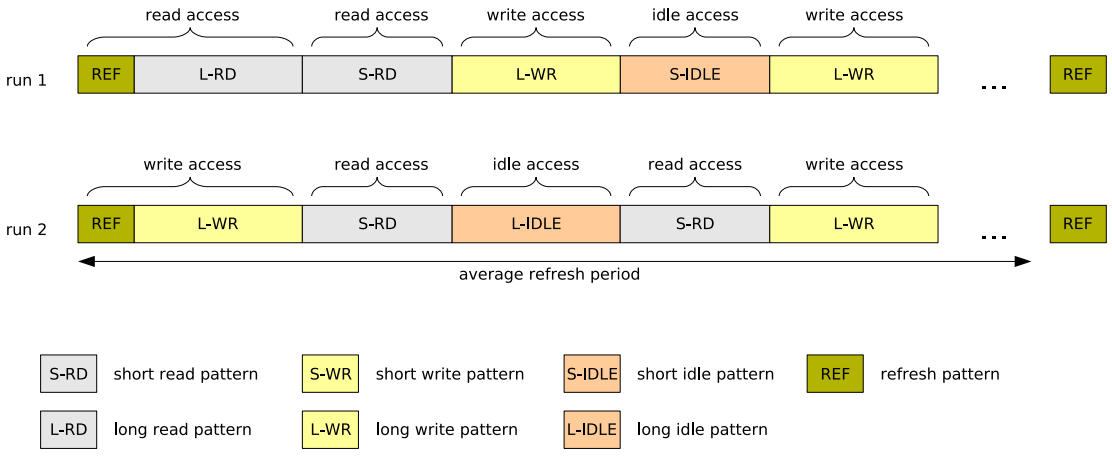


Figure 6.14: Two sequence of patterns according to the CAM

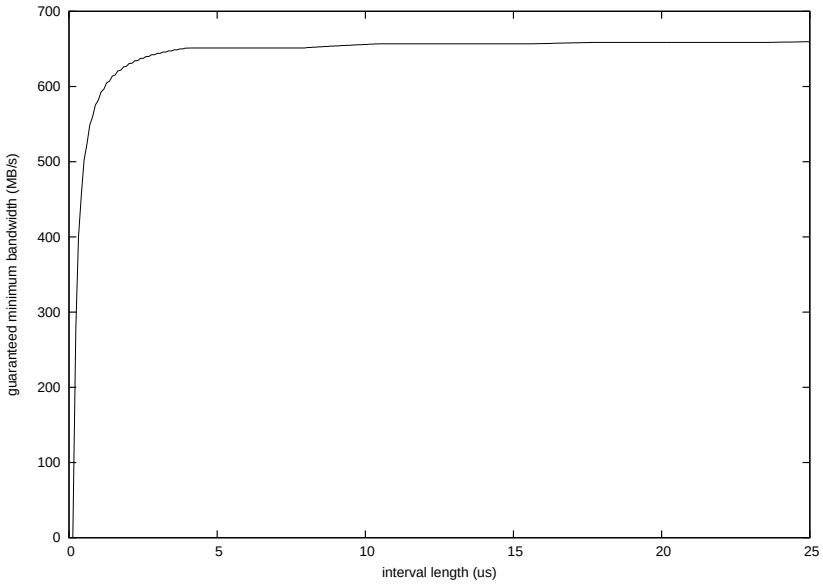


Figure 6.15: Guaranteed net bandwidth for DDR2-400 device (burst length = 8, interleaved bank count = 4, burst count = 1, 256Mb)

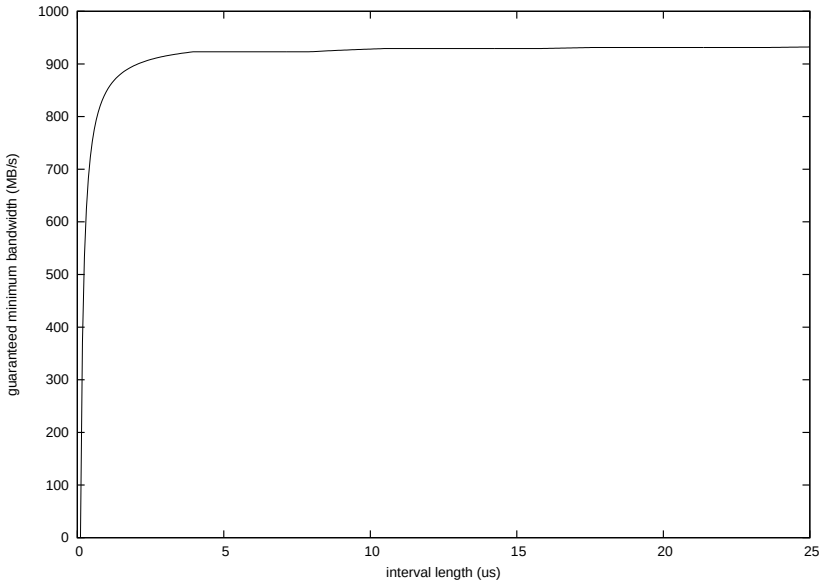


Figure 6.16: Guaranteed net bandwidth for DDR2-800 device (burst length = 8, interleaved bank count = 4, burst count = 1, 256Mb)

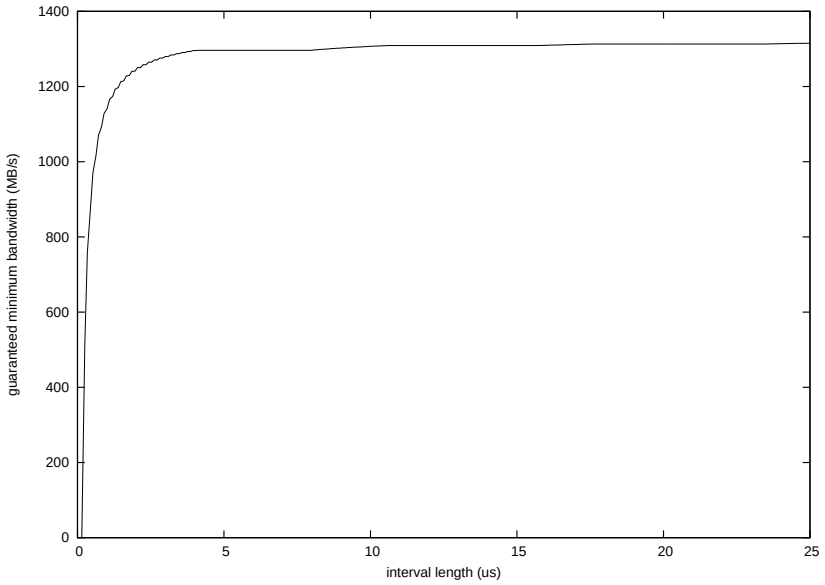


Figure 6.17: Guaranteed net bandwidth for DDR3-800 device (burst length = 8, interleaved bank count = 4, burst count = 2, 512Mb)

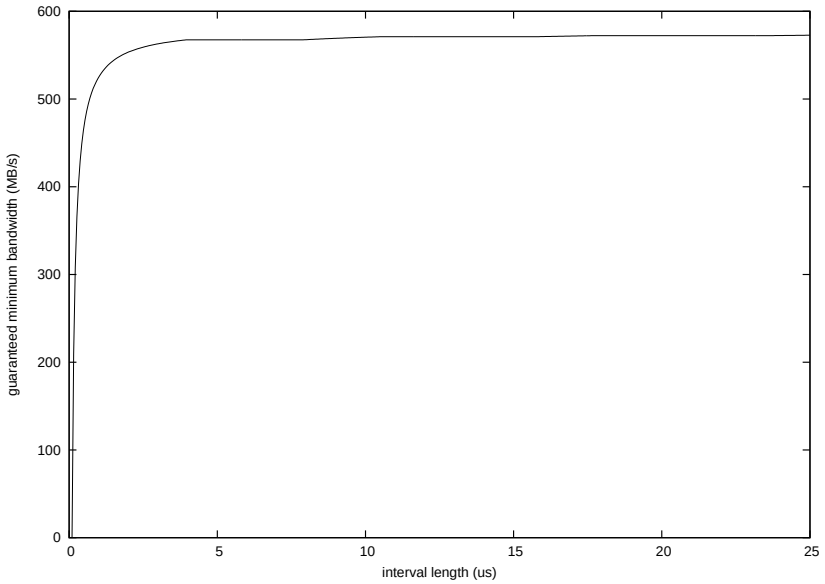


Figure 6.18: Guaranteed net bandwidth for DDR3-1600 device (burst length = 8, interleaved bank count = 2, burst count = 1, 512Mb)

7 Design

Based on the requirements of Section 2, this section proposes a design for the memory controller. The architecture of the design is discussed in Section 7.1. Section 7.2 shows the analysis of this architecture and sets requirements for the implementation to satisfy predictability and composability. The hardware implementation of this design is discussed in Section 8.

7.1 Architecture

The requirements for the memory controller determine the constraints for the architecture. The primary requirements are predictability and composability. Both requirements reduce the verification time of a SoC. All components of the memory controller (requestor interfaces, arbiter and memory interface) could be designed, as depicted in Figure 5.4. The biggest advantage is that the complete design can be tailored to the requirements. However, the problem is that the memory controller depends on the interface of the memory. All memories have a different interface or timing constraints, such that the memory controller is not reusable.

The architecture of the proposed memory controller, makes a strong separation between the front-end and back-end. Figure 7.1 shows that the requestor interfaces, arbiter and back-end interface forms the front-end. The back-end consists of the memory interface. Communication between the front-end and back-end must not depend on the memory, as this complicates the timing analysis. This architecture also allows easy integration of the back-end. When new memory devices should be supported, only the back-end needs to be replaced. As mentioned in Section 3, Sonics [27] proposes a similar memory system that distincts between the front-end (MemMax) and back-end (DRAM controller).

From the perspective of resources, the network and requestor interfaces belong to the interconnect. The remainder of the front-end and back-end correspond to the memory controller.

7.1.1 Requestor interfaces

Every requestor has its own requestor interface to avoid dependencies between requestors in this component. Dependencies between requestors violate the composability of the architecture and increases the effort of static timing analysis. The requestor interfaces prepare the requests in such a way that the back-end does not stall. Section 8.3 discusses this in more detail. The responses from the arbiter are converted to the original format and sent back to the requestor. The requestor interfaces belong to interconnect in our

implementation, because protocol encoding and decoding of network messages is the main activity.

7.1.2 Arbiter

The arbiter is the entry point of the memory controller. Requests are retrieved from the queue of each requestor interface. Only the request at the head of each queue can be scheduled. The scheduler of the arbiter merely selects a requestor and sends the request at the head of the queue to the back-end interface. The response of a request is routed from the back-end interface to the corresponding requestor interface. The arbiter does not reorder requests, because it complicates the timing analysis and increases the worst-case latency. Furthermore, the arbiter is not allowed to depend on a memory technology. Tasks like memory mapping and command generation are performed by the back-end. A disadvantage is that the arbiter cannot exploit memory specific information (like the physical address of a request) to improve efficiency.

7.1.3 Back-end interface

To decouple the arbiter and back-end, the back-end interface is inserted between the arbiter and back-end. It translates the request into back-end commands (not SDRAM commands) and write data when it is a write request. The back-end commands initiate memory accesses. Responses are composed from information of the request and the data returned from the back-end.

7.1.4 Back-end

The back-end is responsible for the real control of the memory. Memory command patterns are sent to the memory to perform operations like read data, write data and refresh. Only the behaviour of the back-end is proposed; no hardware implementation is provided. The interface should be compatible with the back-end interface, described in Section 8.5. Memory controllers that can be modelled by a PAM or CAM can be used as a back-end.

7.1.5 Generalisation

The front-end is not aware of the actual resource being used. It is just a scheduler of requests from different requestors. Therefore, the front-end is not restricted to SDRAM devices, but can support any predictable shared resource that can be accessed through reads and writes (memory mapped). Possible shared resources that can be used are other types of memory, like SRAM. In addition, a single-hop interconnect and memory mapped peripherals could also be shared using the front-end.

7.2 Data flow analysis

Recall from Section 2.1.2 that the predictable memory controller has to bound the following behaviour at design time:

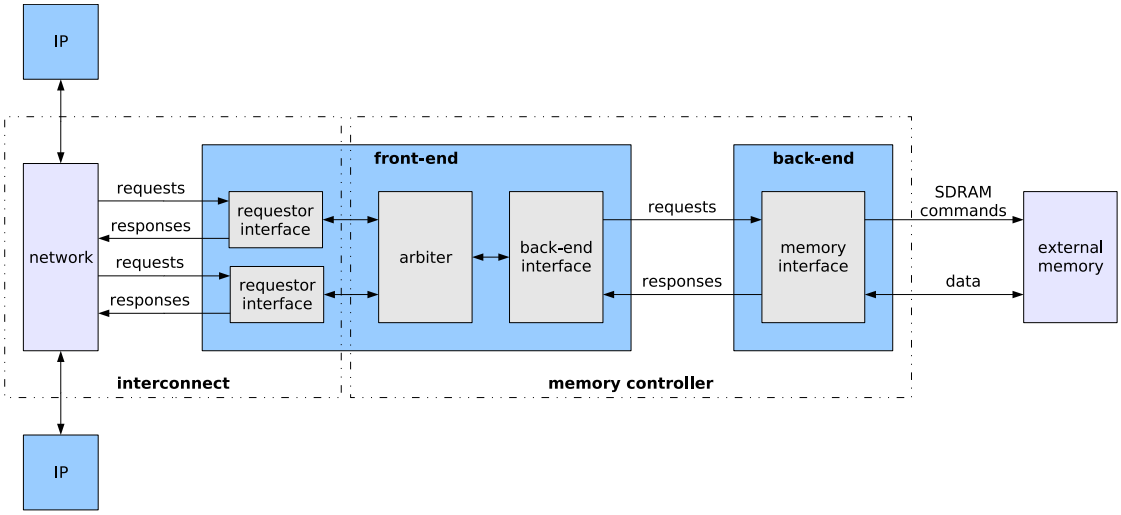


Figure 7.1: Memory controller split into front-end and back-end

- *Latency*: The time between arrival of a request and the head of the response.
- *Net bandwidth*: The amount of data that can be read or written in a certain time interval.

For a composable memory controller, this behaviour may not depend on other requestors (Section 2.1.3). To analyse the data flow of the front-end and back-end, a model comparable to Latency Rate ($\mathcal{LR}$) servers [26] is used. The behaviour of a $\mathcal{LR}$ server can be characterized by latency (Θ) and allocated rate (ρ'). A component that belongs to the class of $\mathcal{LR}$ servers guarantees an allocated rate and a maximum latency. Note that the minimum net bandwidth can be derived from the allocated rate according to Equation (6.14) on page 45.

For predictability and composability, the behaviour of the memory has to be analysed. We use a simple data flow model for this purpose. Figure 7.2 shows three components A, B and C where the data (requests and responses) flows from left to right. In the implementation of the design, requests (and responses) map to headers and streams of data. The arrival and finishing time of request i at some component is defined by Definition 7.1 and denoted by $a(i)$ and $f(i)$, respectively. A subscript is used to distinct between components. A request is identified by an index, where $i = 0$ corresponds to the first request arriving at the component. A higher index denotes a request that arrives later: $a(i+1) > a(i)$. By definition, the finishing time of a request is equal to the arrival time at the next component on the path. For the example data flow model shown by the figure this means that $f_A(i) = a_B(i)$ and $f_B(i) = a_C(i)$. Furthermore, we assume that the components do not reorder requests such that: $f(i+1) > f(i)$. According to Definition 7.2, the relation between the finishing and arrival time of request i is characterized by latency ($\Theta(i)$) and execution time ($\mathcal{E}(i)$). Latency and execution time are not necessarily constant for every request. For convenience, the notation for the data flow models is repeated in Table 7.1. In the real implementation, handshakes between the components assure that the producer is slowed down when they produce data at

Table 7.1: Notation

$\hat{x}$	The upper bound of x
$\check{x}$	The lower bound of x
$\Theta(i)$	The latency of request i; defined as the time between the start of the request and the time that an uninterrupted stream of data can be provided
$\rho(i)$	The rate of request i; defined as the rate that the request can be served
$\mathcal{E}(i)$	The execution time of request i; defined as the inverse rate: $\mathcal{E}(i) = 1/\rho(i)$; it is also known as the data introduction interval
$size_{ra}(i)$	The number of memory accesses that are executed for request i
$type(i)$	The type of memory accesses that are executed for request i

a higher rate than the consumer. This method is called back-pressure. To simplify analysis, we assume that requests do not arrive too fast such that back-pressure does not cause stalling. Lemma 7.1 shows the behaviour for a component with a constant latency. Components with a variable latency still have bounded behaviour when the rate of arrivals is not too high, according to Lemma 7.2. Both lemmas are used in this section to define the behaviour of the components.

The buffers that are used in the data flow model require one cycle (t_{clk}) to store a word. Buffers with a sufficient capacity are required to avoid back-pressure of components with an irregular execution time. Back-pressure is still implemented for robustness. Whenever something fails, the memory controller remains operational and does not discard any data. However, deadlines may be missed.

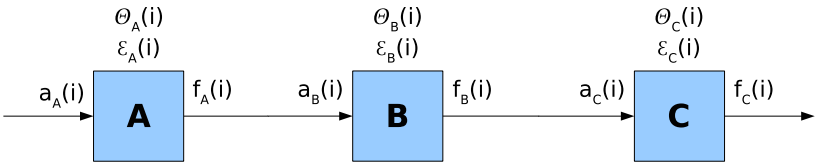


Figure 7.2: Data flow model

Definition 7.1 *The* **arrival time** *and* **finishing time** *of a request (or response) is defined as the time that the complete request can be produced without interruptions.*

Definition 7.2 *The behaviour of a component is defined as:*

$$f(i) = \begin{cases} a(i) + \Theta(i) & for\ i = 0 \\ \max(a(i) + \Theta(i), f(i-1) + \mathcal{E}(i-1)) & for\ i > 0 \end{cases}$$

Lemma 7.1 *If* $\forall j > 0$*:* $a(j+1) \geq a(j) + \mathcal{E}(j)$ *and* $\Theta = \Theta(i)$*, the behaviour of a component is:*

$$f(i) = a(i) + \Theta$$

Proof. We prove this lemma by induction. For $i = 0$, the lemma is satisfied according to Definition 7.2. The proof concludes if the lemma is satisfied for $i+1$ and assuming that the lemma is true for i:

$$\begin{aligned} f(i+1) &= \max(a(i+1) + \Theta, f(i) + \mathcal{E}(i)) \\ a(i+1) + \Theta &\geq f(i) + \mathcal{E}(i) \end{aligned}$$

Substitute $f(i)$ by $a(i) + \Theta$:

$$\begin{aligned} a(i+1) + \Theta &\geq a(i) + \Theta + \mathcal{E}(i) \\ a(i+1) &\geq a(i) + \mathcal{E}(i) \end{aligned}$$

This equality is true according to the condition of the lemma.

Lemma 7.2 *If* $\forall j > 0$*:* $a(j+1) \geq a(j) + \mathcal{E}(j)$ *and* $\hat{\Theta} \geq \Theta(i)$*, the behaviour of a component is:*

$$f(i) \leq a(i) + \hat{\Theta}$$

Proof. We prove this lemma by induction again. For $i = 0$, the lemma is satisfied according to Definition 7.2 and $\hat{\Theta} \geq \Theta(i)$. The proof concludes if the lemma is satisfied for $i+1$ and assuming that the lemma is true for i:

$$f(i+1) = \max(a(i+1) + \Theta(i+1), f(i) + \mathcal{E}(i))$$

The left part of the maximum is satisfied:

$$\begin{aligned} a(i+1) + \hat{\Theta} &\geq a(i+1) + \Theta(i+1) \\ \hat{\Theta} &\geq \Theta(i+1) \end{aligned}$$

The right part of the maximum is satisfied by substitution of $f(i)$ by $a(i) + \hat{\Theta}$

$$\begin{aligned} a(i+1) + \hat{\Theta} &\geq f(i) + \mathcal{E}(i) \\ a(i+1) + \hat{\Theta} &\geq a(i) + \hat{\Theta} + \mathcal{E}(i) \\ a(i+1) &\geq a(i) + \mathcal{E}(i) \end{aligned}$$

This equality is true according to the first condition of the lemma.

Figure 7.3 shows the data flow of the memory controller. This can be used as a reference for the symbols and connections between components of this and Section 8. Components that have an instance for each requestor, are suffixed by $[r]$. Only one component is drawn to simplify the figure. The number of requestors is defined as R. Furthermore, all requestors are ordered from 0 to $R-1$ by priority, where a lower index has a higher priority. Table 7.2 defines the arrival and finishing times that are used in the model. In the next sections, the behaviour of the components are defined, such that the memory controller is predictable and composable. Section 8 shows how this behaviour is implemented.

According to Section 2.1.2, predictable components must have predictable subcomponents. From the perspective of the data flow model, the memory is a subcomponent of the back-end. The back-end is a subcomponent of the back-end interface and so on. The total behaviour of the front-end is captured by the requestor interfaces. For this reason we analyse the data flow bottom-up; from memory to requestor interfaces.

7.2.1 Back-end

The timing behaviour of the back-end depends on the SDRAM commands that are being sent to the memory and the memory itself. Section 6 explained that it is not efficient to send single bursts to the memory when bounded timing behaviour is required. Therefore, the access granularity of the memory is defined by memory accesses.

Figure 7.3(b) shows the data flow model of the back-end. The pattern scheduler is responsible for the mapping from memory accesses to patterns. The PAM is used if no requestor requires composability, otherwise the CAM must be used to avoid dependencies between memory accesses. When the patterns that have to be issued for the pending memory access are known, the pattern scheduler generates the SDRAM commands for the memory according to the pattern definition. Since a request consists of a logical address, the pattern scheduler also performs the memory mapping. The write data delay assures that the write data is synchronized with the write commands of the scheduled patterns, since the write data could arrive earlier. According to the SDRAM commands that have been sent to the memory, read data is produced and sent back to the back-end. The back-end passes the read data to the back-end interface.

Lemma 7.1 is used to define the timing behaviour of the pattern scheduler in Definition 7.3. The behaviour of the read data delay is defined by Definition 7.4. The latency ($\Theta_{read-data}(i)$) and execution time ($\mathcal{E}_{ps}(i)$) depend on the patterns that need to be executed for the memory accesses of request i. Section 6.4.1 (page 41) and Section 6.4.2 (page 46) explain how to derive upper bounds for latency and rate.

The constant delay of the pattern scheduler (Θ_{ps}) enables it to wait for a full burst. This is necessary because the memory cannot write less than one burst. If not all data of the burst is available, invalid data is written to the memory. Using Lemma 7.2 and the behaviour of the subcomponents, Definition 7.5 shows the behaviour of the back-end. The latency of a write request is not defined because no data or other results are returned by the memory. A response for a write request is generated by the back-end interface.

The minimum rate of the back-end depends on the length of the interval. A higher rate can be guaranteed for larger intervals as explained in Section 6.4.3.2 on page 49.

Table 7.2: Symbols of the data flow model of the memory controller

symbol	*meaning*
$a_{ipd[r]}(j)$	Arrival time of the j'th request at the r'th initiator protocol decoder
$f_{ipd[r]}(j)$	Finishing time of the j'th request at the r'th initiator protocol decoder
$a_{ipe[r]}(j)$	Arrival time of the j'th request (response) at the r'th initiator protocol encoder
$f_{ipe[r]}(j)$	Finishing time of the j'th request (response) at the r'th initiator protocol encoder
$a_{arb[r]}(j)$	Arrival time of the j'th request at the arbiter
$f_{arb[r]}(j)$	Finishing time of the j'th request (response) of the r'th requestor at the arbiter
$a_{sched[r]}(j)$	Arrival time of the j'th request of the r'th requestor at the scheduler of the CCSP arbiter
$a_{rsd[r]}(j)$	Arrival time of the j'th request (response) at the r'th response delay block of the arbiter
$f_{rtb[r]}(j)$	Finishing time of the j'th request (release time of response) of the r'th requestor at the response time buffer
$f_{rib}(i)$	Finishing time of the i'th request (information for response) at the response info buffer
$a_{bei}(i)$	Arrival time of the i'th request at the back-end interface
$f_{bei}(i)$	Finishing time of the i'th request (response) at the back-end interface
$a_{be}(i)$	Arrival time of the i'th request at the back-end (one or more memory accesses)
$a_{be-wr}(i)$	Arrival time of the write data of the i'th request at the back-end (one or more memory accesses)
$f_{be-rd}(i)$	Finishing time of the read data of the i'th request at the back-end
$a_{mem}(i)$	Arrival time of the i'th request at the memory (one or more memory command patterns)
$a_{mem-wr}(i)$	Arrival time of the write data of the i'th request at the memory (one or more memory accesses)
$f_{mem-rd}(i)$	Finishing time of the read data of the i'th request at the memory

The minimum rate of the back-end is defined as $\check{\rho}_{backend}$, according to Definition 7.6.

Definition 7.3 (Behaviour of the pattern scheduler) *If* $\forall k > 0$*:* $a_{be}(k+1) = a_{be}(k) + \mathcal{E}_{ps}(k)$*, the behaviour of the pattern scheduler is:*

$$a_{mem}(i) = a_{be}(i) + \Theta_{ps}$$

where: Θ_{ps} Latency of the pattern scheduler
$\mathcal{E}_{ps}(k)$ Execution time of all the successive memory accesses of request k. Equations (6.15) (page 45) and (6.20) (page 48) are used to calculate the maximum execution time for a single request for a PAM and CAM, respectively.

Definition 7.4 (Behaviour of the read data delay) *The finishing time of a read request at the read data delay is:*

$$f_{be-rd}(i) = \begin{cases} f_{ps}(i) + \Theta_{read-data}(i) & if\ type(i) = read \\ undefined & if\ type(i) = write \end{cases}$$

where: $\Theta_{read-data}(i)$ First data latency of the memory. Equation (6.18) (page 45) shows how to compute the maximum latency.

