UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics & Computer Science

CλaSH-based Framework for Hardware Generation of Optimised Real-Time SDRAM Interfaces Using Static Memory Access Patterns

Sander Huisman

M.Sc. Thesis
June 2020

Summary

In real-time systems it is essential that the worst-case latency bound of each task is known. On FPGAs, designers have more control over the implementation such that determinism is easier to guarantee but inherently requires more development effort. A modular real-time framework is being design in the high-level hardware description language CλaSH to support designers. This framework lacks a memory interface for off-chip data storage and hence the research question of this thesis: *How can design-time knowledge of memory access patterns be used to generate an optimised real-time memory interface using a framework in CλaSH?* This thesis aimed to create an analysable real-time memory DRAM interface with deterministic actively managed buffers.

It has been found that memory row switching contributes significantly to the access time of DRAM. Additionally, it must be periodically refreshed to maintain its state with temporary unavailability as a result. Sophisticated memory controllers manage the SDRAM and translate memory requests from the application to sequences of commands. Hence, the command scheduler has a significant influence on the latency of requests. When the access pattern is known at design time, it can be determined when row switches occur such that lower latency bounds can be achieved.

A memory interface has been designed with an open-row real-time memory controller at its base. To exploit the open-row policy, the actively managed buffers are connected to private banks such that each bank has isolated access with bound latency. Additionally, the buffers schedule predefined sequences such that the number of row switches is known at design-time and a lower latency bound can be guaranteed. The refresh logic in the memory controller performs refreshes in burst mode to extend the access interval between refreshes. The logic exposes a refresh ready signal such that the buffers can synchronise their request scheduler and thus eliminate requests during memory unavailability. The buffer synchronises with the application such that the correct data is stored at the correct time.

An abstract model of the memory controller has been realized and simulated in CλaSH. This model uses a simple bank model with fixed access latencies for row-hits and row-misses which are derived from a real-time memory controller. Of this implementation, a dataflow model has been made that can be used for timing analysis. On top of the memory controller a buffer has been realized with a 2D image convolution algorithm as use-case. The buffer performs alternating write and read sequences and is highly parametrised. This allows to tailor the buffer to the application and storage is scaled to those parameters. The memory controller and buffers have to be manually connected, but the process is simplified by the type system of CλaSH. A dataflow model

of the use-case implementation has been made that matches the simulations.

Simulations show that the buffer is able to schedule request sequences with bound latency based on the access pattern of the application. The current implementation performs the request sequences in a best-effort fashion such that an earlier finish of a request accumulates during the sequence. This can lead to non-determinism as the data path might not be ready. This mechanism can be improved in future work by implementing backpressure based on the data path. Further research can reduce the temporarily storage of data inside the buffer and reduce the request queue of the memory controller.

Glossary

BC	Buffer Back Controller. The back controller is the slave controller of the buffer and performs memory requests on behalf of the front controller.
BlockRAM	Dedicated memory containing several kilobits of RAM. The Artix-7 series *Field-Programmable Gate Array* (FPGA) contains between 5 and 1880 dual-port BlockRAM of 36 kB each. Each port is completely independent ports that share nothing but the stored data [1].
CλaSH	CAES Language for Synchronous Hardware. A functional hardware description language that borrows both its syntax and semantics from the purely functional programming language Haskell. The corresponding compiler transforms the high-level descriptions to low-level synthesisable descriptions such as VHDL and Verilog. The language and compiler are originally developed by the CAES group of the University of Twente and is nowadays maintained by the spin-off company QBayLogic.
CAES	Computer Architecture for Embedded Systems group of the University of Twente.
FC	Buffer Front Controller. The front controller is the master controller of the buffer and manages the buffer storage. It is synchronised with both the application and memory controller such that the correct data is stored at the correct time.

List of Acronyms

BC	Back Controller.
CAES	Computer Architecture for Embedded Systems.
CAS	Column-Address Strobe.
COTS	Commercial-Off-The-Shelf.
CSDF	Cyclo-Static Dataflow.
DDR	Double Data Rate.
DRAM	Dynamic RAM.
DSE	Design Space Exploration.
FC	Front Controller.
FPGA	Field-Programmable Gate Array.
FWFT	First Word Fall Through.
HSDF	Homogeneous Synchronous Dataflow.
ISA	Instruction Set Architecture.
MIG	Memory Interface Generator.
NoC	Network-on-Chip.
REPL	Read–Eval–Print Loop.
RTL	Register Transfer Level.
SDF	Synchronous Dataflow.
SDRAM	Synchronous DRAM.
SRAM	Static RAM.
WCET	Worst-Case Execution Time.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Robots and other systems used in mission-critical applications interact with a fast-changing environment where a small failure can have catastrophic consequences. Many of those systems depend on sophisticated algorithms with massive computational resource requirements but essential in those systems is real-time performance that guarantees that a task finishes within a constrained time window. This requires the worst-case latency bound of each task