Definition 7.5 (Behaviour of the back-end) *If* $\forall k > 0$*:* $a_{be}(k+1) = a_{be}(k) + \mathcal{E}_{ps}(k)$*, the finishing time of a request at the back-end is:*

$$f_{be-rd}(i) \leq \begin{cases} a_{be}(i) + \hat{\Theta}_{be} & if\ type(i) = read \\ undefined & if\ type(i) = write \end{cases}$$

where: $\hat{\Theta}_{be} = \Theta_{ps} + \hat{\Theta}_{read-data}$

Definition 7.6 (Minimum rate of the back-end) *The minimum rate of the back-end is defined as* $\check{\rho}_{backend}$*.*

where: $\check{\rho}_{backend}$ corresponds to $\check{\rho}$ in Equation (6.13) on page 44.

7.2.2 Back-end interface

The back-end interface is also part of the data flow model in Figure 7.3(b). Requests are split into memory accesses and write data by the target protocol encoder. The target protocol decoder receives the read data from the back-end and response information from the response info buffer. When the head of the buffer indicates a read request, a response is created from the read data. A write response is only constructed from the information in the buffer and contains no data. After the last word of the response has been sent, the information for that response is popped from the buffer, such that the information for the next request is visible.

The behaviour of the target protocol encoder and decoder are defined by Definition 7.7 and 7.8. The behaviour of the target protocol decoder is not simplified using Lemma 7.1. The required condition cannot always be satisfied, because the arrival time of the responses ($f_{rib}(i)$ and $f_{be-rd}(i)$) cannot be enforced. However, Section 8.5 explains that the latency of the complete back-end can be bounded according to Definition 7.9. This behaviour of the back-end interface is only guaranteed when requests are not arriving too fast, such that no request is stalled by the preceding one. Write requests of the back-end interface potentially have a lower latency, because they do not have to wait for write data.

Definition 7.7 (Behaviour of target protocol encoder) *If* $\forall k > 0$*:* $a_{bei}(k+1) = a_{bei}(k) + \mathcal{E}_{ps}(k)$*, the behaviour of the target protocol encoder is:*

$$\begin{aligned} a_{be}(i) &= a_{bei}(i) \\ a_{be-wr}(i) &= a_{bei}(i) \end{aligned}$$

Definition 7.8 (Behaviour of target protocol decoder)

$$f_{bei}(i) = \begin{cases} \max(f_{rib}(i), f_{be-rd}(i), f_{bei}(i-1) + \mathcal{E}_{tpd}(i-1)) & if\ type(i) = read\ and\ i > 0 \\ \max(f_{rib}(i), f_{bei}(i-1) + \mathcal{E}_{tpd}(i-1)) & if\ type(i) = write\ and\ i > 0 \end{cases}$$

where:	$\mathcal{E}_{tpd}(i)$	The execution time of the target protocol decoder for request i
	$f_{bei}(-1) + \mathcal{E}_{tpd}(-1)$	$\leq f_{rib}(0)$

Definition 7.9 (Behaviour of back-end interface) *If* $\forall k > 0$*:* $a_{bei}(k+1) = a_{bei}(k) + \mathcal{E}_{ps}(k)$*, the behaviour of the back-end interface is:*

$$f_{bei}(i) \leq a_{bei}(i) + \hat{\Theta}_{bei}$$

where: $\hat{\Theta}_{bei} = \hat{\Theta}_{be}$

7.2.3 Arbiter

The arbiter contains a scheduler, switch and response delay (Figure 7.3(a)). The scheduler selects a requestor and sends the first request to the back-end interface. The switch is used to route the response to the response delay block of the corresponding requestor. The response delay block is bypassed when composability is not required. Otherwise, the response delay block is delaying the response before it is returned to the requestor interface. The purpose is explained later in this section.

The scheduler must guarantee bounded service for every requestor. Examples of such schedulers are: Weighted round-robin [14], Deficit round-robin [23], Credit-Controller Static-Priority (CCSP) [2] and Time Division Multiplexing (TDM). In general, schedulers that belong to the class of $\mathcal{LR}$ servers can be used, because $\mathcal{LR}$ servers guarantee a lower bound on service. Our implementation uses the CCSP arbiter that behaves according to Definition 7.11. The CCSP arbiter guarantees that the latency of a request is bounded (Definition 7.12). Proof, requirements and latency bounds for preemption and work-conserving can be found in [2]. The latency bound depends on the allocated service of the requestors that have a higher priority. The allocated service can be configured for every requestor ($[r]$) and is characterized by burstiness ($\sigma'_{[r]}$) and rate ($\rho'_{[r]}$).

7.2.3.1 Virtual time and data to real time and data

However, the CCSP arbiter guarantees bounds on a virtual time and data unit, which we refer to as *service cycle* and *service unit*, respectively. A service cycle corresponds to the

time to serve one service unit. A scheduler issues one service unit for every service cycle. To be able to use the service guarantees, an implementation is needed that conforms to the model, but also maps the service cycle and units to the real time and data domain. In addition, the translation must be bounded for a predictable arbiter.

The approach for the front-end is to create the scheduler in such a way that the virtual clock can be controlled by the arbiter. A service unit is mapped to a memory access and service cycles are mapped to the time to execute that memory access. Furthermore, a non-preemptive scheduler is considered, such that the memory accesses of a request are scheduled successively without interruption of memory accesses of other requests. When no request is scheduled, a virtual request of one idle access must be scheduled.

Based on this approach, the scheduler issues requests at the same rate as the pattern scheduler: $a_{bei}(i+1) = a_{bei}(i) + \mathcal{E}_{ps}(i)$. This assures that the request is not delayed, because the back-end interface or back-end is not yet ready to start with a new request. The condition of Definition 7.9 is guaranteed to be true.

Definition 7.10 (Behaviour of the CCSP rate regulator) *When the requested service of requestor r does not exceed the allocated service, the behaviour of the rate regulator of the CCSP arbiter is:*

$$a_{sched}(i) = a_{arb[r]}(j)$$

Definition 7.11 (Behaviour of the CCSP scheduler) *The behaviour of the scheduler of the CCSP arbiter for request j of requestor r is:*

$$a_{bei}(i) = a_{arb[r]}(j) + \Theta_{sched[r]}(j)$$

where: $\Theta_{arb[r]}(j)$ Latency of the scheduler for request j of requestor r

Definition 7.12 (Latency bound of CCSP) *A non-preemptive and non-work-conserving CCSP arbiter guarantees that the latency of requestor r is bounded by [2]:*

$$\hat{\Theta}_{sched[r]} = \frac{\hat{b}_{[r]} + \sum_{k=0}^{r-1} \sigma'_{[k]}}{1 - \sum_{k=0}^{r-1} \rho'_{[k]}}$$

where:
- $\hat{b}_{[r]}$ $\max_{k=r}^{R-1}(\hat{s}_{[k]}) - 1$
- $\hat{s}_{[k]}$ Maximum size of a request of requestor k
- $\sigma'_{[k]}$ Allocated burstiness of requestor k
- $\rho'_{[k]}$ Allocated rate of requestor k as a fraction of the total available rate ($\check{\rho}_{backend}$, Definition 7.6)

Definition 7.13 (Behaviour of the response time buffer) *The behaviour of the response time buffer is defined as:*

$$f_{rtb[r]}(j) = a_{sched[r]}(j) + \hat{\Theta}_{arb[r]}$$

where: $\hat{\Theta}_{arb[r]} = \hat{\Theta}_{sched[r]} + \hat{\Theta}_{bei[r]} + t_{clk}$

Definition 7.14 (Behaviour of response info buffer) *The behaviour of the response info buffer is defined as:*

$$f_{rib}(i) = a_{bei}(i) + \Theta_{rib}(i)$$

where: $t_{clk} \leq \Theta_{rib}(i) \leq \hat{\Theta}_{bei}$

Definition 7.15 (Behaviour of the demultiplexer) *The behaviour of the demultiplexer is:*

$$a_{rsd[r]}(j) = f_{bei}(i)$$

Definition 7.16 (Behaviour of the response delay block)

$$f_{arb[r]}(j) = \begin{cases} a_{sched[r]}(j) + \hat{\Theta}_{arb[r]} & \textit{in composable mode} \\ a_{rsd[r]}(j) & \textit{otherwise} \end{cases}$$

where: $\hat{\Theta}_{arb[r]} = \hat{\Theta}_{sched[r]} + \hat{\Theta}_{bei[r]} + t_{clk}$

Definition 7.17 (Behaviour of arbiter) *When the requested service of requestor r does not exceed the allocated service, the behaviour of the arbiter is:*

$$f_{arb[r]}(j) = a_{arb[r]}(j) + \Theta_{arb[r]}(j)$$

where: $\Theta_{arb[r]}(j) = \Theta_{sched[r]}(j) + \Theta_{bei}(i) + \Theta_{rsd[r]}(j)$

The behaviour of the demultiplexer is described by Definition 7.15. It does not add a delay to a response. The only purpose of the switch is to route responses to the response delay block. The latency of the response delay block is defined by Definition 7.16. The delay is bypassed when composability is not required. The behaviour of the complete arbiter is shown by Definition 7.17. The latency that is added to a request is $\Theta_{bei}(i)$, $\Theta_{sched[r]}(j)$ and $\Theta_{rsd[r]}(j)$. Definitions 7.9, 7.12 and 7.16 show that these latencies are bounded at design time. The rate for every requestor is also bounded because the non-work-conserving CCSP arbiter guarantees that each requestor gets the allocated rate. As mentioned earlier, the back-end interface does not affect this rate, because the scheduler assures that requests are scheduled in the same rate as the back-end interface. This concludes that the arbiter is predictable. A requestor that request more service than is allocated, is slowed down by the rate regulator of the CCSP arbiter. This assures that such requestors do not affect the service of others. However, the latency of the rate regulator is no longer zero. The exact latency depends on the burstiness and rate of the requestor.

7.2.3.2 Composability

The design is not composable when the PAM is used and response delay is bypassed for the following four reasons:

1. The allocated rate of a requestor ($\rho'_{[r]}$) is affected by other requestors. The allocated rate for the CCSP is a fraction of the rate of the back-end. However, the rate of the back-end is affected by memory accesses of other requestors as explained in Section 6.4.2 on page 46. Hence, the allocated rate of a requestor is also affected.

2. The absence of requests of higher priority requestors affects the behaviour of other requestors. If a requestor does not always have a pending request, lower priority requestors are scheduled earlier. Hence, the behaviour of requestors depend on the traffic of others.

3. The latency of the scheduler for a request ($\Theta_{sched[r]}(j)$) is affected by other requestors. Latency is decreased when there is less interference from other requestors.

4. The latency of the back-end interface for a request ($\Theta_{bei}(i)$) is affected by other requestors. The latency is the time between the start of a request and when the first data is returned. This time depends on the depends on the execution time of a memory access. Like the first reason, the execution time of a memory access depends on requests of other requestors.

A CAM must be used to make the allocated rate independent of other requestors (reason 1). This mapping has the property that the rate of the back-end is constant at run-time, such that the allocated rate is also constant and not affected by others requestors. For the same purpose, a work-conserving arbiter cannot be used, because that tolerates that a requestor benefits from unused service of others.

To solve issue 2, the arrival time of a request must be delayed to the worst-case such that it cannot be scheduled earlier. Hence, when another requestor does not have pending requests, the scheduling time of a lower priority requestor is not affected. However, this is not trivial for the CCSP arbiter, because it has an dynamic scheduler. Currently, there is no implementation of this solution. For the simulations, this problem is circumvented by requestors that always have pending requests.

The problems regarding latency (3 and 4) are solved by enforcing a constant latency which isolates a requestor from the behaviour of others. This is accomplished by delaying every request to the worst-case latency of the corresponding requestor. The response delay block is responsible for this job (Definition 7.16). This definition shows that the response delay block accounts for the delay of the scheduler and back-end interface. It could be possible to reduce the delay for the back-end interface, because the main variation of the delay is caused by the request itself. However, when responses belonging to the same requestor are delayed by a variable amount, it could be necessary to reorder successive responses. Reordering is not only hard to implement but also makes the analysis more difficult.

Delaying a response at the end of the flow is the most easy solution. A request cannot be delayed right after it has been scheduled, because this affects the time schedule of the memory and could lead to a situation that two requests must be sent to the back-end interface at the same time. Delaying before the scheduler is also not possible because the actual scheduler latency is not known at that time such that the additional delay for the request cannot be determined.

Note that the worst-case latency of the CCSP arbiter depends on the allocated service of requestors that have a higher priority. When the service is reconfigured at run-time, the response delay block must also be reconfigured.

It could be possible that not all requestors need composability. In this case, the response delay block of the corresponding requestor(s) can be bypassed. This is a major advantage for requestors that benefit from a low average latency, because the average latency is much lower than the worst case according to the experiments of Section 9. Unfortunately, such requestors cannot get a higher rate, because the back-end cannot use a mix of composable and predictable accesses.

7.2.4 Requestor interfaces

The data flow model in Figure 7.3(a) shows that each requestor interface consists of an initiator protocol decoder and encoder. The initiator protocol decoder is responsible for decoding the requests of a requestor. The responses are encoded to the protocol of the initiator by the initiator protocol encoder. As mentioned earlier, the requestor interfaces are considered to be part of the interconnect and the remainder of the front-end belongs to the memory controller. To decouple both resources, buffers are inserted between the requestor interfaces and arbiter.

Composability is an easy requirement for the requestor interfaces. Every requestor has its own initiator protocol decoder and encoder, such that the behaviour cannot be affected by other requestors. The behaviour of the decoder and encoder are given by Definition 7.18 and 7.19, respectively. Both converters must have a bounded latency for a request or response at design time to guarantee a predictable front-end. The latency is bounded when requests are arriving at the same rate as the decoder and encoder according to Lemma 7.2.

The average rate of arriving responses at the encoder is maximally the allocated rate, because of the rate regulator of the arbiter. Hence, the average execution time of the encoder must be lower than the average time between arriving requests. In addition, the interconnect is not allowed to stall the encoder. In both cases the buffer between the arbiter and encoder get full. Eventually, the latency bounds cannot be guaranteed anymore and deadlines may be missed.

The arbiter guarantees bounded latency according to Definition 7.17, when the requested service does not exceed the allocated service. When the execution time of the initiator protocol decoder is too low, the requestor cannot get its allocated rate. Hence the front-end is not predictable.

Definition 7.18 (Behaviour of the initiator protocol decoder) *The behaviour of the initiator protocol decoder is:*

$$f_{ipd[r]}(j) = \begin{cases} a_{ipd[r]}(j) + \Theta_{ipd[r]}(j) & for\ i = 0 \\ \max(a_{ipd[r]}(j) + \Theta_{ipd[r]}(j), f_{ipd[r]}(j-1) + \mathcal{E}_{ipd[r]}(j-1)) & for\ i > 0 \end{cases}$$

where: $\Theta_{ipd[r]}(j)$ Latency of initiator protocol decoder r for request j
$\mathcal{E}_{ipd[r]}(j)$ Execution time of initiator protocol decoder r for request j

Definition 7.19 (Behaviour of the initiator protocol encoder) *The behaviour of the initiator protocol decoder is:*

$$f_{ipe[r]}(j) = \begin{cases} a_{ipe[r]}(j) + \Theta_{ipe[r]}(j) & for\ i = 0 \\ \max(a_{ipe[r]}(j) + \Theta_{ipe[r]}(j), f_{ipe[r]}(j-1) + \mathcal{E}_{ipe[r]}(j-1)) & for\ i > 0 \end{cases}$$

where: $\Theta_{ipe[r]}(j)$ Latency of initiator protocol encoder r for request j
$\mathcal{E}_{ipe[r]}(j)$ Execution time of initiator protocol encoder r for request j

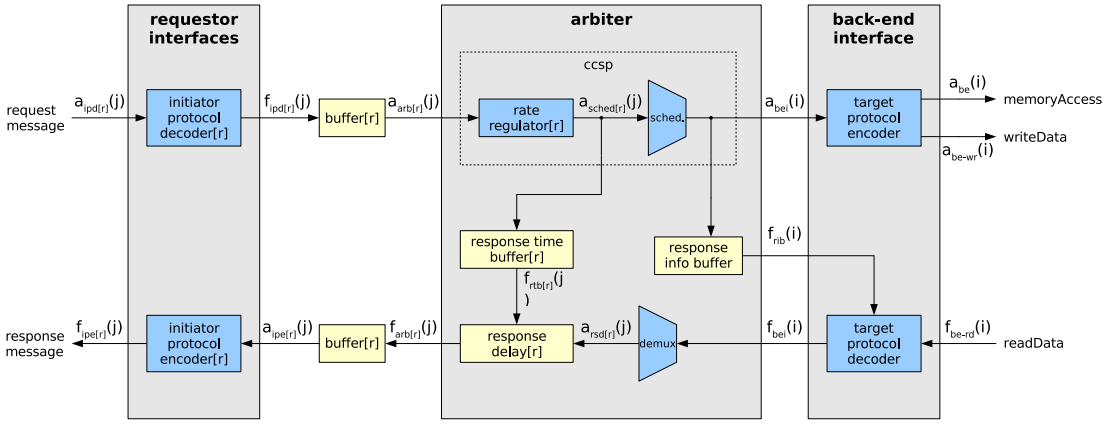
7.3 Conclusions

This section proposed an architecture that makes a strong separation between the front-end and back-end to reduce dependencies with the memory. The separation of concerns improves reusability of the front-end and reduces the design effort. From a perspective of resources, the requestor interfaces belong to the interconnect and the arbiter and back-end interface are part of the memory controller.

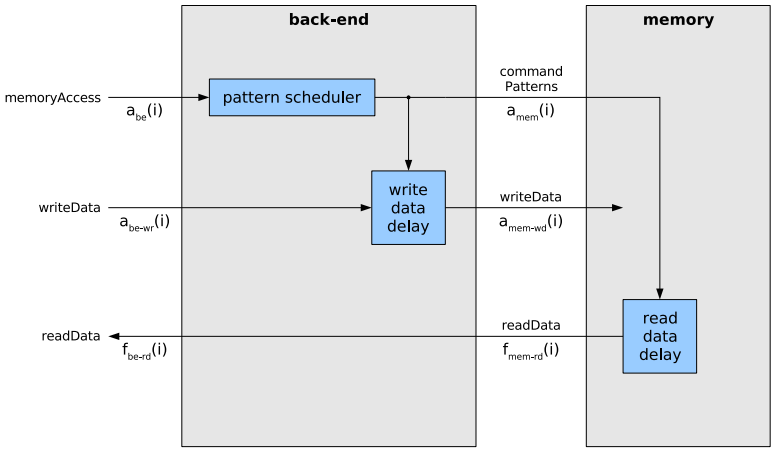
Assuming that no component is stalled, simple data flow models can be used to analyse the behaviour of the memory controller. For predictability, the back-end has to use the PAM. In addition, the behaviour of the scheduler and requestor interfaces must be bounded.

Based on the predictable design, the memory controller is made composable by using a CAM to prevent that the behaviour is affected on the level of memory accesses. The response delay block assures that the interference from other requestors does not affect the latency of a request. This front-end is composable when there are always pending requests.

The maximum latency of the memory controller (excluding the requestor interfaces) for requestor r is $\hat{\Theta}_{arb[r]} = \hat{\Theta}_{sched[r]} + \hat{\Theta}_{bei} + t_{clk}$. This bound is only valid when the requested service does not exceed the allocated service, characterized by $\sigma'_{[r]}$ and $\rho'_{[r]}$. The minimum net bandwidth is $width_{ra} \cdot \rho'_{[r]} \cdot \check{\rho}_{backend}$.



(a) Front-end



(b) Back-end

Figure 7.3: Data flow of the memory controller

8 Implementation

From the design discussed in Section 7, a hardware implementation has been implemented in VHDL. This section shows how the design has been mapped to an implementation and which problems that arise. The hardware model is used for experiments to check if the requirements are satisfied (Section 9).

8.1 Functional behaviour

Figure 8.1 shows the block diagram of the implementation of the front-end. Only the most important components and signals are shown in this figure. Components and signals are appended with $[r]$ to indicate that there is an instance for each requestor. The arbiter block of the data flow model maps to the main part of the controller, multiplexer, demultiplexers and response delay blocks (Figure 7.3, page 75). For each requestor, the requests arrive at the corresponding initiator protocol decoder. This ensures that the request is decoded to a format ($writeData[r]$ and $requestInfo[r]$) that can be used by the controller and target protocol encoder. A decoded request is stored in the appropriate request buffer. Based on request information and other rules, the arbiter inside the controller determines which request is scheduled ($scheduledRequestor$). The multiplexer routes the scheduled request from the corresponding request buffer to the target protocol encoder. When the controller notifies the target protocol encoder ($sendRequest$), the request is encoded and sent to the back-end ($writeData$ and $command$). The back-end returns read data ($readData$) when a read request has been executed. Using information derived from the request ($responseInfo$) and potential read data, the target protocol encoder constructs a read or write response ($readData$ and $responseInfo$). The demultiplexer routes the response back to the response delay block of the corresponding requestor by the selection signal ($destinationRequestor$). The response delay block stores the response until the controller decides to release the response ($sendResponse$). At this point, the response is streamed into the response buffer. The initiator protocol encoder reads responses from that buffer and translates it to the appropriate format for the response queue of the interconnect.

Before a response leaves the front-end, the next request can already be scheduled to enable pipelining. The configuration register contains the run-time configurable settings. It can be programmed by the configuration port ($configurationData$). The front-end can be configured as predictable or predictable and composable at design-time. Some subcomponents are not necessary for a predictable only component and some components need to be configured differently. The next sections provide detailed information of the components. The same partitioning as the data flow model of Figure 7.3 is used for the discussion.

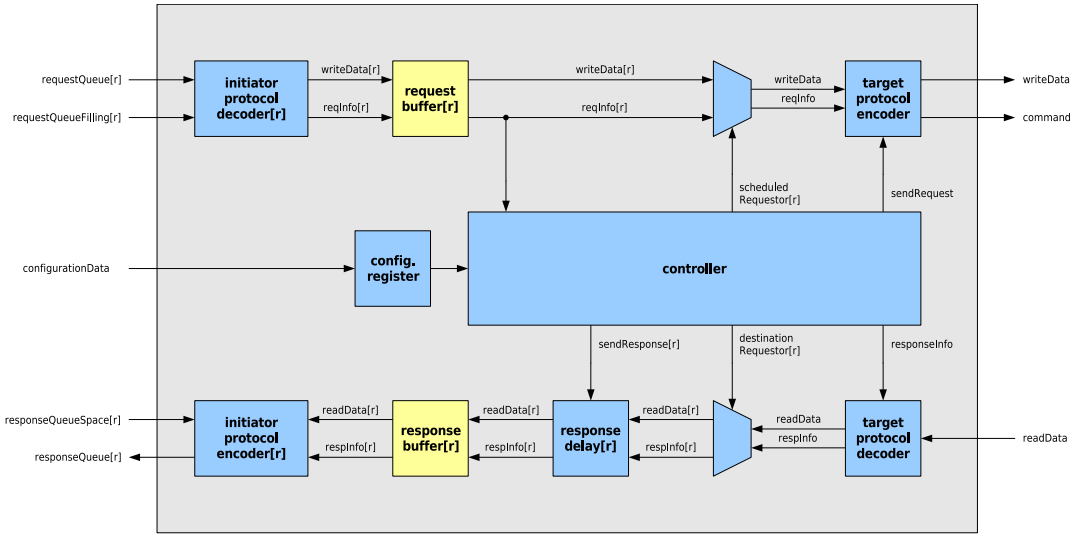


Figure 8.1: Block diagram of front-end

8.2 Request and response format

Throughout the design of the front-end, requests and responses represent the the same information, but in a different format. Internally, requests are sent by a stream with request information and a stream of data to write. Responses are transmitted using a stream with response information and a stream of data that has been read. Tables 8.1, 8.2, 8.3 and 8.4 list the content of the four streams.

According to Definition 7.1 (page 64), a request or response arrives at a component when the producer can guarantee an uninterrupted stream. The first word of *requestInfo* and potential *writeData* must be visible. The producer must guarantee that, the remaining data can be sent to the consumer without empty cycles. Responses have the same requirements for *responseInfo* and *readData*.

The size of the read and write data words are equal to the word size of the back-end ($width_{tgt}$) to prevent data-width conversion by the back-end interface. The data width of the back-end is twice the width of the memory data bus because a DDR memory is used. The width of the request and response queues that are accessed by the requestor interface is $width_{queue}$. Furthermore, the data width of original protocol of the requestor ($width_{ini}$) is equal or greater than $width_{tgt}$ and $width_{queue}$. Finally, all word sizes are a power of two to simplify the hardware implementation. Data widths are expressed in bits. The size of a request represents how much data is accessed. $size_{ini}(i)$ and $size_{tgt}(i)$ denote the number of words of request i in terms of $width_{ini}$ and $width_{tgt}$, respectively.

8.3 Requestor interface

The template of the SoC (Figure 1.2) uses a network as interconnect. To be able to support older IP components that have a bus interface (like AXI), the initiator network interface serializes the bus protocol to messages [22]. At the memory controller (target),

the network interface converts the messages back to the original bus protocol. To prevent overhead due to protocol conversions in the network interface and requestor interface, the requestor interface uses the serialized bus protocol, such that it is not necessary to convert the messages back to the bus protocol. The incoming messages of the network interfaces are stored in the *requestQueue*, and the messages going back to the network are stored in the *responseQueue*. The initiator protocol decoder and encoder form the requestor interface. Currently, the only supported format is the serialized AXI protocol. The format of the request and response messages are shown in Figure 8.2. More details and limitations can be found in Appendix C. The size of the messages are defined in Definitions 8.1 and 8.2.

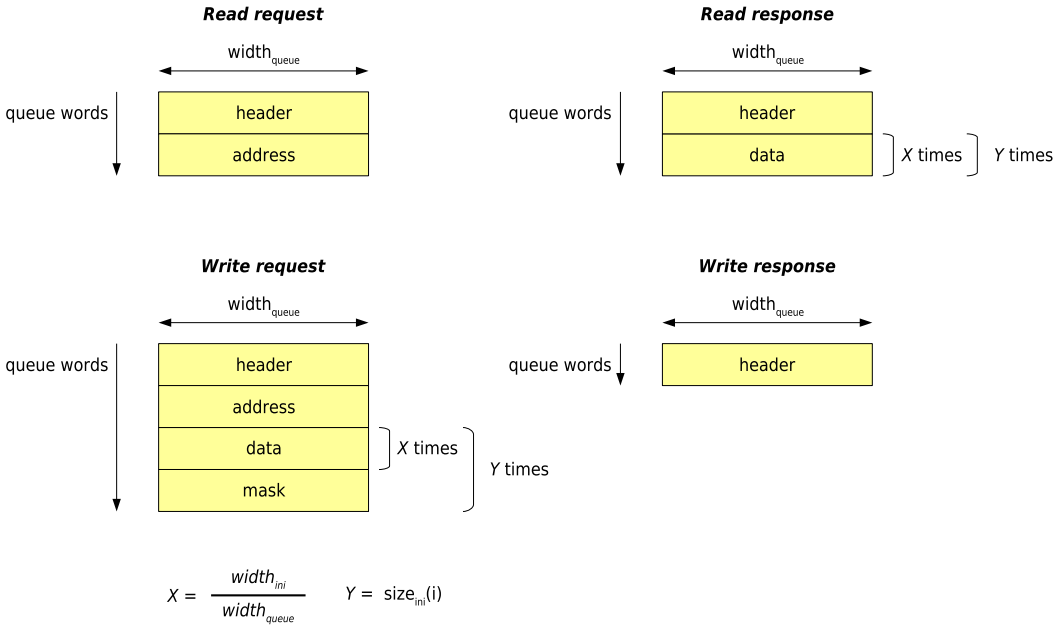


Figure 8.2: Format of the serialized AXI protocol

Definition 8.1 (Size of a request message) *The number of words (where a word has size $width_{queue}$) of the request i for the serialized AXI protocol is:*

$$requestMsgSize(i) = \begin{cases} 2 & if\ type(i) = read \\ 2 + size_{ini}(i) \cdot \left(\frac{width_{ini}}{width_{queue}} + 1\right) & if\ type(i) = write \end{cases}$$

Definition 8.2 (Size of a response message) *The number of words (where a word has size $width_{queue}$) of response i for the serialized AXI protocol is:*

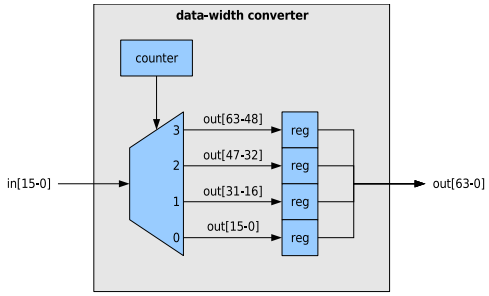
$$responseMsgSize(i) = \begin{cases} 1 + size_{ini}(i) \cdot \frac{width_{ini}}{width_{queue}} & for\ a\ read\ response \\ 1 & for\ a\ write\ response \end{cases}$$

8.3.1 Data-width converter

The protocol encoder and decoder both use data-width converters to convert the width of the read and write data streams, because the requestors could have a different data width than the arbiter expects. When the data width must be enlarged, the converter of Figure 8.3(a) is used. The data width of the output is a multiple of the input data width. A number of words from the input are stored in separate registers. At the time enough words have arrived, the words are made visible at the output as one larger word. Figure 8.3(b) shows how incoming words are merged into one output word. This figure also illustrates that not all cycles at the output contain data because the input cannot deliver data fast enough.

The converter of Figure 8.4(b) is used when the data width must be reduced. The input word is split into smaller words (equal to the data width of the output) and stored in registers. After that, the words from the registers are sent one by one to the output. The timing of the input and output words is shown in Figure 8.3(b). In contrast with the data-width converter that increases the data width, the input is stalled, because it takes more time to transmit the same amount of data.

Both converters are capable of bypassing the register to avoid an additional cycle latency. The first word of Figure 8.3(b) is visible in cycle 3 instead of cycle 4.



(a) Block diagram

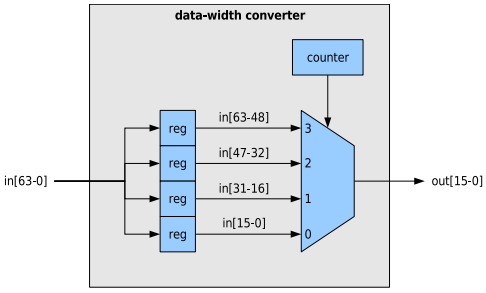
cycle	0	1	2	3	4	5	6	7	8	9	10	11	12
in (16 bit)	0.0	0.1	0.2	0.3	1.0	1.1	1.2	1.3	2.1	2.2	2.3	2.4	
out (64 bit)				0				1				2	

(b) Data stream

Figure 8.3: Data-width converter for 16 to 64 bits

8.3.2 Initiator protocol decoder

Figure 8.5 shows the main parts of the initiator protocol decoder. The decoder reads request messages from the request queue of the interconnect. The messages are decoded to the internal format of the front-end, specified by the *requestInfo* and *writeData* signal groups. In the header of a request message, the amount of data to access is expressed in initiator words ($size_{ini}$). This amount is converted to $size_{tgt}$, because the data words are resized by the data-width converter. The purpose of this conversion is to match the



(a) Block diagram

cycle	0	1	2	3	4	5	6	7	8	9	10	11	12
in (64 bit)	0	1				2							
out (16 bit)	0.0	0.1	0.2	0.3	1.0	1.1	1.2	1.3	2.1	2.2	2.3	2.4	

(b) Data stream

Figure 8.4: Data-width converter for 64 to 16 bits

data width of a request with the arbiter and back-end.

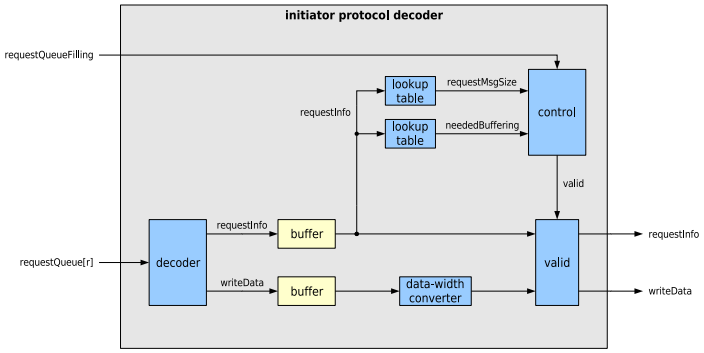


Figure 8.5: Block diagram of initiator protocol decoder

The initiator protocol decoder determines the time that a request arrives at the arbiter. The serialized AXI protocol cannot cope with the rate of the resource when both run at the same clock frequency. The main reason is the serial format of the protocol. The unbuffered example of Figure 8.6(a) illustrates this situation. Note that the *requestInfo* group is constructed using the header and address, and each *writeData* group needs the data and mask of the request message.

Buffering of the request before it is scheduled is needed to be able to provide an uninterrupted stream, and prevents violating the expected execution time of the request. In addition, SDRAM devices cannot be stopped to wait for data after a burst is started. The initiator protocol decoder contains separate buffers for *requestInfo* and *writeData*. The control block validates a request when an uninterrupted stream can be provided. This moment depends on the needed amount of buffering (*neededBuffering*), and how much already has been buffered. Request i is validated when the following conditions

are satisfied:

$$\begin{aligned} bufferFilling &\geq neededBuffering(i) \\ requestQueueFilling + bufferFilling &\geq requestMsgSize(i) \end{aligned}$$

The amount of needed buffering and the size of a message are stored in a lookup tables based on $size_{ini}$. The lookup tables have limited size because the serialized AXI protocol only supports requests up to 16 words. Figure 8.6(b) shows the effect of buffering and the associated timing. The latency of the initiator protocol decoder is equal to the time that the request is buffered. The upper bound on the latency is equal to the amount of buffering for a request that has the maximum size. Table 8.5 lists the needed buffering for write requests and bounds for common data-width combinations. It shows that a lot of buffering has to be done when the width of the queue is small compared to the initiator and target. Figure 8.6(b) also shows that a read request always needs to buffer two words, because the header and address must be extracted from the request message ($neededBuffering = 2$). The amount of buffering can be reduced to maximally two cycles when the mask is sent in parallel with the write data and the data width of the interconnect ($width_{queue}$) and back-end $width_{tgt}$ are equal. A higher clock frequency can be used if the interconnect has a smaller data width.

Buffering cannot increase the average incoming rate of requests however. The interconnect is responsible that it can transport the requested service of a requestor without decreasing the rate. This depends on the data width, request allocated rate and the clock frequency of interconnect and memory controller. Recall that the rate of requests may not exceed the allocated rate when a predictable latency is required. The current initiator protocol decoders do not support a clock bridge such that the maximum allocated rate is restricted. The execution time of a request by the request queue depends on the size of the message:

$$\mathcal{E}_{request_queue}(i) = requestMsgSize(i) \cdot t_{clk} \tag{8.1}$$

According to Definition 8.1, the execution time of read requests are much shorter than write requests, because all read request messages consists of two words. Section 9.3.1 shows an experiment that shows an example where the allocated rate cannot be provided because the execution time of requests is to high. For this use case, the maximum allocated service of a single requestor is 77% of the total net bandwidth.

The latency and execution time of the initiator protocol decoder are shown by Figure 8.6(b) and are equal to:

$$\Theta_{ipd}(i) = neededBuffering(i) \cdot t_{clk}$$

$$\mathcal{E}_{ipd}(i) = size_{tgt}(i) \cdot t_{clk}$$

From the figure can be concluded that the a new request can only finish when the last word of the previous request has been sent. Hence, this component behaves according to Definition 7.18 (page 73).

Table 8.1: Request information

field	*description*
type	Type of the request, write or read
size	Number of words to read or write from/to the back-end
id	General-purpose identifier for the request
address	Start address of the request (logical address)

Table 8.2: Write data

field	*description*
value	Write data word
mask	Mask to enable or disable bytes of the *value*

Table 8.3: Response information

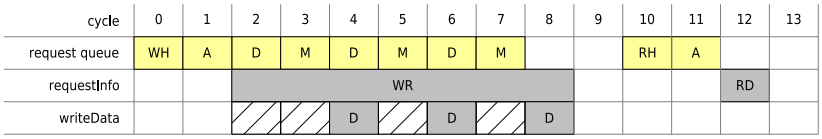
field	*description*
type	Type of the response, of a write or read request
size	Number of words read or written from/to the back-end
id	Identifier of the corresponding request

Table 8.4: Read data

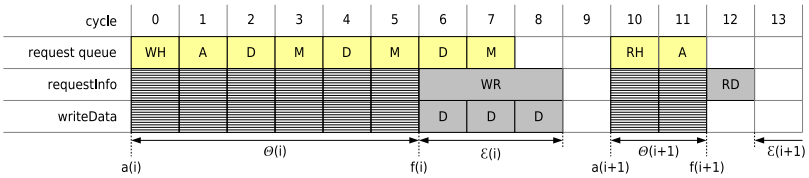
field	*description*
value	Read data word

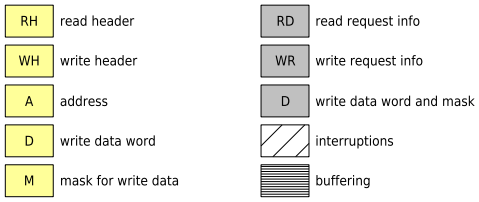
Table 8.5: Amount of buffering for write requests; interconnect and memory controller run at the same clock frequency

$width_{ini}$	$width_{queue}$	$width_{tgt}$	$neededBuffering$	*upper bound*
64	64	64	$size_{ini} + 3$	19
64	64	32	4	4
64	64	16	4	4
64	32	64	$2 \cdot size_{ini} + 3$	35
64	32	32	$size_{ini} + 4$	20
64	32	16	5	5
64	16	64	$4 \cdot size_{ini} + 3$	67
64	16	32	$3 \cdot size_{ini} + 4$	52
64	16	16	$size_{ini} + 6$	22
32	32	64	$3 \cdot (size_{ini}/2) + 3$	27
32	32	32	$size_{ini} + 3$	19
32	32	16	4	4
32	16	64	$5 \cdot (size_{ini}/2) + 3$	43
32	16	32	$2 \cdot size_{ini} + 3$	35
32	16	16	$size_{ini} + 4$	20
16	16	64	$7 \cdot (size_{ini}/4) + 3$	31
16	16	32	$3 \cdot (size_{ini}/2) + 3$	27
16	16	16	$size_{ini} + 3$	19



(a) Unbuffered (not implemented)





(b) Buffered

Figure 8.6: Timing behaviour of an initiator protocol decoder. $width_{queue} = width_{tgt}$

8.3.3 Initiator protocol encoder

After some time, the request is scheduled and a response arrives at the initiator protocol encoder through the response buffer. Like the request buffer, this decouples the arbiter from the requestor interface and avoids that the target protocol encoder directly stalls the arbiter block. This component is responsible for converting the *responseInfo* and *readData* back to a response message. The response message is sent to the response queue of the interconnect. The architecture consists of the following parts: the encoder and a data-width converter (Figure 8.8). Before the response of a read request is encoded, the data width of the read data is converted to the width of the response queue. The encoder composes a message from the *responseInfo* and *readData*. To support AXI initiators, the current version of the initiator protocol encoder implements the serialized AXI protocol. The format for the responses is shown by Figure 8.2. More details and limitations can be found in Appendix C.

Encoding of a response message has the same problem as decoding the requests messages: the transmission of a serialized response potentially cost more cycles than the internal format (Figure 8.7(a)). The header that is inserted costs one additional cycle. Stalling can occur when too much responses must be encoded. To avoid stalling the resource, the buffer between the arbiter and initiator protocol encoder must be large enough, and the average provided rate of the arbiter may not exceed the maximum rate of the initiator protocol encoder and interconnect. When a requestor misbehaves by issuing lots of read requests, the interconnect cannot handle the rate because a response message of a read request is much larger than the request message. In this situation, the buffer gets full and the arbiter does not schedule new requests anymore, until there

is enough space again. Misbehaving requestors are stalled instead of the arbiter and resource. The resource may never be stalled, because this affects other requestors as well and thus violates predictability and composability. If the words of the target are smaller than the response queue of the interconnect, the execution time of the initiator protocol encoder is always lower than the smallest amount of time between arriving response. Hence, the initiator protocol encoder does not cause back-pressure. Figure 8.7(b) illustrates this situation.

The latency and rate of the initiator protocol encoder can be derived from the figures:

$$\Theta_{ipe}(i) = \max(0, size_{tgt}(i) - responseMsgSize(i)) \cdot t_{clk}$$

$$\mathcal{E}_{ipe}(i) = responseMsgSize(i) \cdot t_{clk}$$

The latency is caused by data-width conversion. It is zero when the data width of the target ($width_{tgt}$) is not smaller than the response queue. The execution time depends on the size of the response message. Write responses are executed in less time than a read response, because a write response message only consists of one word (Figure 8.7).

Like the initiator protocol decoder, multiple words cannot be sent in the same cycle. In this case the latency of a request is increased. Along with the assumption that the interconnect does not cause back-pressure, this component behaves according to Definition 7.19 (page 74).

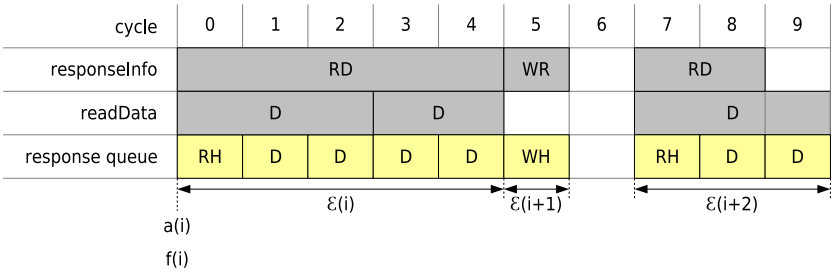
8.4 Arbiter

This section discusses the arbiter of the data flow model. It includes the response delay block, demultiplexer, multiplexer and a large portion of the controller, depicted in Figure 8.1. The subsections discuss the controller and response delay block in detail. The implementation of the demultiplexer and multiplexer is trivial. They consist of combinational logic only and not introducing a delay on the cycle level. Therefore, the demultiplexer behaves according to Definition 7.15 (page 71). The multiplexer is part of the scheduler of the CCSP arbiter (see Figure 7.3(a)).

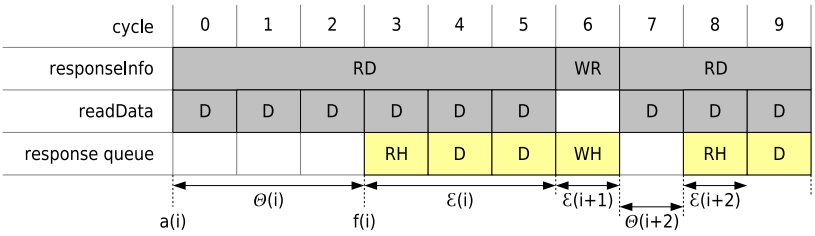
The controller consists of several subcomponents as shown in Figure 8.9. This figure shows only the most important signals. The main task of the controller is to schedule a request and route responses back to the corresponding requestor by controlling the components of the front-end (Figure 8.1). The controller does not change the content of the request and response streams.

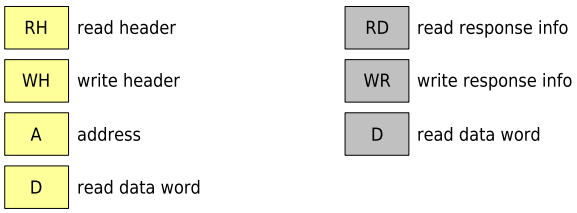
8.4.1 Storable response checker

The storable response checker verifies that the buffers of the response delay block have enough space for the response of the pending requests (request at the head of the request buffer). The free space of the buffers cannot be checked directly, because there can be requests in the pipeline that are not yet in the response delay block. Therefore, the storable response checker reserves space for every request that is scheduled. A response can be stored in the response delay block if the response fits in the unreserved space.



(a) $width_{tgt} = 2 \cdot width_{queue}$





(b) $width_{queue} = 3 \cdot width_{tgt}$

Figure 8.7: Timing behaviour of an initiator protocol encoder

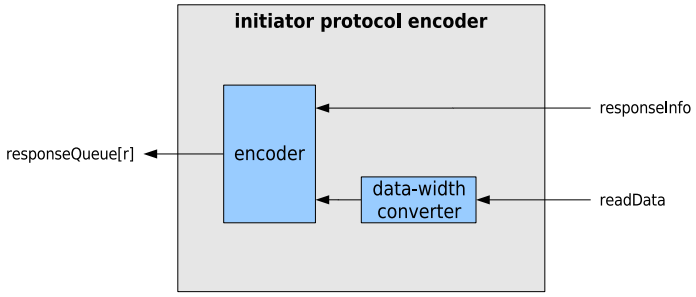


Figure 8.8: Initiator protocol encoder

A disadvantage of this method is that the buffers of the response delay block must be larger than strictly necessary. In the time that a request travels through the arbiter and resource, requests could also leave the response delay block, such that more space is available than reflected by the unreserved space.

No responses are delayed when composability is not required at design time. In this case, this component is not generated.

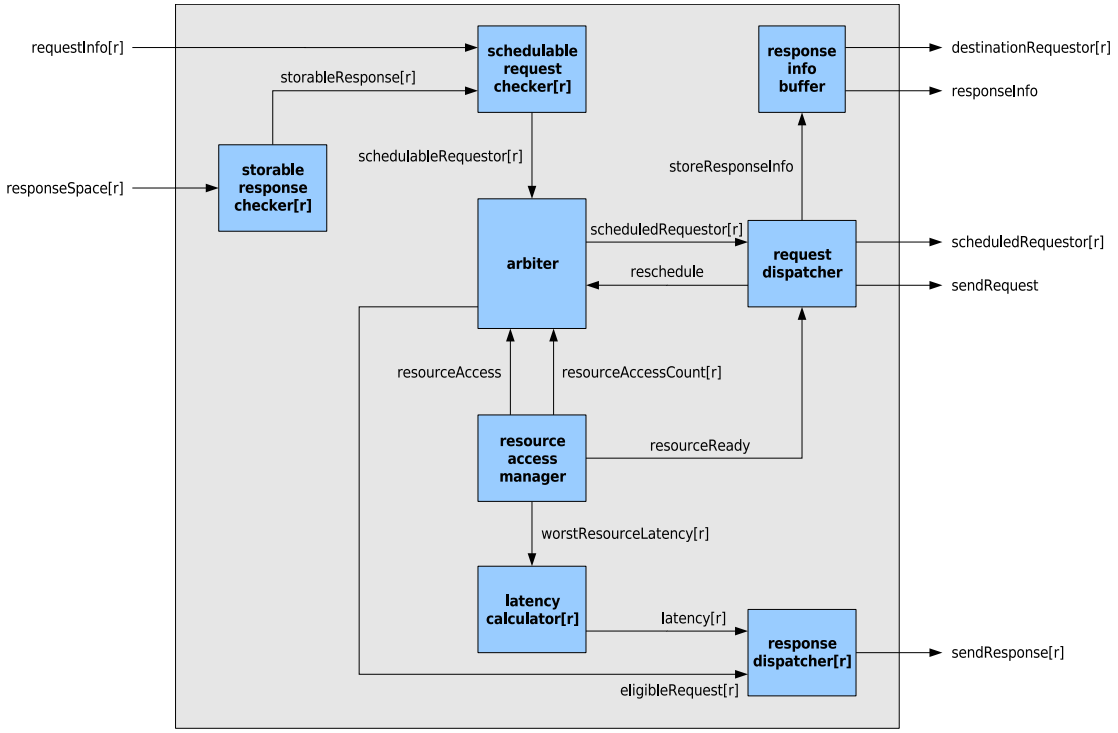


Figure 8.9: Block diagram of the controller

8.4.2 Schedulable request checker

A request is considered schedulable when it is guaranteed that all components in the path do not stall the request due to full buffers. For this purpose, the schedulable request checker verifies the buffers of the response delay block (using the storable response checker) and the response time and info buffers. Figure 7.3(a) shows these buffers. A request can only be scheduled when all buffers have sufficient space.

This mechanism assures that back-pressure does not stall the resource, but only the requestor. As soon as back-pressure stalls a request, the bounds on the timing behaviour for the corresponding requestor cannot be guaranteed anymore. However, the behaviour of other requestors is not affected, such that the front-end is still composable. The size of the buffers must be computed correctly to avoid that requests are stalled.

The current implementation lacks checking the response delay buffer when composability is disabled because the buffers are not instantiated. A misbehaving requestor could stall the resource in this case. The net bandwidth guarantees of other requestors may be violated.

8.4.3 CCSP Arbiter

The implementation of the Credit-Controller Static-Priority (CCSP) arbiter is based on the architecture proposed in [2] where also the formal proof of the behaviour can be found.

The CCSP arbiter consists of a rate regulator and a scheduler (Figure 7.3). The scheduler is a static priority scheduler that schedules the pending request with the highest priority is scheduled. Low latency can be guaranteed to high priority requestors because they do not have to wait for a lot of other requestors. The static priority scheduler is simply implemented by cascading multiplexers.

Soft real-time requestors are very bursty and often request more service than allocated. To prevent that low priority get no service at all because high priority requestors always have pending requests, a rate regulator is inserted before the scheduler. This regulates the provided service of a requestor according to the allocated burstiness and rate. The rate regulator is implemented by a credit mechanism. A requestor gets credits when it is not scheduled. When there are enough credits for a pending request, the request is considered to be eligible. Credits are decreased when the requestor is scheduled. The rate regulator is also relatively cheap to implement as it only requires integer arithmetic.

The credit mechanism and priority scheduler are illustrated by a simplified architecture in Figure 8.10. A more precise functional description is given in Appendix D. The CCSP arbiter can be work-conserving or non-work-conserving. The memory is not fully utilized by a non-work-conserving arbiter when the sum of the allocated service is less than the memory can deliver, or requestors do not use all their allocated service. A work-conserving arbiter also schedules pending requests at times that no requestor has enough credits. A work-conserving arbiter violates composability since it uses slack time. Both versions are implemented, but the work-conserving arbiter is not discussed here because it is not yet compatible with the test bench for the simulations. It can be used when $a_{sched[r]}(j)$ is defined for a request that is scheduled but does not have enough credits. The arbiter is non-preemptive, such that all memory accesses of a single request are scheduled successively.

Time and data of the CCSP arbiter are expressed in service units and service cycles. Both must be controllable by external component, as explained in Section 7.2.3 (page 69). For this purpose, the interface of the CCSP arbiter consists of the following signals:

- **(in)** *configuration[r]*: The configuration contains the allocated rate, priority and initial credits of each requestor. The configuration parameters are expressed in memory accesses (time and data).
- **(in)** *schedulableRequestor[r]*: Bit vector that represents the requestors that are allowed to be scheduled. This defines the arrival times of requests at the rate regulator ($a_{arb[r]}(i)$ when bit r is high for the first time). The schedulable request checker determines which requestors have a schedulable request.
- **(in)** *resourceAccessCount[r]*: Size of the pending request for each requestor. The size is expressed in the number of memory accesses since this is the service unit. The resource access manager, discussed in 8.4.6, calculates the number of memory accesses.
- **(in)** *resourceAccess*: When this signal is asserted, all the operations that have to be done for a service cycle are executed. For the CCSP arbiter, it is sufficient to update the credits of every requestor. The controller determines the duration of a service cycle by asserting *resourceAccess*. The resource access manager is

responsible for driving this signal. It is equal to the execution time of the memory access for that service cycle. More details are discussed in Section 8.4.6.

- **(in)** *reschedule*: A new requestor is scheduled when this signal is asserted. Essentially, the CCSP arbiter knows that it can schedule the next request after the execution time of all memory accesses, but the external signal allows that this decision can be made by an external component. The request dispatcher, discussed in Section 8.4.4, drives this signal and explains the purpose. This signal must be synchronized with *resourceAccess* to conform with the mapping to memory accesses. This signal corresponds to the time that a request is scheduled ($a_{bei}(i)$ when bit is high for the first time).
- **(out)** *eligibleRequestor[r]*: Bit vector that represents the requestors that are eligible. A requestor is eligible when the previous request of the requestor has finished, the requestor has a schedulable request and enough credits. This corresponds to the arrival time of a request at the priority scheduler ($a_{sched[r]}(j)$ when bit r is high for the first time).
- **(out)** *scheduledRequestor[r]*: Bit vector that indicates the current scheduled requestor (if any). It is updated when *reschedule* is asserted.

The CCSP arbiter is not aware of the data unit of a request nor the time to serve a request. The description of the interface shows that the service unit and cycle are fully controlled by external signals. This has the advantage that the implementation of the CCSP arbiter also can be used for other resources with different service unit and cycles. The arbiter still behaves according to Definitions 7.11 and 7.12 (page 70) but on the memory access time and data unit. Furthermore, this interface is not only suitable for the CCSP arbiter. Other schedulers like TDM can also be used. Using the *resourceAccess* and *reschedule* signals, a virtual time slot of TDM could be mapped to real time similarly. When *resourceAccess* is asserted, the TDM scheduler moves to the next time slot.

8.4.4 Request dispatcher

After a request has been scheduled by the CCSP arbiter, the request dispatcher performs some bookkeeping and assures that the request of the scheduled requestor is sent to the back-end. Request dispatching consists of the following tasks:

- *Controlling multiplexer*: The right requestor has to be selected, such that the target protocol encoder receives the right request (Figure 8.1).
- *Notify target protocol encoder*: The target protocol encoder can send the request to the back-end (*sendRequest*).
- *Notify response info buffer*: The response info buffer has to be notified that it can store information from the scheduled request.
- *Notify storable response checker*: The storable response checker is notified that buffer space can be reserved for the response of the scheduled request.

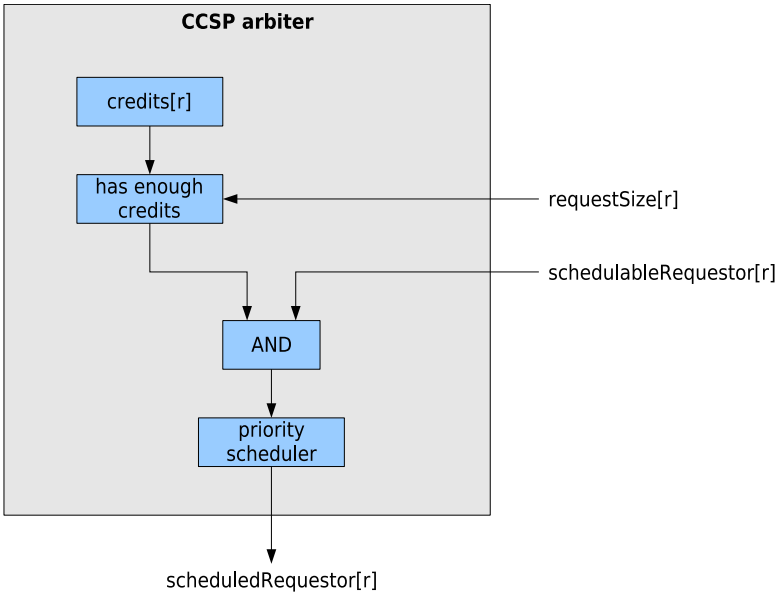


Figure 8.10: Simplified block diagram of CCSP arbiter

- *Reschedule*: When the resource access manager and target protocol decoder indicate that the request has been sent, the CCSP arbiter is instructed to schedule the next request. When the resource is stalled, it could be the case that the resource access manager thinks that the request is executed, but target protocol decoder did not sent its last word to the back-end yet. Therefore, the CCSP arbiter does not determine the moment to schedule the next requestor by itself. However, this can only happen when the target protocol decoder or back-end does not behave as expected (i.e. due to back-pressure).

8.4.5 Response info buffer

The response info buffer stores information derived from the scheduled request. In the first place, this information is intended for the target protocol encoder, such that the response can be constructed. In the second place, it stores which requestor was scheduled (*scheduledRequestor*). This is used to control the demultiplexer of the arbiter (Figure 8.1) and assures that the response can be sent to the right response delay block. The depth of this buffer depends on the pipelining of requests in the back-end interface. Table 9.9 shows that the response info buffer needs two elements for the use case. The schedulable request checker is notified when this buffer is full. Section 8.5.2 shows that the response info buffer behaves according to Definition 7.13 (page 70).

8.4.6 Resource access manager

To be independent of a particular resource, the CCSP arbiter is not aware of the latency and rate of the resource but operates with service units and cycles. The resource access

manager controls the CCSP arbiter in such a way that this abstract service is translated to memory accesses in terms of time and data. Because the arbiter block is not aware of the actual resource, memory accesses are generalized to resource accesses. A block diagram is shown in Figure 8.11.

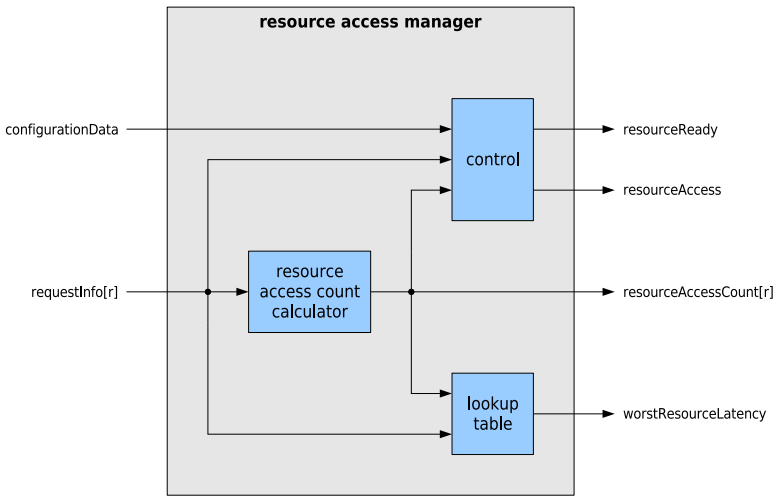


Figure 8.11: Block diagram of resource access manager

Based on the resource, the back-end has a certain access granularity. The logical address and size of a request determine the location and number of resource accesses. The resource access count calculator computes the number of resource accesses. Due to an unaligned address or the size of a request, more resource accesses are needed to read or write the data (Figure 6.2, page 34). Equation (8.2) shows how this computation is implemented.

$$
\begin{aligned}
start &= \left\lfloor \frac{address(i)}{width_{ra}} \right\rfloor \\
end &= \left\lfloor \frac{address(i) + size_{tgt}(i) \cdot width_{tgt} - 1}{width_{ra}} \right\rfloor \\
size_{ra}(i) &= end - start + 1
\end{aligned}
$$

where:	$address(i)$	Address of first byte of request i
	$width_{ra}$	Size of a resource access (bytes)
	$width_{tgt}$	Size of a resource word (bytes)
	$size_{tgt}(i)$	Size of request i, number of resource words
	$start$	Index of first resource access
	end	Index of last resource access
	$size_{ra}(i)$	Size of request i, number of resource words, number of resource accesses

In terms of hardware, the divisions and floors can be done using shift operations.

Therefore, this computation consists of shifts, subtractions and additions, which are relatively cheap to implement. The number of resource accesses is used by this component and the CCSP arbiter.

The control block generates the *resourceReady* and *resourceAccess* signals. The *resourceAccess* signal determines the length of a service cycle of the CCSP arbiter. It is asserted at the start of every resource access. When all accesses have been executed for the scheduled request, the *resourceReady* signal is asserted to indicate that the resource is ready to serve a new request. This method guarantees that $a_{bei}(i+1) = a_{bei}(i)+\mathcal{E}_{ps}(i)$, such that the back-end interface does not stall the request and behaves according to Definition 7.9 (page 69).

The control block generates the signals by emulating the behaviour of the back-end according to the PAM and CAM. Only the length of the memory command patterns have to be known because the actual commands are not issued by the resource access manager. Like the pattern scheduler of the back-end (Section 7.2.1), the rules and pending resource access determine the pattern that has to be issued. Counters are used to determine when the pattern ends. The *resourceAccess* signal is asserted when the last pattern of the resource access finished. When all resource accesses of the request have finished, the *resourceReady* is asserted additionally. Figure 8.12 illustrates when both signals are asserted and how this defines the service cycle of the CCSP arbiter. The duration of a service cycle is not constant. A refresh pattern causes a much longer service cycle.

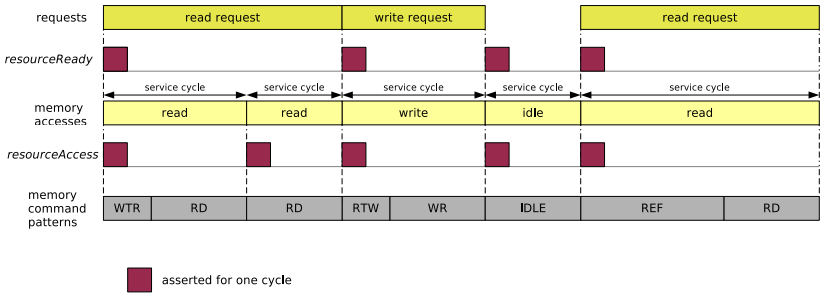


Figure 8.12: Timing behaviour of resource access manager

Emulation avoids that the the back-end interface needs to give run-time information of the execution time of the accesses. The handshake of the memory cannot be used for this purpose because it only indicates whether the request is stored in the internal buffers of the back-end. Eventually, the buffers get full and cause back-pressure. Hence, timing analysis is more difficult. The weakness of the emulation is that the back-end must behave exactly according to the PAM or CAM, such that the resource access manager is synchronized with the back-end. When the back-end behaves differently, the timing analysis may be wrong. In the worst case, a request is scheduled when the resource is not yet ready, resulting in a latency that could be higher than the analytical upper bound. However, back-pressure assures that no wrong data is read or written.

The lookup table returns the worst-case latency of the back-end interface for a particular request. This is used by the latency calculator for the total worst-case latency of a request, discussed in 8.4.7. The current implementation uses the worst-case latency for any request because the latency of a request could be affected by the preceding one

($\hat{\Theta}_{bei}$, Definition 7.9, page 69). This means that the latency is pessimistic in general, especially when there is a lot of variation in the size of the requests. According to Equation (6.18) and (6.19) (page 45), larger request have a higher worst-case latency than smaller ones. A more precise analysis of the back-end and back-end interface could lead lower latency guarantees for individual requests but makes analysis more difficult. This issue is discussed in Section 8.5. The single value in the lookup table is computed by Equation (6.18) and (6.19).

8.4.7 Latency calculator

The latency calculator determines the worst-case total latency of the pending requests ($\hat{\Theta}_{arb[r]}$, Definition 7.13, 70). This latency is used by the response dispatcher to delay responses. The latency calculator is only used when composability is required. The latency is built from three components:

1. *Scheduler latency*: The worst-case latency of the scheduler for the request ($\hat{\Theta}_{sched[r]}$, Definition 7.12). This latency is derived from the configuration register.
2. *Back-end interface latency*: The worst-case latency of the back-end interface for the request ($\hat{\Theta}_{bei}(i)$). This latency is derived from the lookup table in the resource access manager ($worstResourceLatency$).
3. *Response delay block*: The response delay block adds an additional cycle delay to the worst-case latency of a response, because the response is stored in a FIFO.

The total worst-case latency of the arbiter block (starting at $a_{sched[r]}(j)$) for a request is upper bounded by the sum of the three latencies:

$$\hat{\Theta}_{arb[r]} = \hat{\Theta}_{sched[r]} + \hat{\Theta}_{bei[r]} + t_{clk}$$

The sum of upper bounds results in a pessimistic upper bound for the total latency. Consider the use case of Section 9.2, where the maximum scheduler latency is 4030 ns and the latency of the back-end interface is 360 ns for some requestor. Both accounted for a refresh, but from the total latency (4435 ns) can be seen that two refreshed are never performed (the refresh period is 7600 ns). A less pessimistic number of refreshes can be determined when the number of refreshes is calculated for the total latency instead of for the individual components. However, when the worst-case latency of the back-end interface is not constant for all requests, this computation has to be implemented in hardware.

8.4.8 Response dispatcher

The purpose of the response dispatcher is to generate the *sendResponse* signal for the corresponding response delay block. As soon *sendResponse* is asserted, the response delay block is allowed to validate the response, such that it can leave the arbiter block. Figure 8.13 shows the architecture of the response dispatcher. At the time a request is eligible (a_{sched}), the finishing time of the response is pushed in the response time buffer (f_{rib}). This is computed by adding the latency from the latency calculator and the

current time. When the current time equals or is later than the computed finishing time in the buffer, *sendResponse* is asserted. When the response delay block indicates that the last word has been sent (*responseSent*), the time is popped from the buffer, such that the finishing time of the next response is visible.

The hardware implementation is slightly different than illustrated by the figure. An exact representation of the time is too expensive, because this requires too many bits (for the buffer, counter, adder and comparator). The number of bits for the time should be large enough to allow the worst-case latency to be represented. For example, when the maximum latency is 1024 cycles, the time must have at least 11 bits. Besides the current time, a finish bit is pushed into the response time buffer (a FIFO) that indicates whether the response must already be sent. The current time and finishing times are compared every cycle for all elements in the FIFO. When the current time is larger or equal to the finishing time, the finish bit is set. No action is needed if the bit already has been set. Finally, the response dispatcher asserts *sendResponse* when the finish bit at the head of the FIFO is set.

The response time buffer complies with Definition 7.13 when the behaviour is:

$$f_{rtb[r]}(j) = a_{sched[r]}(j) + \hat{\Theta}_{arb[r]}$$

This is satisfied when a response is not stalled because the previous response has not yet been removed from the response time buffer. The following properties guarantee that blocking does not occur:

- Delay of each response is constant: $\hat{\Theta}_{arb[r]}$
- Response $j+1$ arrives later than response j (actually $a_{sched[r]}(j+1) > a_{sched[r]}(j) + \mathcal{E}_{ps}(i)$, according Section 8.4.3).
- The execution time of the response delay block does not exceed the time between two successive finishing responses: $\mathcal{E}_{rsd[r]}(j) \leq \mathcal{E}_{ps}(i)$. The execution time of the response delay block is denoted by $\mathcal{E}_{rsd[r]}(j)$. Section 8.4.9 shows that this condition is satisfied.

Figure 8.15 illustrates the relation between the response dispatcher and response delay block. It shows that the response is delayed until the response time buffer notifies the response delay block to release the response by the *sendResponse* signal.

The depth of the response time buffer depends on the latency of the responses. Section 9 shows the usage of the buffer for a use case. The schedulable requestor checker is notified when the buffer is full such that back-pressure assures that no data is corrupted. Deadlines may be missed however.

When the front-end is configured for predictability only, *sendResponse* is always asserted such that the response is not delayed. All other hardware is not generated in this case.

8.4.9 Response delay block

The response delay block is responsible for delaying responses, such that the finishing time ($f_{arb[r]}(j)$) is not affected by other requestors. Figure 8.14 illustrates the basic

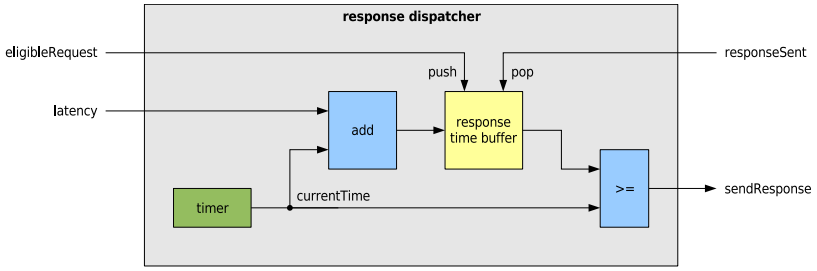


Figure 8.13: Block diagram of response dispatcher

structure of the response delay block. The response information and read data are only validated when the response is allowed to be sent, indicated by the *sendResponse* signal. The response dispatcher is responsible to assert this signal at the right time. In addition, the response delay block notifies the response dispatcher when the last word of the response has been sent, such that its timestamp can be removed from the response time buffer. Figure 8.15 illustrates the timing behaviour of the response delay block for a composable arbiter. Normally, the response is delayed for a much longer time, because the average latency is significantly smaller than the worst case. The figure shows that the execution time of the response delay block depends on the number of words a response consists of:

$$\mathcal{E}_{rsd}(i) = size_{tgt}(i) \cdot t_{clk}$$

This execution time does not exceed the time between two eligible requests, such that the response time buffer is not stalled. A request can only be eligible when the previous request is executed. The time to execute a request is $\mathcal{E}_{ps}(j)$ according to Section 8.4.6. The execution time of the pattern scheduler is equal or longer than the response delay block, because it needs at least that amount of cycles to send the data to the memory.

The behaviour of the response complies with Definition 7.16 (page 71), because a response is finished when the response dispatcher asserts *sendResponse*. In Section 8.4.8 it has been shown that this at $a_{sched[r]}(j) + \hat{\Theta}_{arb[r]}$. The response from the back-end interface is available at this time, since the actual arrival time is earlier than the worst-case by definition. When composability is not required, the response from the back-end interface is immediately forwarded to the requestor interface, as *sendResponse* is always asserted. If a part of the requestors do not require composability, a response delay block can be bypassed to improve average latency. However, this is currently not possible because response delay blocks cannot be configured independently.

The size of the buffers inside the response delay block depend on the worst-case latency of a requestor and the rate of responses. A requestor with a high worst-case latency and rate requires large buffers. Section 9.3.6 shows the maximum buffer filling for a use case.

8.5 Back-end interface

The behaviour of the back-end interface depends on the back-end. However, no hardware implementation for the back-end has been created. A SystemC model is made for

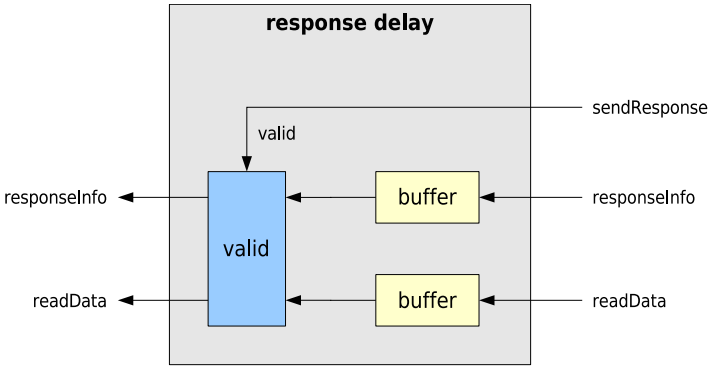


Figure 8.14: Response delay block

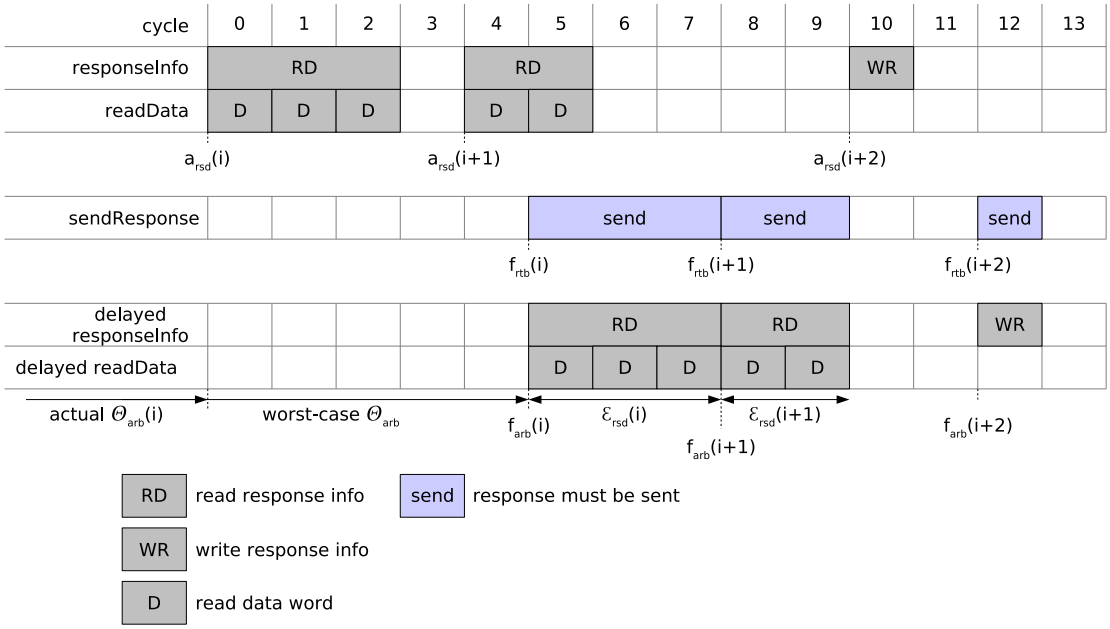


Figure 8.15: Timing behaviour of the response delay block

simulations that behaves according to Definition 7.5 on page 68. The back-end interface consists of the target protocol encoder and decoder. The main task is to send the requests in the right format to the back-end and create responses from the output of the back-end. These components are discussed in the next sections.

8.5.1 Target protocol encoder

The target protocol encoder sends the request that has been scheduled by the arbiter to the back-end. However, the format of the request from the arbiter has a slightly different format and needs to be converted. Two groups of signals are available for a request: commands and write data. The command group contains the address, access type and id for every word. The write data group consists of the write data itself and a

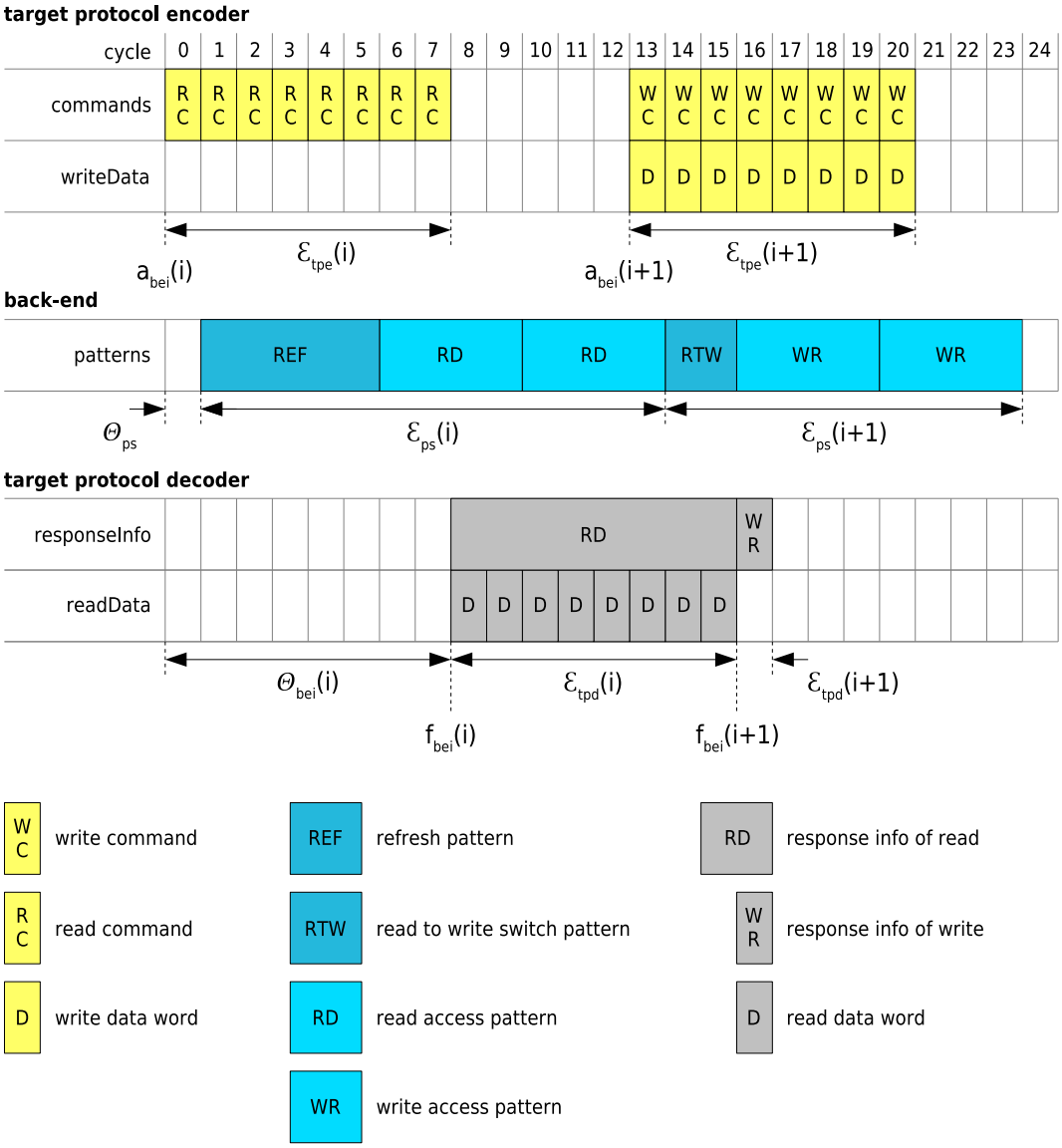


Figure 8.16: Timing diagram of the back-end interface

mask. Both groups have a handshake. A command needs to be sent for every word that needs to be accessed. The data of a write request is sent word by word along with the commands.

The back-end requires that at least the commands and data for one burst are sent, because this is the smallest amount of data that the memory can access. The target protocol encoder supports requests that are not aligned with a memory access as long as they are a multiple of the programmed burst size. The write data mask is used to write smaller amounts of data.

The back-end can only guarantee the timing behaviour of a request according to

Definition 7.5, when commands and write data are sent at the same time and without interruptions. Figure 8.16 illustrates how a read and write request of eight words are sent to the back-end. It also shows the relation with the memory command patterns. The arrival time of a request at the target protocol encoder (a_{bei}) is defined as the moment that the *requestInfo* group is available and write data can be delivered in an uninterrupted stream. The latency between the finishing (a_{be}) and arrival time (a_{bei}) of a request is zero. It is guaranteed that the target protocol encoder is not stalled because the arbiter delivers the request in an uninterrupted stream. The signals of the command group can be generated from the *requestInfo* group without any time overhead. The write data groups of the arbiter and back-end have the same format and data width such that there is no additional latency. The execution time of the target protocol encoder depends on the time to send all commands and data words to the back-end:

$$\mathcal{E}_{tpe}(i) = size_{tgt}(i) \cdot t_{clk}$$

The arbiter guarantees that the time between arriving requests is equal to $\mathcal{E}_{ps}(i)$. The execution time of the the target protocol encoder does not exceed $\mathcal{E}_{ps}(i)$, because the pattern scheduler needs at least $size_{tgt}(i)$ cycles to send all word to the memory. Therefore, this component does not stall requests and behaves according to Definition 7.7 (page 69).

8.5.2 Target protocol decoder

The purpose of the target protocol decoder is to construct a response from request information and read data from the back-end. To reduce dependencies with the resource, the target protocol decoder does not expect status information from the back-end. The back-end only returns the data of a read request.

A read response is created when the request at the head of the response info buffer of the arbiter is a read request and read data arrives. A read response consists of one *responseInfo* group and multiple read data words (*readData* group). A write response is created without waiting for data and therefore only consists of a *responseInfo* group. A write response must be created after is has been scheduled to prevent read after write hazards do not occur. The back-end can not cause this problem since requests are issued in-order. Figure 8.16 shows the relation between the start of a request by the target protocol encoder, the memory command patterns, and the responses created by the target protocol decoder.

After the response information and optionally read data have arrived, this component does not delay the response, because it only forwards the read data and response information. The rate depends on the number of words the response consists of (see Figure 8.16):

$$\mathcal{E}_{tpd}(i) = \begin{cases} size_{tgt}(i) \cdot t_{clk} & if\ type(i) = read \\ t_{clk} & if\ type(i) = write \end{cases}$$

According to Definition 7.8 (page 69), this component could stall the back-end. However, read data from the back-end is immediately forwarded to the arbiter, because the response information ($f_{rib}(i)$) arrives earlier than read data. The read data is also not

stalled when this component is still busy with the previous response, because the execution time never exceeds the time between the read data of two successive read requests.

In Figure 8.16, read data is delivered by the back-end without interruptions (empty cycles), such that arrival time of read data ($f_{be-rd}(i)$) is defined at the time the first word arrives. However, some command patterns could produce a stream with empty cycles (Figure 6.12, page 57). In this case, $f_{be-rd}(i)$ would be later than the first word.

The latency of the response info buffer depends on the time that the target protocol encoder sends the last word to the arbiter, because at that time, the response information is removed from the buffer. In the worst case, the previous response belongs to a read request that has a data latency of $\hat{\Theta}_{be}$. Figure 8.16 shows that the write response is delayed by the previous read response. Hence, the delay of the response info buffer never exceeds $\hat{\Theta}_{bei}$, such that it complies with Definition 7.14 on page 71. We assume that the worst-case latency of the memory is longer than one cycle.

Definition 7.9 requires that $\Theta_{bei}(i) \leq \hat{\Theta}_{be}$. The response info buffer and back-end are the only components that affect the finishing time of a response at the back-end interface ($f_{bei}(i)$). Definitions 7.14 and 7.5 show that this condition is satisfied.

8.6 Configuration

Some parts of the front-end can be configured at run time and some at design time. For run-time configuration the Device Transaction Level (DTL) protocol is used. The configuration registers can be read and written. The following subsections discusses the registers that must be programmed before the front-end can operate. Section 9.2.1 shows how the parameters are calculated for a use case.

8.6.0.1 Allocated rate

The allocated rate of each requestor is represented as a fraction of the total net bandwidth of the resource. The allocated rate is in in terms of resource accesses per amount of time. This fraction is configured by a denominator and numerator. Equation (8.2) shows the relation between the actual allocated rate and the denominator and numerator.

$$\rho'_{[r]} = \frac{numerator_{[r]}}{denominator_{[r]}} \tag{8.2}$$

The sum of all fraction should not exceed one, because the CCSP arbiter cannot guarantee the allocated service anymore [2].

8.6.0.2 Initial credits

The credit mechanism of the CCSP arbiter requires the initial credits for every requestor. The initial credits are calculated by Equation (8.3) and reflects the allocated burstiness of a requestor. Burstiness must be expressed in a number of resource accesses.

$$initialCredits_{[r]} = \sigma'_{[r]} \cdot denominator_{[r]} \tag{8.3}$$

8.6.0.3 Priority

When the priority switch of the CCSP arbiter has been enabled, the priorities of each requestor needs to be configured. Priorities are represented by 0 to $R-1$, where 0 and $R-1$ correspond to the highest and lowest priority respectively. The number of requestors is denoted by R. If the priority switch has not been enabled, the input port of the requestor determines the priority.

8.6.0.4 Latency

The latency register contains the worst-case latency of the scheduler of the CCSP in cycles ($\hat{\Theta}_{sched[r]}$). This latency is used by the latency calculator. Latency needs to be reconfigured when the allocated service or priorities are changed, because this affects the worst-case latency of the scheduler.

8.6.0.5 Design-time configuration

Composability can only be configured at design-time. This has the advantages that when composability is not required, chip area can be saved because some components are not used. Furthermore, data width of the back-end interface and requestor interfaces are configurable ($width_{tgt}$, $width_{ini}$ and $width_{queue}$). The lengths of the memory command patterns and data latency can be configured. These are used by the resource access manager, such that it is easy to switch to another SDRAM device. Finally, the sizes of buffers can be configured to support different use cases. The buffers must be sized according to the use case that needs the most space.

8.7 Conclusions

A hardware implementation of a front-end for a memory controller has been proposed in this section. It has been shown that the implementation corresponds to the design of chapter 7. This proves that the front-end is predictable and also supports composability.

The serialized AXI protocol used by the NoC causes that the network is slower than the memory when running at the same clock frequency. In the first place, this affects the requested service. However, this is not affecting the predictability and composability of the front-end and is the concern of the interconnect. For requestors with a high allocated rate, the network is unable to accept responses fast enough and bounds on rate and latency cannot be guaranteed anymore. We recommend one or more of the following solutions:

1. The NoC runs at an higher speed such that it can accept responses fast enough
2. Write mask is sent in parallel with the data (partial solution)
3. The data width of the NoC is larger than the resource (partial solution)
4. Requestors do not allocate too much service

For the composable front-end, a reservation mechanism is implemented in the arbiter to prevent that back-pressure stalls the memory. This is vital for a predictable and composable front-end, because the behaviour of other requestors is affected when the memory is stalled. For the predictable only front-end, there is currently no protection. A misbehaving requestor may stall the resource and cause violations of bandwidth and latency guarantees. A simple solution is to enable the response delay block.

The controller contains the CCSP arbiter and resource access manager among others. The resource access manager translates the resource independent data and time unit of the CCSP arbiter, to separate concerns. This modular design allows that a wide range of resources and schedulers can be used, simply by replacing the scheduler or resource access manager.

The worst-case latency of requests can be reduced when the worst-case latency of the back-end interface is allowed to depend on the request size and type. However, the analysis of the timing behaviour of responses is more complicated.

9 Experiments

The hardware implementation discussed in Section 8 has been simulated to verify its behaviour. In addition, the hardware description has been synthesized to obtain performance and area information. This section discusses the results of these experiments. The environment of the front-end is explained in Section 9.1. Section 9.2 introduces a video application that is used by the experiments. Simulation results are discussed in Section 9.3. The results of synthesis are presented in Section 9.4.

9.1 Test bench

A test bench has been made to verify and simulate the front-end. The test bench is implemented in SystemC to reduce the development effort. A SystemC shell has been created for the front-end to allow a mixed SystemC/VHDL simulation. The structure of the test bench is shown by Figure 9.1. The boxes outside the test bench represent files that are used for input and output. The back-end provides an interface based on memory accesses and executes the patterns according to the PAM or CAM. The SDRAM commands of the patterns are sent to the memory. Only read and write commands are executed by the memory to simplify the model. The memory assumes that the SDRAM commands do not violate timing constraints and the appropriate rows are activated and precharged. Traffic generators and response loggers emulate the behaviour of the NoC or any other interconnect. A traffic generator creates requests according to the traffic description. The requests are stored in the request queue of the traffic generator, such that the target protocol decoders can read the requests. Unfortunately, the traffic generator is not able to generate traffic according to a predefined rate and burstiness. Therefore, real behaviour of requestors cannot be emulated. The traffic generator fills the queue in an infinite rate such that there are always waiting requests. The response queue of the interconnect is implemented by the response loggers and filled by the front-end. The responses are stored in log files for verification. Incoming and outgoing traffic is inspected by the monitor. It verifies that the responses leave the front-end at the right time and that they contain the right header. Data is not verified because that would require a shadow memory. In addition, bandwidth and latency are measured and stored in files. Before the front-end can be used, it must be configured as explained in Section 8.6. The configuration master reads the configuration from a file and uses the DTL port to configure the front-end.

9.2 Use case

For the experiments, a use case is derived from the system design case in [25]. This is a video processing application that performs different operations on a video stream. The

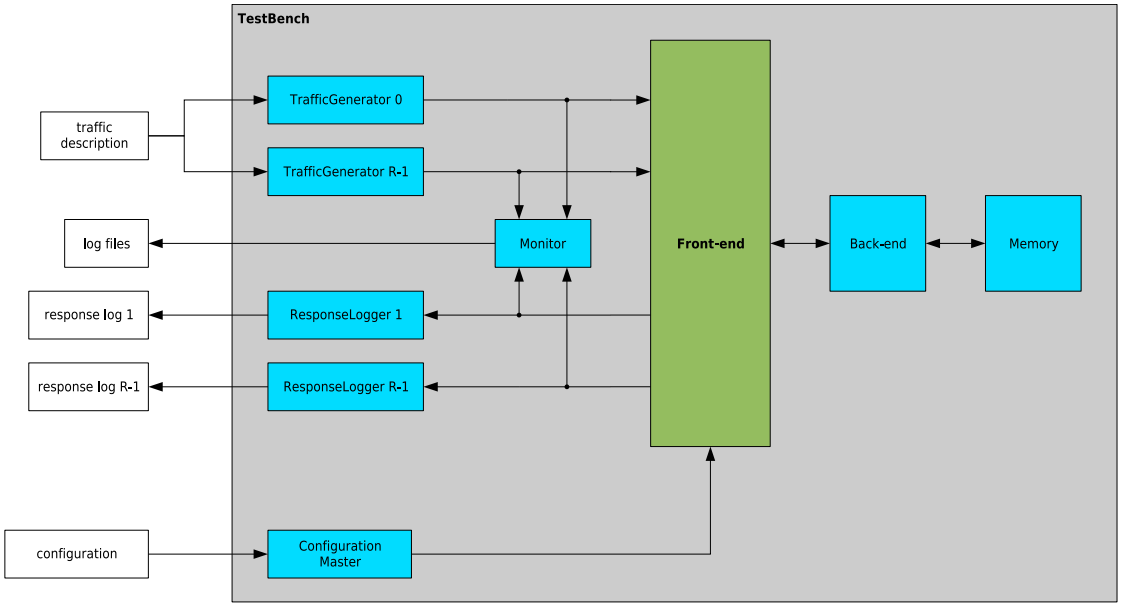


Figure 9.1: Test bench

application is mapped to a SoC with four processors and a memory system, illustrated by Figure 9.2. All processors communicate by a shared external memory. The input processor retrieves the input data and stores it in the memory. The input data is decoded by the Trimedia processor. The format of the video stream is enhanced to the High Definition (HD) format by the video processor before it reaches the LCD controller. This controller produces the appropriate signals for the display.

Five requestors can be identified in this use case: IP_{out}, TM, VP_{out}, VP_{in} and LCD_{in} as depicted by Figure 9.2. Except the Trimedia, all have uni-directional data streams. All memory requests are aligned and have a size of 128 bytes to allow efficient use of the memory. The processors have real-time requirements to guarantee that the application behaves properly. Input data is lost when the input processor fails to write the data in the same rate to the memory. When the LCD processor cannot get enough data from the memory, the displayed video is corrupted. The Trimedia and video processor also have requirements to guarantee that the execution time of their tasks are bounded. Table 9.1 lists the requirements of all requestors. The Trimedia needs the highest priority for low latency. In general, such processors benefit from a low average latency because execution is stalled when data has to be stored or loaded from memory.

The exact traffic of the requestor cannot be simulated because of the limitation of the traffic generator of the test bench. This means that all requestors request as much service as possible.

The memory of the test bench is configured for a DDR2-400 memory with an 16 bits data bus to satisfy the service requirements. The gross bandwidth of the memory is 800 MB/s. The memory controller back-end is configured to use a PAM or CAM for the mapping from memory accesses to SDRAM commands. The memory access granularity is 64 bytes. Details of the memory and the patterns that are used by the mappings are

Table 9.1: Service requirements of the use case

requestor	*burstiness (bytes)*	*net bandwidth (MB/s)*	*priority*
TM	384	220.00	0
VP_{out}	128	184.31	1
VP_{in}	128	62.21	2
IP_{out}	128	1.00	3
LCD_{in}	128	191.99	4

Table 9.2: Memory and back-end configuration

Memory device	DDR2-400, 256Mbit, 16 bit data bus
t_{clk}	5 ns
$width_{tgt}$	32 bit
$burstLength$	8
$interleavedBankCount$	4 banks per memory access
$burstCount$	1 burst per bank
$width_{ra}$	64 bytes
t_{access}	16 cycles
t_{ref}	26 cycles
$t_{refPeriod}$	1560 cycles
t_{rtw}	2 cycles
t_{wtr}	4 cycles
Θ_{ps}	4 cycles

listed in Table 9.2. The actual memory command patterns can be found in Appendix B.

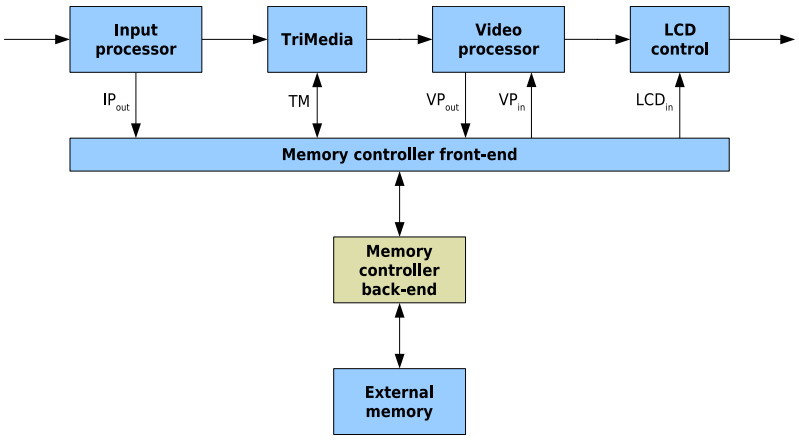


Figure 9.2: Architecture of the use case

Table 9.3: Service requirements expressed in resource accesses

requestor	$max\ size_{ra}$	σ'	ρ'
TM	2	6	0.332
VP_{out}	2	2	0.278
VP_{in}	2	2	0.0940
IP_{out}	2	2	0.00151
LCD_{in}	2	2	0.290

Table 9.4: Front-end configuration for the video application

requestor	*numerator*	*denominator*	*initial credits*	*scheduler latency*	*priority*
TM	170	511	3066	46	0
VP_{out}	142	510	1020	236	1
VP_{in}	48	510	1020	482	2
IP_{out}	1	511	1022	748	3
LCD_{in}	148	510	1020	806	4

9.2.1 Configuration

As mentioned in Section 8.6, the front-end must be configured before it can operate. The configuration for the front-end is derived from the service requirements of the requestors (Table 9.1). First, the requirements must be converted to resource access time and data units, shown in Table 9.3. The maximum request size for all requestors is two memory accesses. The allocated bandwidth is expressed as the fraction of the total net bandwidth. For this use case, the memory can guarantee 10,344,093 memory accesses per second (MA/s) for an interval of at least 188 us. This corresponds to 662 MB/s (Table 6.9).

The next step is to convert the real fraction to a discrete representation by a numerator and denominator. Since the discrete representation is less precise, more bandwidth must be allocated than really necessary. Nine bits are used for the numerator and denominator to guarantee that all requestors get their allocated bandwidth. This is the minimum amount of bits that are needed to assure that the total allocated bandwidth is below 100%. For nine bits precision, the total allocated bandwidth is 99.7%. Details about service allocation can be found in [3]. Such a heavy load is useful, because a memory controller is the bottleneck of embedded systems and hence heavily used. In addition, the system is stressed such that a violation of bounds on latency and bandwidth can easily be generated and observed.

Now the real allocated bandwidth is known, the initial credits can be computed by Equation (8.3) (page 99). The maximum latency of the priority scheduler in terms of memory accesses can be computed using Definition 7.12 on page 70. Equation (6.8) shows how to convert this latency to cycles. Table 9.4 shows the final configuration for this use case.

Experiments are performed on the front-end where composability is enabled or disabled. The front-end is predictable in both cases.

Table 9.5: Guaranteed maximum latency

requestor	$\hat{\Theta}_{sched[r]}$	*buffers*	$\hat{\Theta}_{bei}$		$\hat{\Theta}_{arb[r]}$	
			predictable	*composable*	*predictable*	*composable*
TM	230 ns	5 ns	360 ns	400 ns	595 ns	635 ns
VP_{out}	1180 ns	5 ns	360 ns	400 ns	1545 ns	1585 ns
VP_{in}	2410 ns	5 ns	360 ns	400 ns	2775 ns	2815 ns
IP_{out}	3740 ns	5 ns	360 ns	400 ns	4105 ns	4145 ns
LCD_{in}	4030 ns	5 ns	360 ns	400 ns	4395 ns	4435 ns

9.2.2 Latency

The configuration allows us to compute the latency bounds of the memory controller. Note that this includes the latency of back-end and memory. We do not account for the latency caused by the requestor interfaces because protocol conversion is the responsibility of the interconnect. In addition, the rate regulator of the CCSP arbiter introduces latency for requests that arrive too early. The latency of the rate regulator is high, because the traffic generates requests at a higher rate than they are served. Therefore, we do not consider this as part of the latency of the arbiter. The latency that is used in this section is the latency of the arbiter according to Definition 7.17 on page 71.

Section 9.2.1 already explained how to calculate the latency bound of the priority scheduler of the CCSP arbiter. According to Definitions 7.5 and 7.9 (page 68) the upper bound on the latency of the back-end interface is:

$$\hat{\Theta}_{bei} = \Theta_{ps} + \hat{\Theta}_{read-data}$$

The SystemC model of the back-end waits for a full burst before a pattern is issued. Therefore, $\Theta_{ps} = \frac{burstLength}{\rho_{memory}} = 20\ ns$. Formulas for the bound on read data latency can be found in Sections 6.4.1 and 6.4.2. Table 9.5 shows the guaranteed maximum latency for every requestor. Note that the buffers of the response delay block increases the worst-case latency by one cycle. The calculations for $\hat{\Theta}_{read-data}$ assume that a request could be unaligned. However, this results in a conservative latency of the back-end interface, since the requests of this use case are always aligned.

9.3 Simulation

In this section, the behaviour of the front-end is illustrated and explained by simulation results. The section starts with net bandwidth and latency. After these sections, the service of the CCSP arbiter is shown in an abstract world. Buffers play an important role for composability. Therefore, this section ends with showing the buffer usage during the simulation.

9.3.1 Average net bandwidth

Average net bandwidth is measured from the time that the first data is finished to the current simulation time. In all situations, the graph starts at the maximum bandwidth

(gross bandwidth) when the first data is finished and converges to the average net bandwidth of the end of the simulation.

The first simulation is performed using one requestor has been allocated 100% of the available net bandwidth. This allows verification of the behaviour of the back-end and memory in isolation, since the arbiter always schedules the only requestor. Figure 9.3 shows the average net bandwidth for a predictable and composable configuration. The memory and back-end of Table 9.2 is used, except that the width of the memory data bus has been reduced to 8 bits ($width_{tgt} = 16$ bits, $width_{ra} = 32$ bytes). This ensures that the requestor interfaces lower execution time for an memory access than the memory and do not stall requests. Table 6.9 shows that the 16 bit DDR2-400 device has an analytical minimum net bandwidth of 662 MB/s. Hence, the 8 bit memory can guarantee 331 MB/s. The curve for the composable front-end converges that bandwidth from above (Figure 9.3(b)). As motivated in Section 7.2.3, the composable front-end is not allowed to provide a higher bandwidth than allocated. However, when composability is not required, a requestor can get more bandwidth when there are less read/write conflicts. The traffic generates randomly 50% read and 50% write requests of 128 bytes. Statistically, there is a 25% chance for a read to write switch and 25% chance that a write to read switch pattern is necessary between two requests. Since bank efficiency, command efficiency and data efficiency are 100%, the analytical net bandwidth can be calculated by multiplying the read/write efficiency, refresh efficiency and gross bandwidth:

$$\begin{aligned} netBandwidth &= \eta_{rw} \cdot \eta_{refresh} \cdot grossBandwidth \\ &= 0.977 \cdot 0.983 \cdot 400\ MB/s \\ &= 384\ MB/s \end{aligned}$$

From the figure can be seen that the average net bandwidth of the predictable converges to 384 MB/s. The average bandwidth drops at multiples of 7800 ns, because this is the time that the memory is refreshed. The analytical read/write efficiency is not exactly equal to the actual efficiency, since requests are generated randomly. This is shown by Figure 9.3(b) where the net bandwidth is sometimes slightly below 384 MB/s.

The predictable memory controller enjoys the maximum net bandwidth when there are no read write switches at all. In this case, all efficiencies are 100% except refresh efficiency:

$$\begin{aligned} netBandwidth &= \eta_{rw} \cdot \eta_{refresh} \cdot grossBandwidth \\ &= 1 \cdot 0.983 \cdot 400\ MB/s \\ &= 393\ MB/s \end{aligned}$$

To illustrate the situation that the interconnect cannot deliver requests in a rate higher than the memory, the same use case with one requestor is used, except that a

16 bits memory is used and only write requests are generated. This memory has a net bandwidth of 662 MB/s, but the interconnect cannot handle that. From Equation (8.1) (page 82), we derive that a request arrives each 50 cycles. The maximum service that can be provided is $\frac{128\ bytes}{50\ cycles} = 512\ MB/s$ as can be seen from Figure 9.4.

Figures 9.5 and 9.6 show the average bandwidth for the video application. Recall that the traffic generator issues requests in an higher rate than is allocated. To guarantee that the memory controller can provide the allocated bandwidth, the rate of requests is regulated. All figures show that a requestor get at least its allocated bandwidth. Most curves have a sawtooth because the rate regulator has a periodic behaviour when there are always pending requests.

Figures 9.5(a) and 9.6(a) show the real use case. The allocated bandwidth depends on the total net bandwidth that is available at run-time. Since the worst case does not happen (i.e. read write switches are not always necessary), the requestors of the predictable front-end get more bandwidth. In contrast, every requestor of the composable front-end receives exactly the allocated bandwidth because the total net bandwidth at run-time is constant.

The purpose of composability can be noticed when the behaviour of some requestor changes. Figures 9.5(b) and 9.6(b) show the same use case as before, except that the LCD controller has been removed. Compared to the original predictable use case, the bandwidth of the remaining requestors is slightly higher. There are less read/write conflicts, because more idle accesses are issued. However, for the composable front-end, the bandwidth of the other requestors are not affected. This reduces verification effort, because it is not necessary to verify the real-time requirements of those requestors again.

9.3.2 Latency distribution

Figures 9.7 and 9.8 show how often requests have a specific latency. Figure 9.7 shows the situation for the predictable front-end. The frequency of high latencies are hardly visible, because they occur rarely. The majority of the requests have a much lower latency than the maximum latency. Furthermore, the requests are clustered to specific latencies. The latency is the sum of the scheduler and back-end interface latency. Both are often (around) a multiple of the execution time of an access pattern (80 ns), resulting in a discrete distribution. Table 9.6 lists the maximum latency that occurred during simulation. Note that the maximum latency for a requestor does not exceed the analytical worst case listed in Table 9.5. For all requestors with a low priority, the worst case situation is at the start of the simulation. At that time, every requestor has enough credits to be scheduled. Because there are always pending requests, a low priority requestor is only scheduled when the other requestors run out of credits.

Figure 9.8 shows the latency distribution for the composable front-end where the response delay block has been enabled. This figure shows that all requestors have a constant latency that cannot be affected by behaviour of others. The figure also shows that the latency is equal to the calculated worst-case latency of Table 9.5. The difference between the maximum latency of the predictable and composable front-end is around 200%. The reason is that the simulation does not trigger the worst case and the analytical worst case is too pessimistic as explained in Section 8.4.7. In addition, requests are always

Table 9.6: Maximum latency during simulation

requestor	$\hat{\Theta}_{arb[r]}$		*difference*
	predictable	*composable*	
TM	270 ns	635 ns	235%
VP_{out}	730 ns	1585 ns	217%
VP_{in}	1400 ns	2815 ns	201%
IP_{out}	1675 ns	4145 ns	247%
LCD_{in}	2740 ns	4435 ns	162%

Table 9.7: Average latency during simulation

requestor	*avg.* $\Theta_{arb[r]}$		*difference*
	predictable	*composable*	
TM	95 ns	635 ns	668%
VP_{out}	135 ns	1585 ns	1174%
VP_{in}	260 ns	2815 ns	1083%
IP_{out}	400 ns	4145 ns	1036%
LCD_{in}	430 ns	4435 ns	1031%

aligned.

The average latency of the requestors is listed in Table 9.7. The average latency for the composable front-end is equal to the worst-case latency since it is constant. The Trimedia processor has the highest priority and therefore has the lowest average latency. The average latency of the predictable front-end is an order of magnitude lower than the composable front-end.

9.3.3 Latency of subcomponents

The latency of the arbiter of the predictable front-end consists of the CCSP arbiter and back-end interface. As mentioned earlier, the latency of the requestor interface and rate regulator of the CCSP arbiter are not included. Figure 9.9(a) shows the average contribution of the subcomponents to the total latency. Resource refers to the back-end interface and scheduler to the CCSP arbiter. The latency of the scheduler increases for requestors with a lower priority, because there is more interference. The contribution of the resource is decreasing, as it does not depend on the requestor. Interference from other requestors has the highest impact on the scheduler latency of requestor 3 (IP_{out}). Its scheduler latency is high compared to the resource latency because it has a much lower bandwidth, such that there is more interference from requestors with a higher priority.

Figure 9.9(b) shows the results for the composable component. The response delay block is also included now. On average, it is responsible for around 85% of the total latency. This confirms that the response delay block is the main contribution to the latency of the composable front-end.

9.3.4 Abstract service

As explained in Section 8.4.3, the service unit and cycle of the CCSP arbiter can be controlled. The resource access manager assures that the service units and cycles are mapped to resource accesses (Section 8.4.6, page 90). Figure 9.10 shows the provided service curves for all requestors. The maximum service curve is derived from the allocated rate and burstiness. The minimum service is shifted by the guaranteed maximum latency. Service is measured at the output of the CCSP arbiter. The accumulated service is expressed in the amount of served resource accesses. Time is also expressed in resource accesses. All service curves start at the sixth resource access because the front-end is configured initially. Note that the real execution time of the resource accesses are not constant as shown in Section 9.3.5. On the real time domain, the minimum and maximum service bounds are not straight lines.

The provided service curve shows clearly when the requestor is scheduled (curve increases). If the requestor is not scheduled, the curve remains flat. All requestors are scheduled periodically according to the provided service. This is caused by the rate regulator of the CCSP arbiter which is always active since the requestors ask more service than is allocated.

The figure shows that the provided service remains within the bounds. This proves that all requestors get the allocated rate and the maximum scheduler latency is not exceeded.

On the data and time domain of resource accesses, the service of the predictable and composable front-end are identical (Figure 9.11). The CCSP arbiter is unaware of composability at the level of service units and cycles.

9.3.5 Mapping from memory access to real time domain

As explained earlier, the provided service of the composable and predictable front-end are identical in terms of resource accesses. The reason that the predictable front-end provides more net bandwidth is due to the mapping from resource accesses to real time. The resource access manager performs this mapping to control the CCSP arbiter. Figure 9.12(a) shows the difference between the mapping of the predictable and composable front-end for the original use case, and when a requestor has been removed. The execution time of a resource access at the predictable front-end is shorter, because read/write switches are not always necessary. Even less switch patterns are executed when a requestor is removed because idle accesses are executed instead of the read or write accesses. An idle access is never preceded by a switch pattern according to Table 6.4.

In contrast, the resource accesses of the composable front-end always include time to perform a switch, such that the time to perform an access is not traffic dependent (Section 6.4.2). The figure shows that the execution time of a resource access does not change when a requestor is removed (the composable curves of Figure 9.12(a) are identical). However, the rate is lower than the predictable front-end, because the average execution time of a resource access is longer.

To illustrate the difference of the execution time of a resource access, Figure 9.12(b) zooms in to the time that a refresh is performed. Around 7800 ns, there is one resource access that is much longer because it includes the refresh pattern. This figure also shows

Table 9.8: Maximum filling of buffers for composability

requestor	*response time buffer (responses)*	*response delay*	
		responseInfo (responses)	*readData (words)*
TM	4	4	64
VP_{out}	3	3	0
VP_{in}	2	2	64
IP_{out}	1	1	0
LCD_{in}	8	8	256

Table 9.9: Maximum filling of response info buffer

predictable	2 responses
composable	2 responses

that the curves are not straight lines, due to the variable resource access times. Note that the execution time of a memory access for the composable front-end is also not constant, because long and short patterns are interleaved (Table 6.6, page 47).

9.3.6 Buffer filling

All buffers are monitored during simulation. Table 9.9 shows the maximum filling of the response info buffer. Composability does not require a larger response info buffer. Table 9.8 shows the maximum filling of the buffers that are instantiated for the composable front-end. The request and response buffers in front of the arbiter are not listed in the table, because they are not considered to be part of the memory controller. The worst-case filling of the buffers can be determined analytically using the allocated service and latency of the front-end. However, this is outside the scope of this project. The read data buffer of the response delay block is the largest buffer. For write-only requestors, this buffer is not necessary because there is no read data to delay. The LCD controller needs the largest buffers. Due to the high provided bandwidth, responses enter the buffers in a high rate. In addition, responses remain in the buffers for a long time, because the LCD controller has the highest worst-case latency of all requestors.

9.4 Synthesis

The main purpose of the hardware implementation of the front-end is to check if the design can be mapped to a real hardware description and what kind of hardware is necessary. The hardware implementation is synthesized to get estimations on speed and area. Performance and scalability are verified by analysing the results. The synthesizer uses CMOS090LP technology.

The front-end is configured for the video application of Section 9.2. Only design-time configuration like the width of signals and buffer sizes affect the synthesis. The hardware does not depend on the amount of allocated service for example. The buffers between the requestor interfaces and arbiter and the buffers inside the response delay block have

a capacity of two elements to reduce the synthesis time. These buffers can be found in Figure 8.1. However, the speed estimation is too optimistic because buffers with the right dimensions are larger and slower.

Figure 9.13 shows the maximum clock frequency of the front-end for different number of requestors. Without the response delay buffers, composability has not a huge impact on the speed of the front-end. The front-end is not capable of running at 200 Mhz however (the speed of a DDR2-400 memory device). The maximum frequency decreases gradually when more requestors are supported as this requires more hardware. The design is not optimized for speed and does not utilize pipelining, except between the requestor interfaces and arbiter. The speed can be improved by applying more pipeline stages at the cost of some cycles latency. This is not a problem, since this is a very small fraction of the total latency.

The area of the front-end is discussed in two parts: the requestor interfaces and the arbiter and back-end interface as a whole. The reason is that the main task of the requestor interface (protocol encoding and decoding) is the responsibility of the interconnect. The arbiter and back-end interface belong to the memory controller. The buffers between the arbiter and back-end interface are not included.

9.4.1 Requestor interfaces

A requestor interface consists of an initiator protocol encoder and decoder. The area of all requestor interfaces is shown in Figure 9.14. Area scales linear with the amount of requestors because every requestor uses the same requestor interface. There could be more variation when requestors use different protocols. Furthermore, the design of the requestor interface is not changed when composability is required, because there are no dependencies between the requestor interfaces.

The buffers that are used to guarantee an uninterrupted stream are the largest contribution to the area of a requestor interface. The remaining part of the initiator protocol decoder is still larger than the encoder, because of the lookup tables and decoder as illustrated by Figure 9.15.

9.4.2 Arbiter and back-end interface

The hardware inside the arbiter depends on the number of requestors and whether composability is enabled. Figure 9.16 illustrates the area consumption of the arbiter and back-end interface. Both area curves show a linear trend, but composability requires more area per requestor. Components like the multiplexer and demultiplexer have $\mathcal{O}(R^2)$ complexity with respect to area. However, since the number of requestors (R) is too low, they do not have a significant effect on the total area.

Figure 9.17 shows the area consumption of the internal components. The contribution of the back-end interface decreases when the number of requestors increase, as only one instance is required. The controller consumes the most area and scales proportional with the number of requestors. The response time buffer is only instantiated for a composable controller. It has a depth of 12 elements to have enough room for the use case (Table 9.8). Likewise, the response info buffer has a capacity of three responses. Besides the

Table 9.10: Size of the response delay block

requestor	*responseInfo*		*readData*	
	responses	*bits*	*words*	*bits*
TM	8	168	192	6336
VP_{out}	7	147	0	0
VP_{in}	6	126	192	6336
IP_{out}	5	105	0	0
LCD_{in}	12	252	384	12672

buffers, the CCSP arbiter consumes the most area of the controller due registers and arithmetic units.

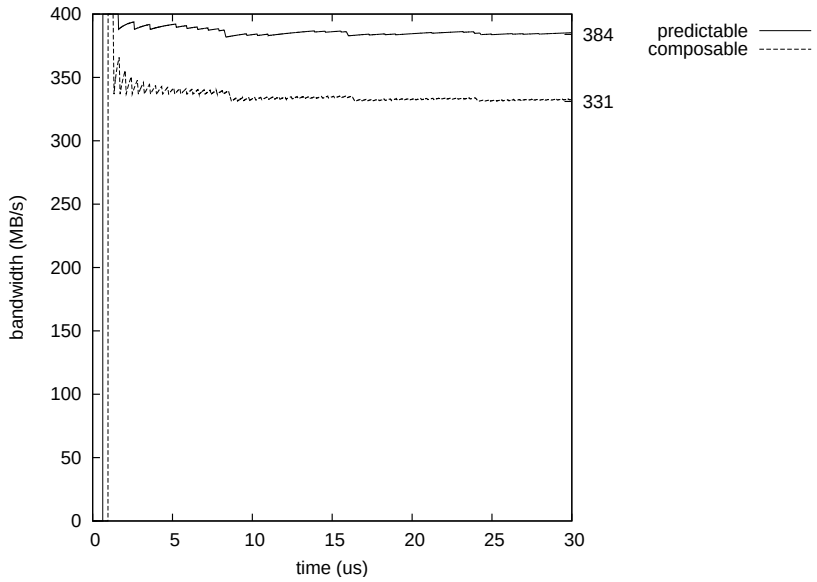
Until now, the size of the response delay block has not been discussed. Normally, it dominates the area of the controller and therefore hides the impact of other logic in the front-end. Recall from Section 8.4.9 that the response delay block consists of a buffer for the response information and one for the read data. Table 9.10 lists the minimum size of the buffers for the video application. Note that this is substantially more than the maximum filling listed in Table 9.8, because of the reservation mechanism of the storable response checker. The collective size of the buffers is 26142 bits. The response delay blocks are not synthesized, but the total area is estimated to be $0.76mm^2$. This is 89% of the area of the composable front-end when the response delay blocks are included..

9.5 Conclusions

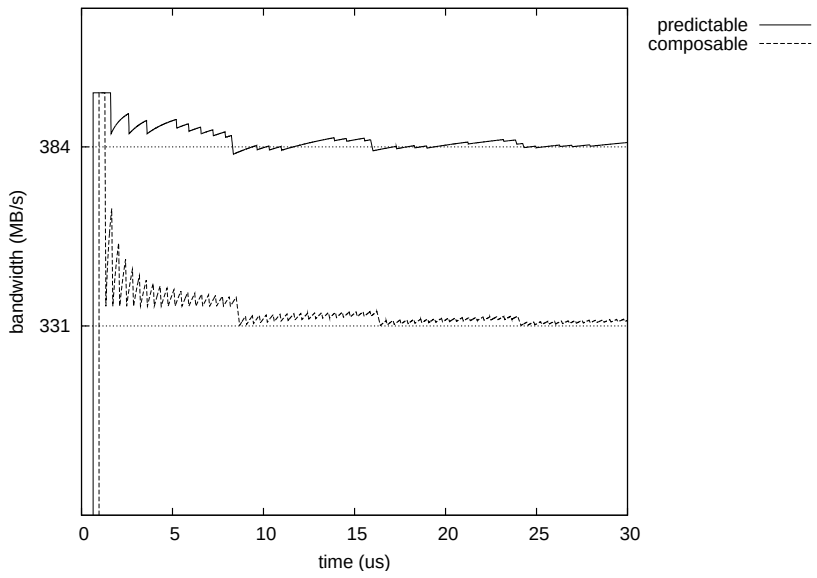
This section showed that the implementation of the front-end conforms to its requirements. Bounds on latency and bandwidth are not exceeded, even for a use case that uses 99.7% of the guaranteed net bandwidth. The latency bounds range from 595 ns for the Trimedia of the predictable front-end to 4435ns for the LCD controller when composability is enabled. Furthermore, abstract service curves show that the implemented CCSP arbiter behaves according to the formal model.

When composability is enabled, the latency or provided bandwidth of the front-end is not affected when a requestor is removed. Hence, integration of the memory controller in a composable system is easier, because it does not affect the behaviour of the requestors implicitly.

Synthesis shows that the hardware implementation is too slow for modern memories. However, the implementation is scalable in terms of requestors. For the use case with five requestors, the requestor interfaces consume $0.15mm^2$ of chip area, using CMOS090LP technology. The predictable arbiter and back-end interface require $0.048mm^2$. When composability is enabled, $0.093mm^2$ is required, but the response delay blocks are excluded. The buffers for the response delay are very large because it must have a capacity of 26k bits. The estimated area of all five response delay blocks is $0.76mm^2$.



(a) Full bandwidth range



(b) Analytical bounds on net bandwidth

Figure 9.3: Average net bandwidth for maximum requested service on a 16 bits memory

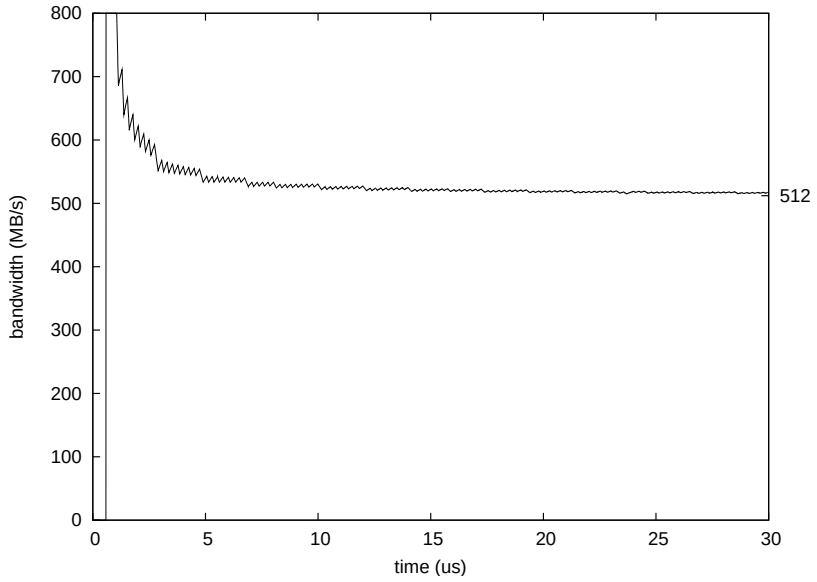
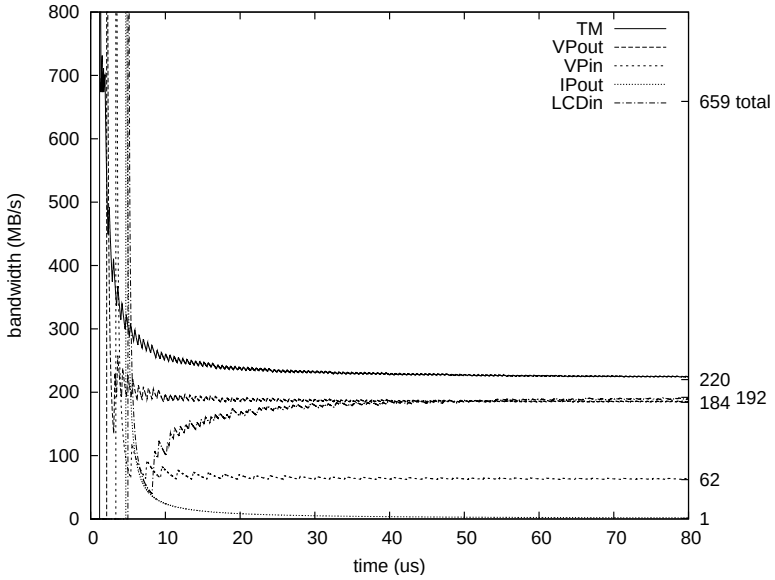
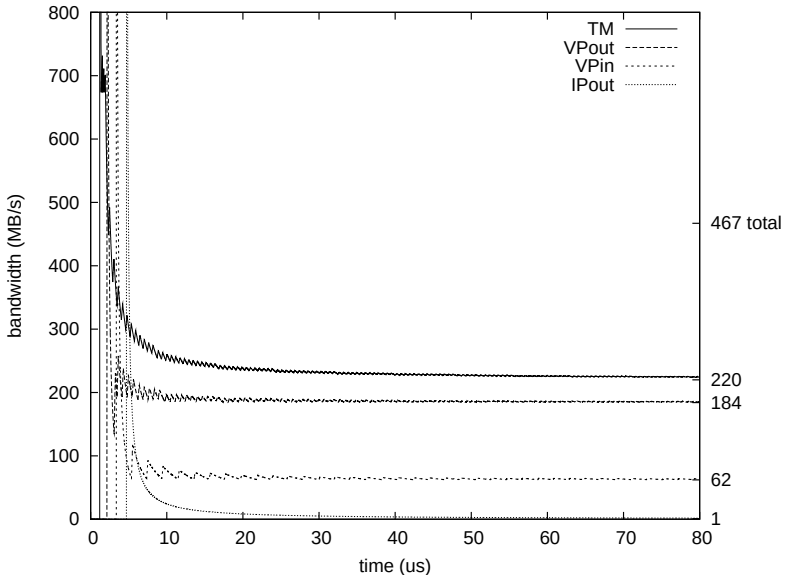


Figure 9.4: Average net bandwidth for maximum requested service on a 16 bits memory

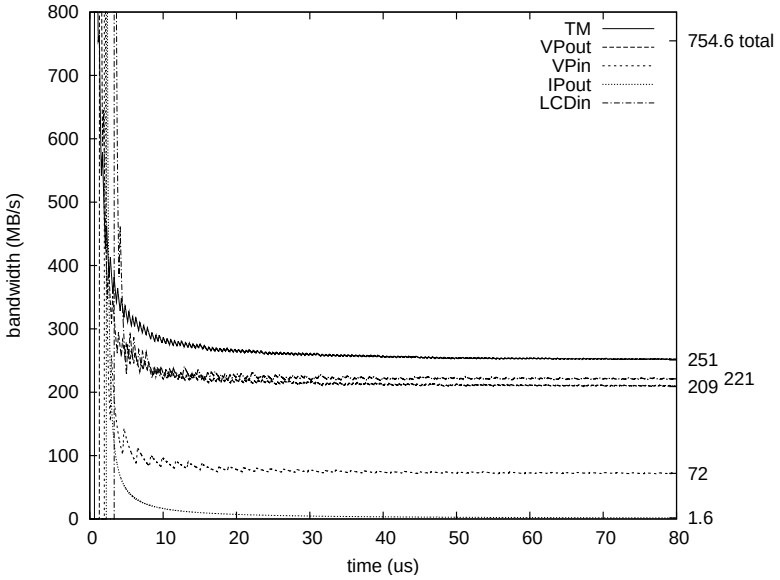


(a) Composable, 5 requestors

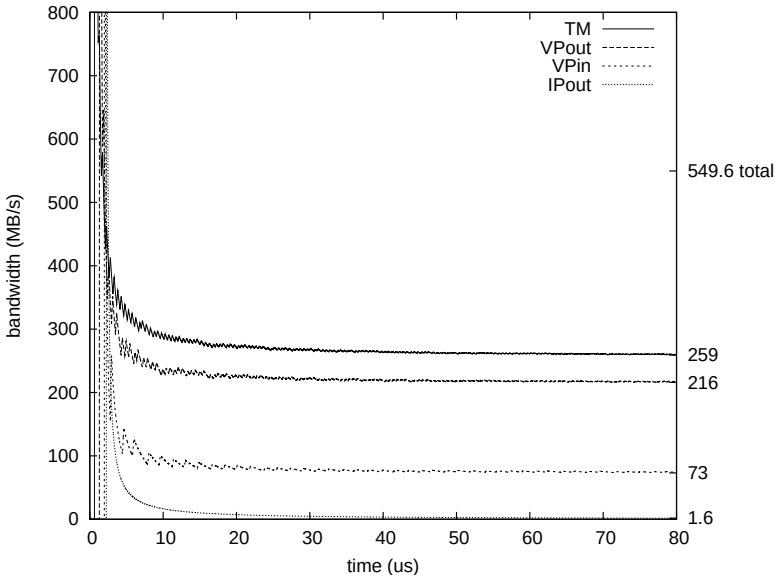


(b) Composable, 4 requestors

Figure 9.5: Average net bandwidth for the composable use case

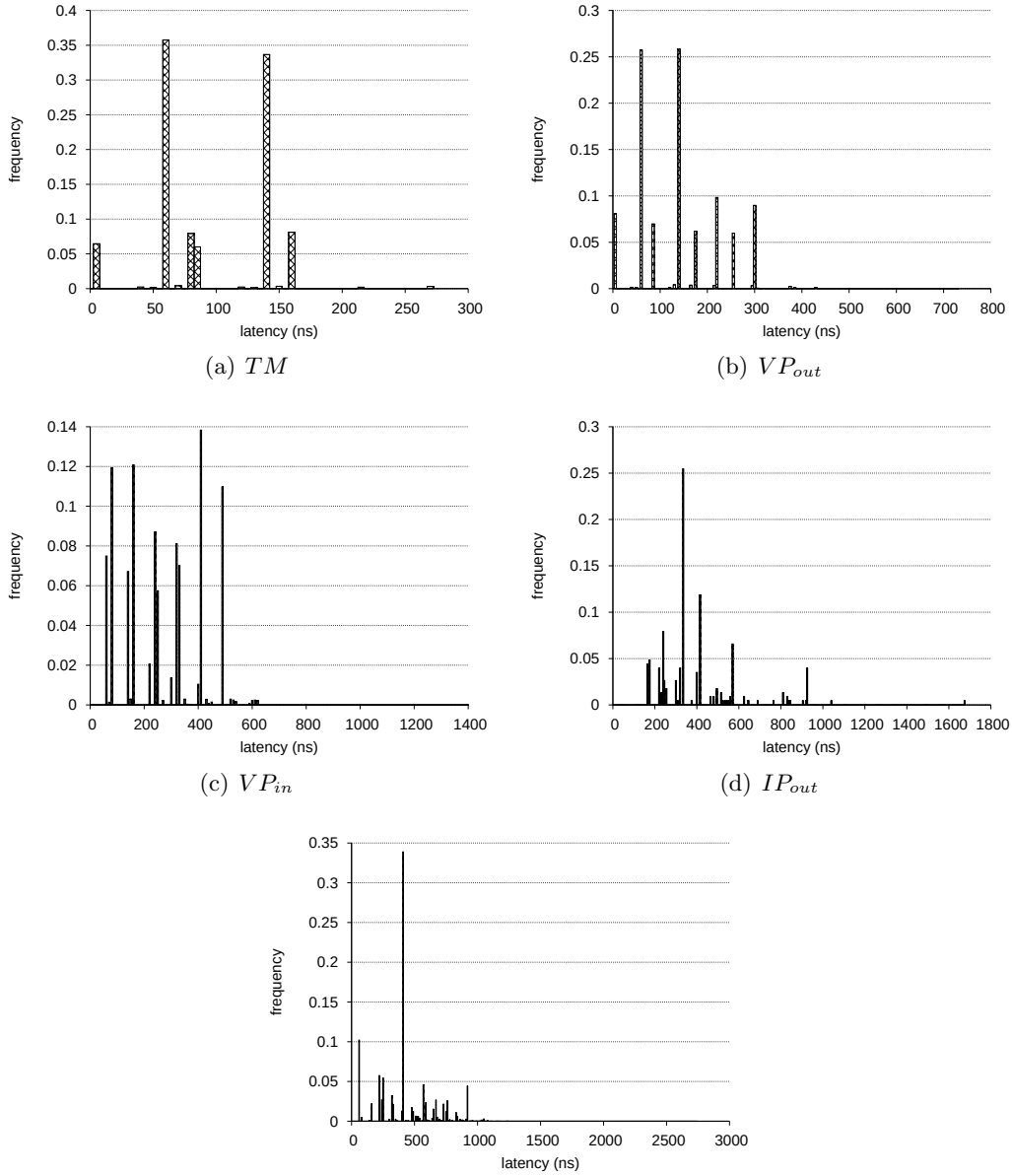


(a) Predictable, 5 requestors



(b) Predictable, 4 requestors

Figure 9.6: Average net bandwidth for the predictable use case



(a) TM

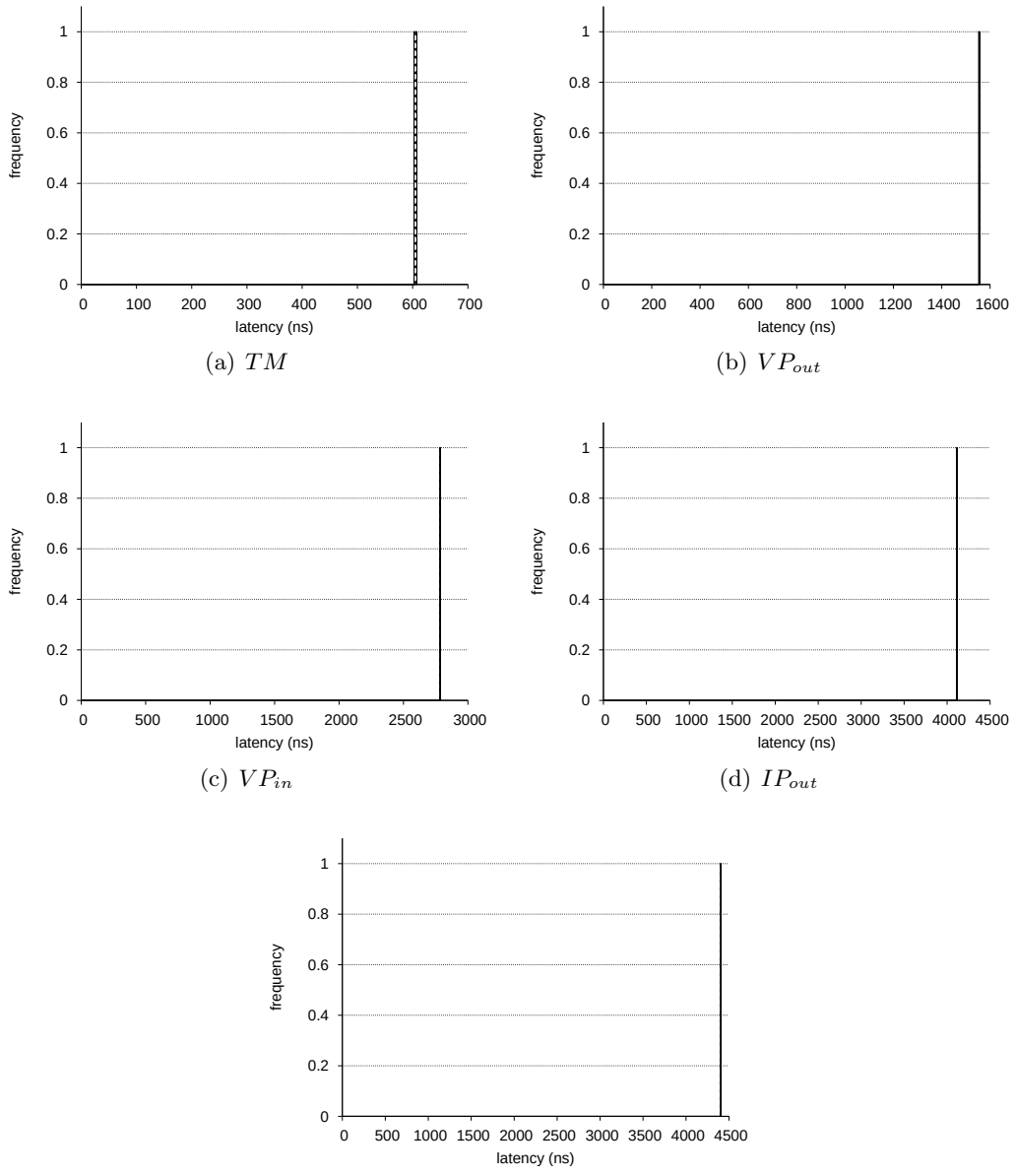
(b) VP_{out}

(c) VP_{in}

(d) IP_{out}

(e) LCD_{in}

Figure 9.7: Latency distribution of the predictable front-end



(a) TM

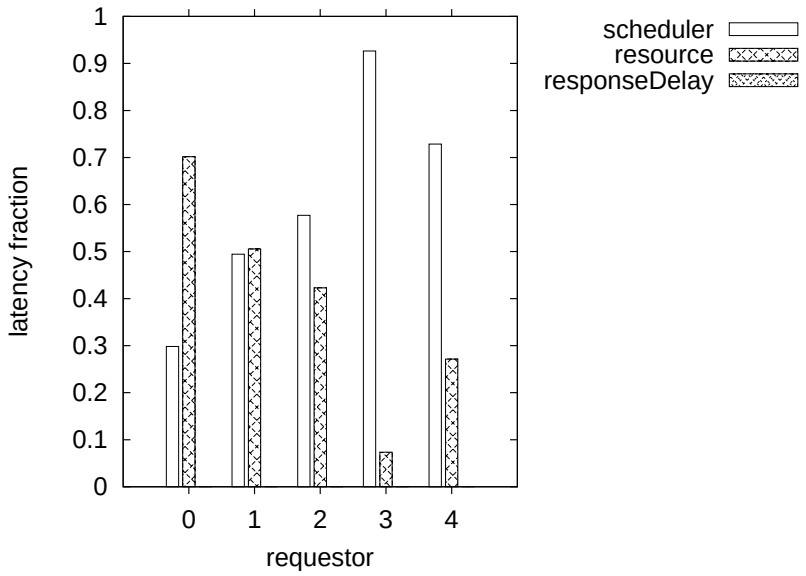
(b) VP_{out}

(c) VP_{in}

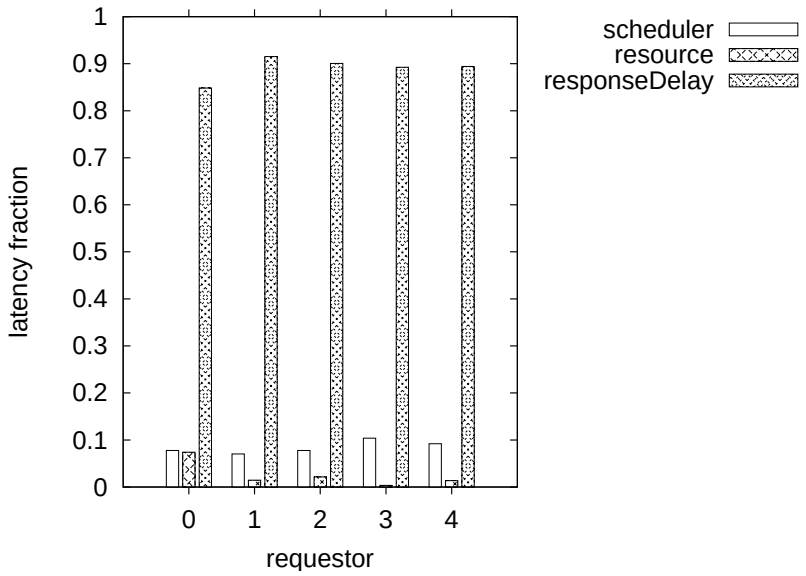
(d) IP_{out}

(e) LCD_{in}

Figure 9.8: Latency distribution of the composable front-end

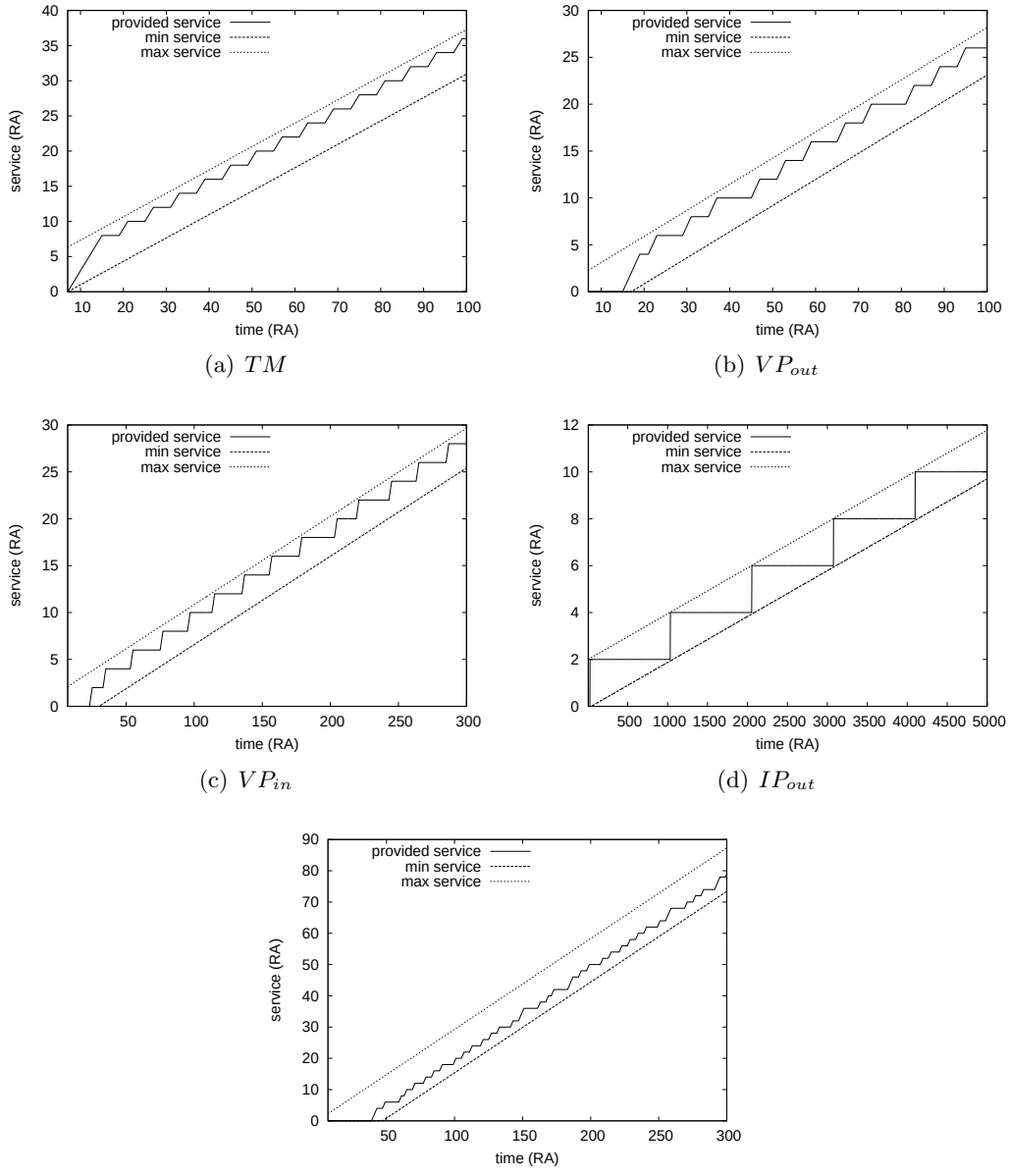


(a) Predictable front-end



(b) Composable front-end

Figure 9.9: Latency fraction of the subcomponents



(a) TM

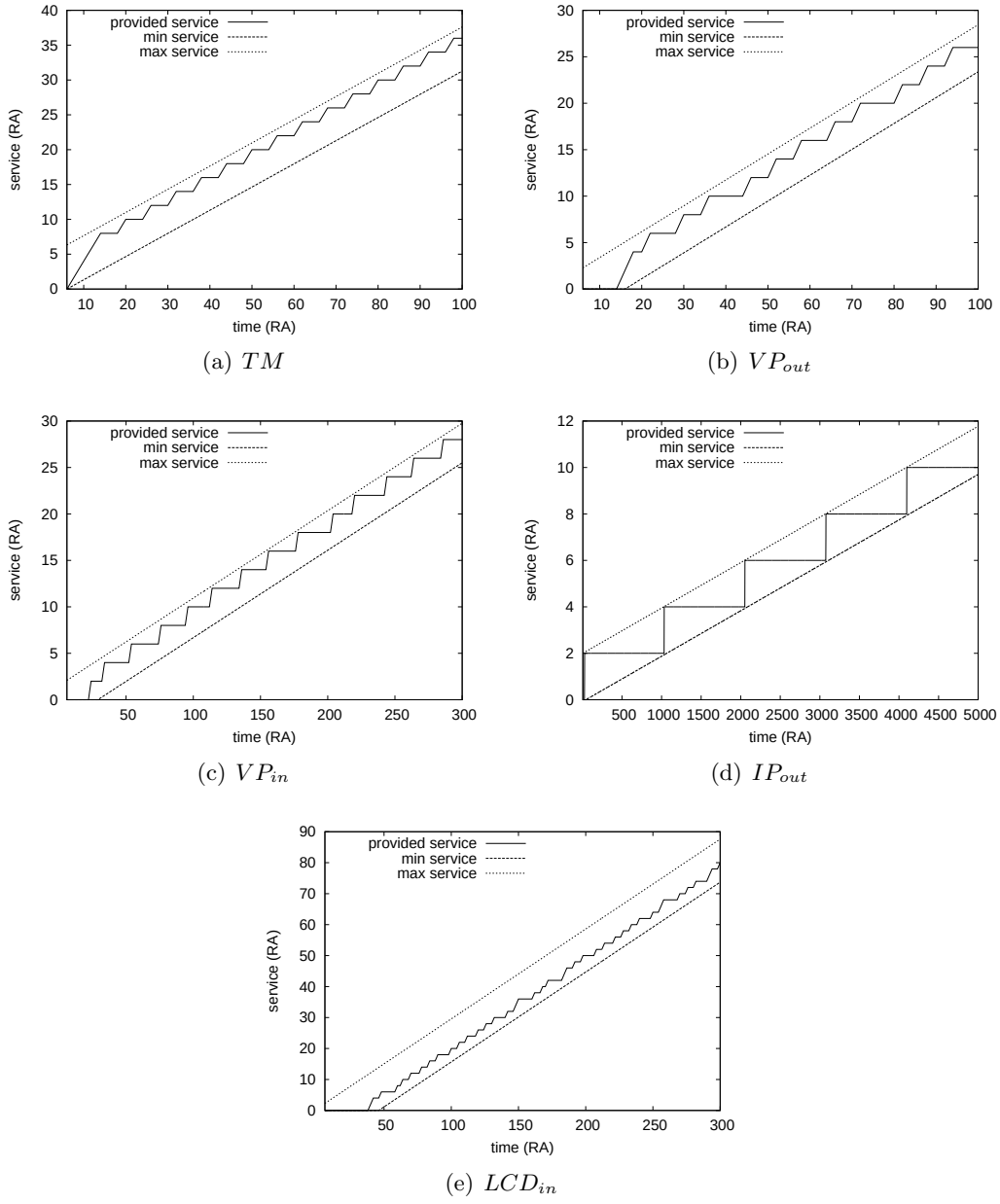
(b) VP_{out}

(c) VP_{in}

(d) IP_{out}

(e) LCD_{in}

Figure 9.10: Service of the predictable front-end



(a) TM

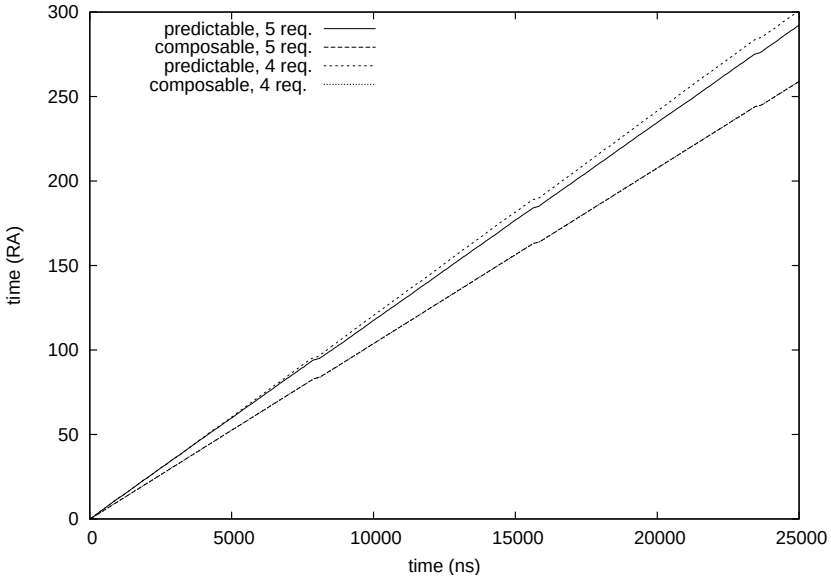
(b) VP_{out}

(c) VP_{in}

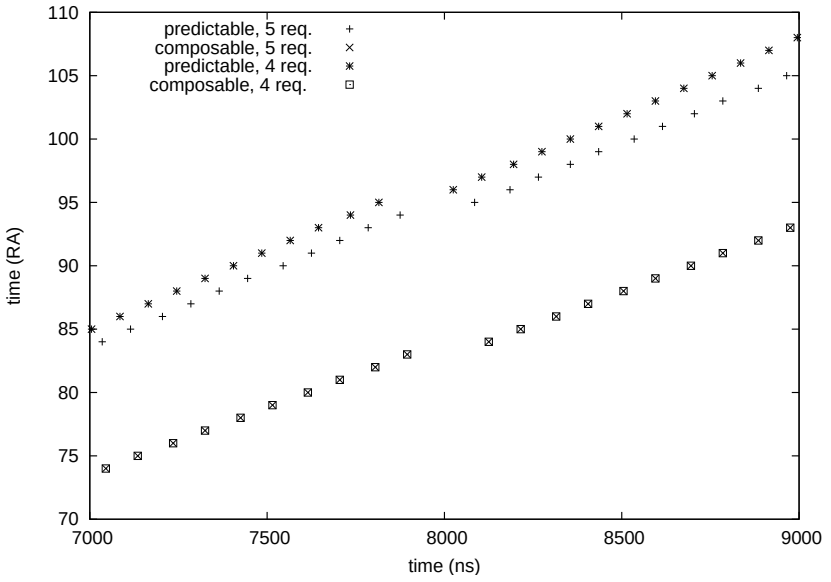
(d) IP_{out}

(e) LCD_{in}

Figure 9.11: Service of the composable front-end



(a) Mapping of predictable and composable front-end



(b) Zoomed in to the first memory refresh of 9.12(a)

Figure 9.12: Mapping of resource accesses to real time

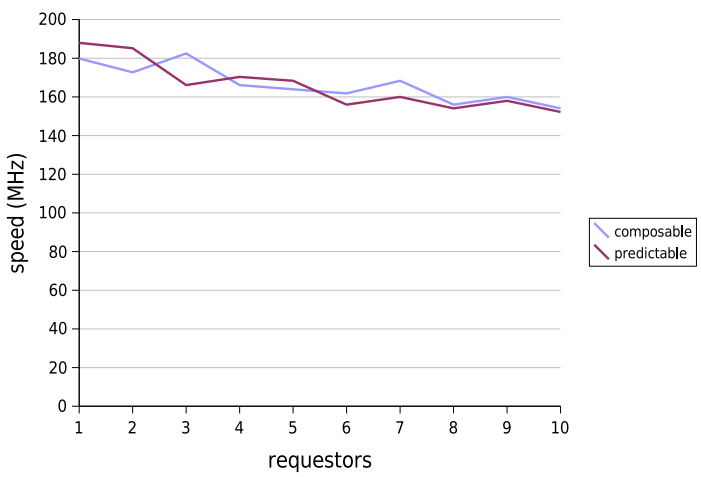


Figure 9.13: Maximum frequency for front-end

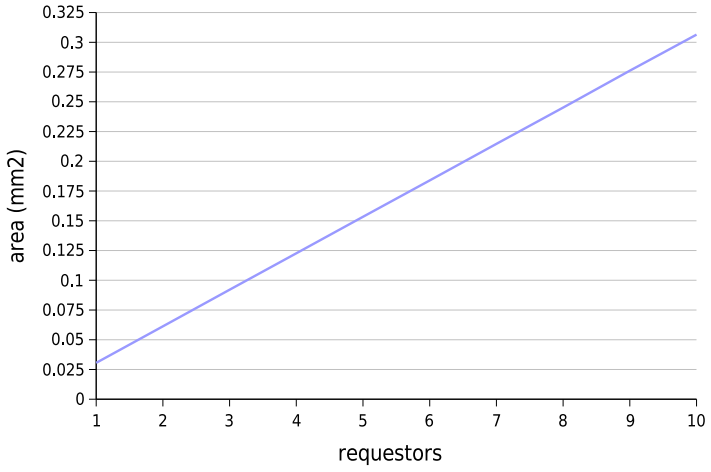


Figure 9.14: Area of all requestor interfaces

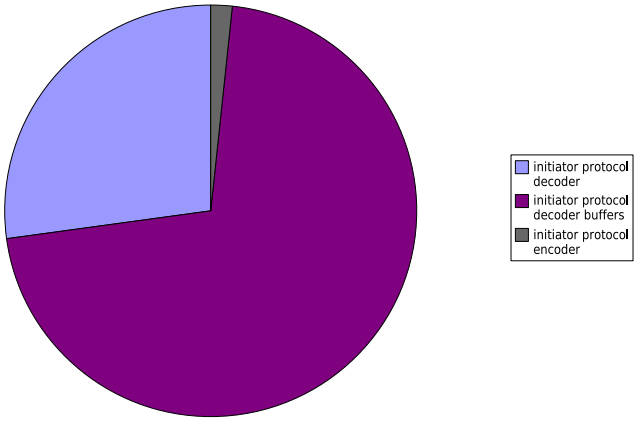


Figure 9.15: Area of the components of a single requestor interface

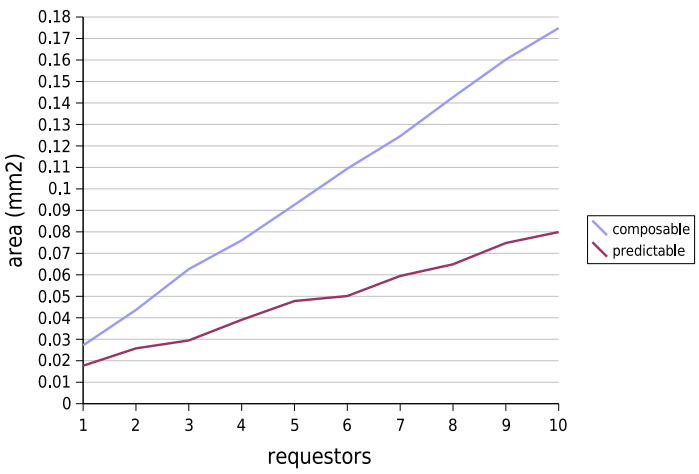
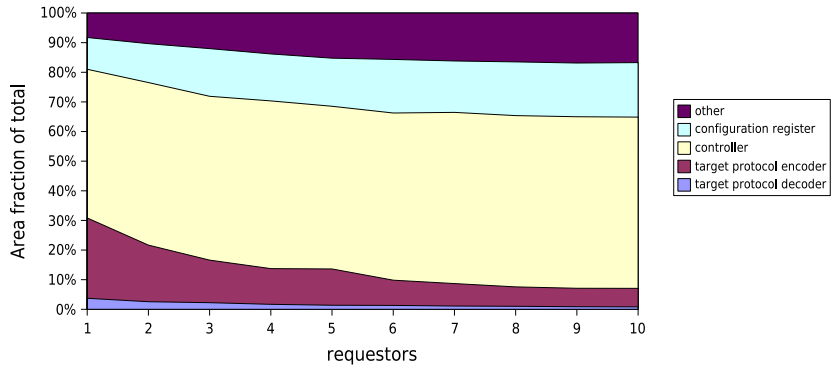
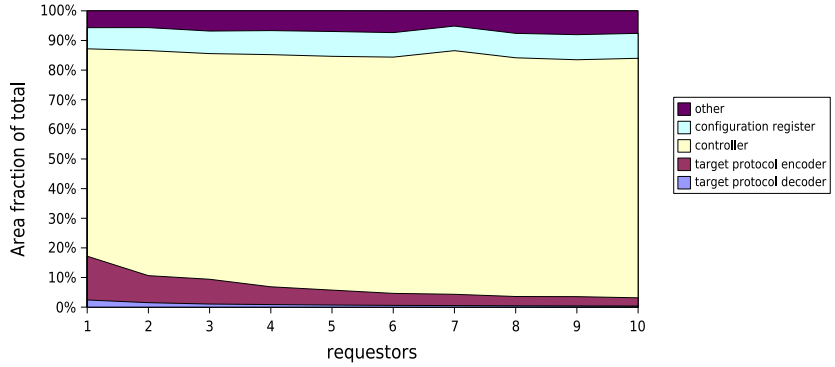


Figure 9.16: Area of the arbiter and back-end interface

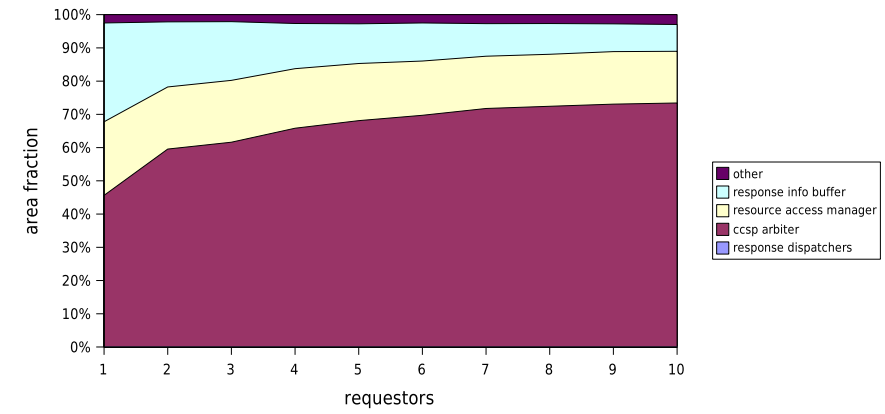


(a) Predictable

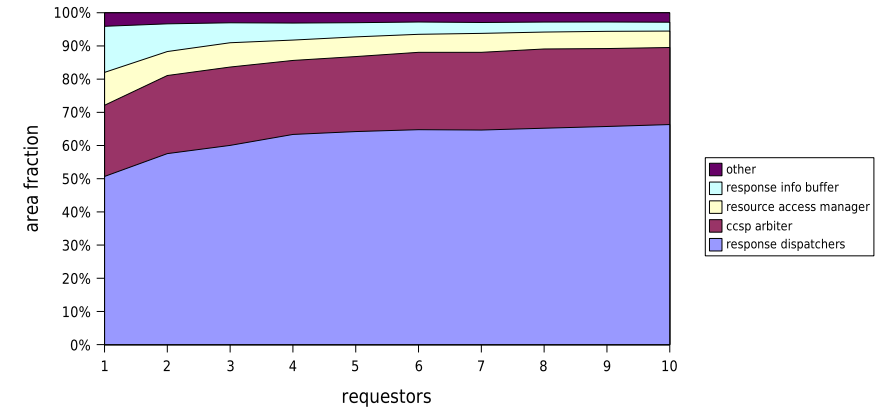


(b) Composable

Figure 9.17: Area of the components of the arbiter and back-end interface



(a) Predictable



(b) Composable

Figure 9.18: Area of the components of the controller

Conclusions
10

Today, verification and integration is the major cost of the development of System-on-chips. This memory controller front-end, in combination with a suitable back-end, is able to give bounds on latency and net bandwidth. In addition, it is capable of isolating the behaviour of different requestors. These properties help reducing the integration and verification time of a SoC. Furthermore, the front-end has a modular design that does not depend on a specific memory technology.

A predictable memory access to pattern map (PAM) has been proposed to model an efficient and predictable back-end. A composable variant (CAM) has been introduced that is used to avoid that the behaviour is affected by other requestors. These mappings allow a predictable and composable translation from abstract time units of the scheduler to real time. Any back-end that can be modelled according to a PAM and CAM is suitable for the proposed front-end.

From analysis of the memory command patterns can be concluded that the access granularity of the back-end must increase for newer memory devices to maintain high efficiency. A DDR3-1600 memory device cannot guarantee a higher net bandwidth than a DDR2-400 device for an access granularity of 32 bytes. Although last data latency does improve, first data latency is almost unaffected by the frequency of SDRAM devices.

The modular structure of the front-end has several advantages. It eases static timing analysis that is required for predictability and composability. Furthermore, the front-end can be reused as resource scheduler since it is not tailored to a specific interconnect or resource. The design of the front-end does not exploit advanced techniques to optimize the efficiency or latency, because most methods are only improving average performance, are not analysable, or do not have bounded behaviour.

Composability is obtained by avoiding interference from other requestors. In the first place, requestors do not share hardware when this is not required and could lead to interference. Secondly, requests are delayed to their worst-case latency, such that there are no variations due to other requestors. A non-work-conserving arbiter is used in conjunction with a CAM to prevent that the provided bandwidth is affected by other requestors. The composable front-end takes care that back-pressure does not stall the resource, but only the misbehaving requestor. Resource stalling would affect the behaviour of other requestors. A current limitation for composability is that requestors always must have pending requests.

An interconnect can be slower than the memory controller due to the interface, protocol and clock frequency. Maximum latency and minimum net bandwidth are only guaranteed when the interconnect produces requests and consumes responses fast enough.

Simulation of the front-end shows that the analytical worst-case latency is approximately twice as long as the maximum latency during simulation. A more exact tim-

ing analysis can give tighter bounds, such that the average latency for the composable front-end improves and lower latency can be guaranteed. The average latency of the composable front-end is very high because requests are delayed (ten times higher than the predictable front-end).

The cost of composability in terms of bandwidth is less dramatic because there is not that much room for improvement. The back-end of the simulation has an minimum guaranteed efficiency of almost 83%. The actual efficiency of the composable back-end does not differ. The best case for the predictable back-end is 98.3%.

The bandwidth guarantee can also be improved when the memory command patterns make a distinction between read and write (like read and write access patterns of different lengths).

The predictable front-end consumes $0.201mm^2$ for five requestors in CMOS090LP. The area of the composable front-end is $0.246mm^2$ for the same configuration but excluding the response delay buffers. For the use case, the response delay buffers require 26k bits in total. The estimated size is $0.76mm^2$. Hence, a very large buffer is necessary. The predictable front-end has a maximum frequency of 168 MHz. The response delay buffers have not been synthesized for the composable front-end. For five requestor it runs at an maximum frequency of 164 MHz. When the response delay block is included, the maximum speed is expected to drop significantly. The current hardware implementation is therefore not suitable for current memory devices. However, the implementation has not been optimized for speed. A pipelined implementation can increase the maximum speed at the cost of a couple cycles.

The purpose of predictable and composable architectures is to reduce development effort. However, this project shows that they also introduce new development problems and have a cost in terms of performance and area.

Future work
11

The latency and net bandwidth guarantees of the memory controller depends on the access method of the back-end. The proposed method uses memory command patterns that exploit bank interleaving. However, this method is not scalable when the request granularity does not get coarser. Research on other access methods is necessary for high guaranteed efficiency of modern and future memories. The use of an existing memory controller as back-end reduces development time. However, the behaviour must be modelled by a PAM and CAM for predictability and composability, respectively. Current memory controllers must be analysed if they satisfy the requirements and still can deliver a high net bandwidth and low latency.

The average performance can be improved when the patterns distinct between reads and writes. Currently, the access and refresh patterns are assumed to have the worst-case length. Refresh patterns could be executed when the memory is idle for example and only be forced when necessary. However, the guaranteed net bandwidth is harder to derive.

A more accurate and formal data flow model could help to minimize the worst-case latency. This is especially important for the composable front-end because the latency of all requests are delayed to the worst case.

This project did not present methods to calculate the buffer sizes analytically. These methods can be used to explore the requirements of more use cases.

A solution has to be found for the huge response delay buffers that are required for composability. First, they are expensive and secondly, no suitable hardware implementation has been found yet. A static RAM (SRAM) could be cheaper than a large register file. A comparison could be made between the performance and area of a front-end with the CCSP arbiter and a TDM scheduler for example. A TDM scheduler is composable since requests are executed at fixed times. Only the interference of the back-end interface must be compensated by the response delay. Hence, a much smaller response delay is required. The arrival time of requests at the arbiter have to delayed to the worst-case to make the front-end completely composable. More research is needed to determine the cost of composability and predictability in terms of performance and power. A comparison between composable, predictable and current memory controllers could be performed.

The speed of the front-end can be improved when more pipelining is applied. The CCSP arbiter has a relatively high propagation time. We expect that a two stage CCSP arbiter could increase the speed of the front-end significantly. A clock bridge between the requestor interface and arbiter can be implemented to allow higher frequency of the requestor interfaces. This has the advantage that the allocated service of requestors is not limited.

Bibliography

[1] *International Technology Roadmap for Semiconductors (ITRS) - Design*, 2007, http://www.itrs.net/reports.html.

[2] Benny Akesson, Liesbeth Steffens, Eelke Strooisma, and Kees Goossens, *Real-Time Scheduling of Hybrid Systems using Credit-Controlled Static-Priority Arbitration* , Tech. report, NXP Semiconductors, 2007, NXP-TN-2007-00119.

[3] Benny Akesson, Liesbeth Steffens, Eelke Strooisma, and Kees Goossens, *Efficient service allocation in hardware using credit-controlled static-priority arbitration*, submitted to CODES+ISSS (2008).

[4] Kees Goossens Benny Akesson and Markus Ringhofer, *Predator: A predictable SDRAM memory controller*, CODES+ISSS, ACM Press New York, NY, USA, September 2007, pp. 251–256.

[5] Artur Burchard, Ewa Hekstra-Nowacka, and Atul Chauhan, *A real-time streaming memory controller*, Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), 2005, doi:http://ntasbieb.natlab.research.philips.com:2531/10.1109/DATE.2005.34.

[6] Chih-Da Chien, Chih-Wei Wang, Chiun-Chau Lin, Tien-Wei Hsieh, Yuan-Hwa Chu, and Jiun-In Guo, *A low latency memory controller for video coding systems*, IEEE International Conference on Multimedia and Expo, July 2007, pp. 1211–1214.

[7] 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*, Interconnect-Centric Design for Advanced SoC and NoC (Jari Nurmi, Hannu Tenhunen, Jouni Isoaho, and Axel Jantsch, eds.), Kluwer, 2004, pp. 399–423.

[8] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken, *Compsoc - a composable and predictable multi-processor system-on-chip template*, Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE), 2008.

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

[10] Sven Heithecker and Rolf Ernst, *Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements*, DAC '05: Proceedings of the 42nd annual conference on Design automation, 2005.

[11] John L. Hennessy and David A. Patterson, *Computer architecture: A quantitative approach, third edition*, Morgan Kaufmann, San Mateo, CA, 2003.

[12] JEDEC Solid State Technology Association, *DDR2 SDRAM specification*, JESD79-2C ed., May 2006.

[13] JEDEC Solid State Technology Association, *DDR3 SDRAM specification*, JESD79-3 ed., June 2007.

[14] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, *Weighted round-robin cell multiplexing in a general-purpose ATM switch chip*, IEEE Journal on Selected Areas in Communication **9** (1991), no. 8, 1265–1279.

[15] H. Kopetz and N. Suri, *Compositional design of rt systems: A conceptual basis for specification of linking interfaces*, 2003.

[16] Tzu-Chieh Lin, Kun-Bin Lee, and Chein-Wei Jen, *Quality-aware memory controller for multimedia platform SoC*, IEEE Workshop on Signal Processing Systems, SIPS 2003, 2003.

[17] C. Macian, S. Dharmapurikar, and J. Lockwood, *Beyond performance: Secure and fair memory management for multiple systems on a chip*, IEEE International Conference on Field-Programmable Technology (FPT), 2003.

[18] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith, *Fair queuing memory systems*, MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (Washington, DC, USA), IEEE Computer Society, 2006, pp. 208–222, doi:http://dx.doi.org/10.1109/MICRO.2006.24.

[19] Markus Ringhofer, Kees Goossens, and Benny Akesson, *Design and implementation of a memory controller for real-time applications*, Tech. Report TN-2006-00500, Philips Research, sep 2006.

[20] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens, *Memory access scheduling*, ISCA '00: Proceedings of the 27th annual international symposium on Computer architecture, 2000.

[21] J. Roest, *Spider project: Detailed design description of the DDR SDRAM controller*, Tech. Report 1.3, Philips Consumer Electronics, March 2004, Philips confidential.

[22] Andrei Rădulescu, John Dielissen, Kees Goossens, Edwin Rijpkema, and Paul Wielage, *An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming*, Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE) (Washington, DC, USA), vol. 2, IEEE Computer Society, February 2004, pp. 878–883, doi:http://dx.doi.org/10.1109/DATE.2004.1268998.

[23] M. Shreedhar and George Varghese, *Efficient fair queueing using deficit round robin*, SIGCOMM, 1995, pp. 231–242, Available from: citeseer.ist.psu.edu/shreedhar95efficient.html.

[24] Frits Steenhof, *Columbus SDRAM interface*, Tech. Report 0.8, Philips Consumer Electronics, November 2002, Philips confidential.

[25] Liesbeth Steffens, Manvi Agarwal, and Pieter van der Wolf, *Real-time analysis for memory access in media processing socs, a practical approach*, ECRTS (2008).

[26] Dimitrios Stiliadis and Anujan Varma, *Latency-rate servers: a general model for analysis of traffic scheduling algorithms*, IEEE/ACM Trans. Netw. **6** (1998), no. 5, doi:http://ntasbieb.natlab.research.philips.com:2531/10.1109/90.731196.

[27] Wolf-Dietrich Weber, *Efficient shared dram subsystems for SOCs*, Sonics, Inc, 2001, White paper.

[28] Luud Woltjer, *Optimal DDR controller*, Master's thesis, University of Twente, January 2005.

SDRAM command timing constraints

A

Table A.1 shows the actual timing constraints of the devices that have been used to analyse latency and bandwidth (Section 6.4.3). The constraints depend on the additive latency (AL) and burst length (BL). Both parameters must be programmed in a register of the memory. The meaning of the timing constraints can be found in Table 4.1. Time is expressed in cycles, unless mentioned otherwise.

Table A.1: Command timing constraints

device	*DDR2-400*	*DDR2-800*	*DDR3-800*	*DDR3-1600*
speed bin	3-3-3	6-6-6	6-6-6	10-10-10
clock period	5 ns	2.5 ns	2.5 ns	1.25 ns
size	256Mb	256Mb	512Mb	512Mb
page size	2KB	2KB	2KB	2KB
AL	0	5	4	0
BL	8	8	8	8
ACT_R	3	1	2	10
ACT_W	3	1	2	10
4ACT_ACT	10	18	20	32
ACT_ACT	11	24	21	38
ACT_ACT_D	2	4	4	6
ACT_PRE	8	18	15	28
R_PRE	4	10	8	6
W_PRE	9	20	19	24
PRE_ACT	3	6	6	10
R_ACT	7	16	14	16
W_ACT	12	26	25	34
PRE_REF	3	6	6	10
REF_ACT	15	30	36	72
R_R	4	4	4	4
W_W	4	4	4	4
R_W	6	6	7	8
W_R	6	10	13	18
R_RDATA	3	11	10	10
W_WDATA	2	10	9	8

Memory command patterns

B

Tables B.1, B.2, B.3, and B.4 list the memory command patterns that has been used for analysis in Section 6.4.3. Table B.5 lists the symbols of the patterns. In this appendix, time is expressed in cyles. The timing constraints are listed in Table A.1.

Table B.1: DDR2-400 8/4/1 patterns

pattern	*length*	*commands*
read	16	¡(0, ACT-0), (3, RD-0), (4, ACT-1), (7, RD-1), (8, ACT-2), (11, RD-2), (12, ACT-3), (15, RD-3)¿
write	16	¡(0, ACT-0), (3, WR-0), (4, ACT-1), (7, WR-1), (8, ACT-2), (11, WR-2), (12, ACT-3), (15, WR-3)¿
idle	16	¡ ¿
read to write switch	2	¡ ¿
write to read switch	4	¡ ¿
refresh	26	¡(11, REF)¿

Table B.2: DDR2-800 8/4/1 patterns

pattern	*length*	*commands*
read	27	¡(0, ACT-0), (1, RD-0), (4, ACT-1), (5, RD-1), (8, ACT-2), (9, RD-2), (12, ACT-3), (13, RD-3)¿
write	27	¡(0, ACT-0), (1, WR-0), (4, ACT-1), (5, WR-1), (8, ACT-2), (9, WR-2), (12, ACT-3), (13, WR-3)¿
idle	27	¡ ¿
read to write switch	0	¡ ¿
write to read switch	0	¡ ¿
refresh	42	¡(12, REF)¿

Table B.3: DDR3-800 8/4/2 patterns

pattern	*length*	*commands*
read	32	¡(0, ACT-0), (2, RD-0), (6, RD-0), (8, ACT-1), (10, RD-1), (14, RD-1), (16, ACT-2), (18, RD-2), (22, RD-2), (24, ACT-3), (26, RD-3), (30, RD-3)¿
write	32	¡(0, ACT-0), (2, WR-0), (6, WR-0), (8, ACT-1), (10, WR-1), (14, WR-1), (16, ACT-2), (18, WR-2), (22, WR-2), (24, ACT-3), (26, WR-3), (30, WR-3)¿
idle	32	¡ ¿
read to write switch	3	¡ ¿
write to read switch	9	¡ ¿
refresh	59	¡(23, REF)¿

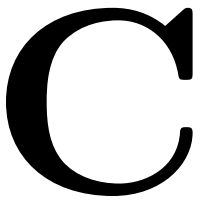
Table B.4: DDR3-1600 8/2/1 patterns

pattern	*length*	*commands*
read	44	¡(0, ACT-0), (6, RD-0), (10, ACT-1), (16, RD-1)¿
write	44	¡(0, ACT-0), (6, WR-0), (10, ACT-1), (16, WR-1)¿
idle	44	¡ ¿
read to write switch	0	¡ ¿
write to read switch	0	¡ ¿
refresh	78	¡(6, REF)¿

Table B.5: Symbols for the patterns

symbol	*DDR2-400*	*DDR2-800*	*DDR3-800*	*DDR3-1600*
t_{access}	16	27	32	44
t_{rtw}	2	0	3	0
t_{wtr}	4	0	9	0
t_{ref}	26	42	59	78
$t_{first\_read\_cmd}$	3	1	2	10
$t_{last\_read\_cmd}$	15	13	30	16
$t_{first\_write\_cmd}$	3	1	2	10
$t_{last\_write\_cmd}$	15	13	30	16

Serialized AXI protocol



The following tables list the default format of the serialized AXI protocol for responses and requests. However, the width and location of the fields within the words are configurable at design time. The only restriction is that they fit in the word and are not moved to other words. Refer to the AMBA/AXI specification for more details about AXI. The default word size of the serialized AXI protocol is 32 bits ($width_{queue}$). The data width of the AXI protocol is denoted by $width_{axi}$.

Table C.1: Read and write requests

word	*bits*	*description*	*axi name*	*notes*
0	31-31	'0' = read request '1' = write request		
0	30-27	burst length	awlen/arlen	1)
0	26-24	burst size	awsize/arsize	2)
0	23-22	burst type	awburst/arburst	3)
0	21-20	lock type	awlock/arlock	4)
0	19-16	cache type	awcache/arcache	4)
0	15-13	protection type	awprot/arprot	4)
0	12-9	reserved		
0	8-0	address id	awid/arid	
1	31-0	address		
$2+(X+1)b+i$	31- 0	write data word i of burst b	wdata[$(32b+31)$-$(32b)$]	5)
$2+(X+1)b+X$	31-A	reserved		5)
$2+(X+1)b+X$	$(A-1)$- 0	strobe of burst b	wstrb	5)6)

Notes:

1) number of transfers = $burst\ length + 1$
2) size of transfer = $2^{burst\ size}$ (bytes), assumed to be equal to $width_{axi}$
3) burst type is assumed to be INCR
4) not supported/ignored
5) only available for a write request
   $0 \leq b \leq burst\ length$
   $0 \leq i < X$
   $X = width_{axi}/width_{queue}$
   $A = width_{axi}/8$
6) when bit m is high, byte lane m must be written
   mapping of byte lanes to write data:
   byte lane 0 = data bits 7-0
   byte lane 1 = data bits 15-8
   ...

Table C.2: Read and write responses

word	*bits*	*description*	*axi name*	*notes*
0	31-31	'0' = read response '1' = write response		
0	30-27	burst length		1)
0	26-25	response status	rresp/wresp	
0	24-9	reserved		
0	8-0	id tag	rid/wid	
$1 + Xb + i$	31-0	read data word i of burst b	rdata[$(32b + 31)$-$(32b)$]	2)

Notes:

1) number of transfers = $burst\ length + 1$

2) only available for read data
$0 \leq b \leq burst\ length$
$0 \leq i < X$
$X = width_{axi}/width_{queue}$

CCSP arbiter pseudo code

D

This appendix contains the pseudo code of the CCSP arbiter like it has been implemented in VHDL. Table D.1 shows the interface. In VHDL, the code is implemented in a separate process. A software implementation could execute the code when some input changes.

```
1  // Check which requestors are eligible:
2  // 1. have enough credits to be scheduled
3  // 2. are schedulable
4  // 3. is not scheduled currently
5  // 4. credits are being updated in this cycle
6  //    (this implies that a requestor can only be scheduled
7  //     when credits are being updated)
8  for r = 0 to REQUESTOR_COUNT - 1
9  {
10   if (credits[r] >= (size[r] * denominator[r]) - numerator[r]
11       schedulable_requestor[r] and
12       (scheduled_mask[r] or update_scheduled_mask) and
13       update_credits)
14   {
15     unmapped_eligible_requestor[r] = true
16   }
17   else
18   {
19     unmapped_eligible_requestor[r] = false
20   }
21 }
22
23 // Determine the next requestor to schedule
24 if (update_scheduled_mask)
25 {
26   // Optional priority map
27   if (ENABLE_PRIORITY_MAP)
28   {
29     for r = 0 to REQUESTOR_COUNT - 1
30     {
31       mapped_eligible_requestor[r] =
32         unmapped_eligible_requestor[priority_map[r]]
33     }
34   }
35   else
36   {
37      mapped_eligible_requestor = unmapped_eligible_requestor
38   }
39
```

```
40    // Select an eligible requestor with the highest priority
41    found_eligible = false
42    for r = 0 to REQUESTOR_COUNT - 1
43    {
44      if (mapped_eligible_requestor[r] and not found_eligible)
45      {
46        scheduled_mask[r] = true
47        found_eligible = true
48      }
49      else
50      {
51        scheduled_mask[r] = false
52      }
53    }
54
55    // Optional work conservation: select schedulable requestor
56    // with highest priority, also when it is not eligible
57    work_conservation = false
58    if (ENABLE_WORK_CONSERVATION and not found_eligible)
59    {
60      found_schedulable = false
61      for r = 0 to REQUESTOR_COUNT - 1
62      {
63        if (schedulable_mask[r] and not found_schedulable)
64        {
65          scheduled_mask[r] = true
66          found_schedulable = true
67          work_conservation = true
68        }
69        else
70        {
71          scheduled_mask[r] = false
72        }
73      }
74    }
75
76    // Optional priority unmap
77    if (ENABLE_PRIORITY_MAP)
78    {
79      for r = 0 to REQUESTOR_COUNT - 1
80      {
81        scheduled_mask[priority_map[r]] = mapped_scheduled_mask[r]
82      }
83    }
84    else
85    {
86       scheduled_mask = mapped_scheduled_mask
87    }
88
89    eligible_requestor = unmapped_eligible_requestor
90  }  // end if update scheduled mask
```

```
91
92  if (update_credits)
93  {
94    // Update credits
95    for r = 0 to REQUESTOR_COUNT - 1
96    {
97      // A requestor that is scheduled to conserve work
98      // does not need to pay for the resource usage,
99      // so they are handled like it is not scheduled
100     if (scheduled_mask[r] and not work_conservation)
101     {
102       credits[r] = credits[r] + (numerator[r] - denominator[r])
103     }
104     else if (schedulable_requestor[r])
105     {
106       if (overflow(credits[r] + numerator[r]))
107       {
108         credits[r] = credits_init[r]
109       }
110       else
111       {
112         credits[r] = credits[r] + numerator[r]
113       }
114     }
115     else
116     {
117       if (credits[r] + numerator[r] > credits_init[r])
118       {
119         credits[r] = credits_init[r]
120       }
121       else
122       {
123         credits[r] = credits[r] + numerator[r]
124       }
125     }
126   } // end for each update credits
127
128 } // end if update credits
```

Table D.1: CCSP arbiter interface

	port	*description*
in	credits_init[n]	initial number of credits of all requestors
in	numerator[n]	Numerator of the service rate of all requestors
in	denominator[n]	Denominator of the service rate of all requestors
in	priority_map[n]	integer for each requestor which represents the priority of the requestor (lower value = higher priority)
in	schedulable_requestor[n]	mask of the requestor that are allowed to be scheduled
in	size[n]	size of the pending request for each requestor
in	update_scheduled_mask	determine the next requestor
in	update_credits	update the credits of all requestors
in	ENABLE_PRIORITY_MAP	enable priority map, when false the index of the requestor determines the priority (lower id = higher priority)
in	ENABLE_WORK_CONSERVATION	enable work conservation
out	scheduled_mask[n]	one-hot mask of the requestor that is scheduled
out	eligible_requestor[n]	one-hot mask of the requestors that are eligible (are schedulable and have enough credits)

Abstraction layers

E

The abstraction layers proposed in this report are visualized by two figures. Figure E.1 shows the abstraction layers on the data domain. This domain has four layers: request, memory access, burst and memory cells. Requests map to memory accesses according to their address and size. The figure shows an unaligned request. Hence, more memory accesses are necessary. It is assumed that requests always align with bursts. Smaller amounts of data can be written by using a write mask. This mapping is done at run-time as address and size are not constant at design-time. The remaining two translations (memory accesses to burst and burst to memory cells) are fixed at run-time. The size of a memory access (access granularity) determines the number of bursts. The number of memory cells of a burst depends on the programmed burst length. The data of a memory access is the service unit of the scheduler.

Figure E.2 shows the abstraction layers of the time domain. The following layers can be identified: request, memory access, memory command patterns and SDRAM commands. The type, address and size of a request determine the memory accesses to execute. A memory access to pattern mapping (PAM or CAM) defines the patterns that have to be issued for a memory access. In the figure the PAM of Table 6.4 on page 43 is used. The final step is the conversion of the patterns to SDRAM commands. A simple lookup table can be used since the patterns are static. The execution time of a memory access is the service cycle of the scheduler.

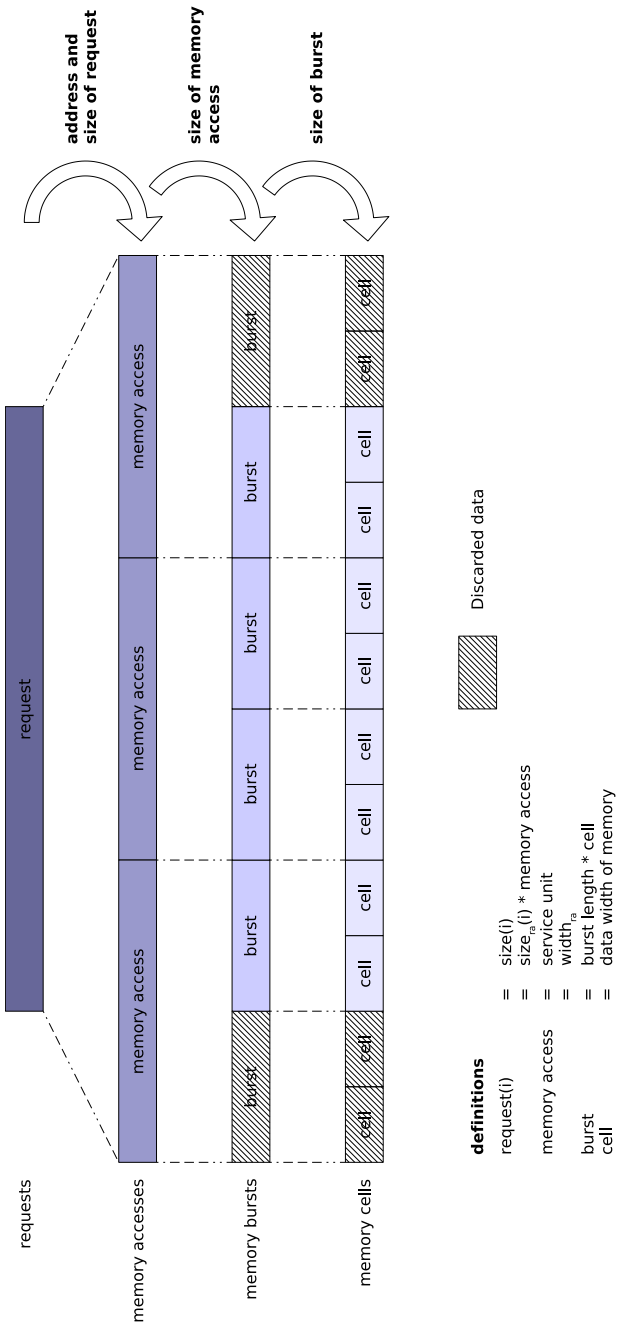


Figure E.1: Abstraction layers on the data domain

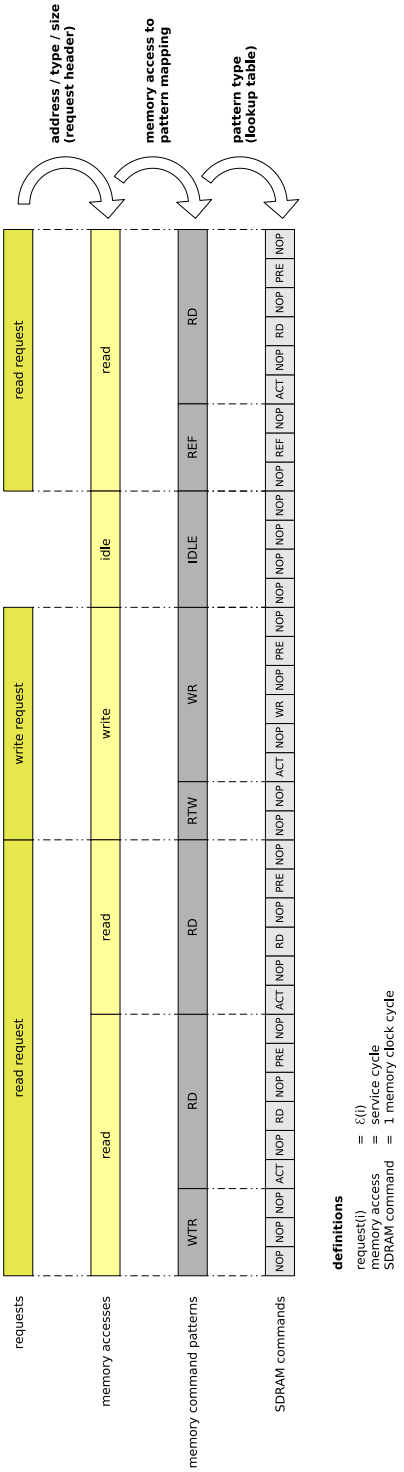


Figure E.2: Abstraction layers on the time domain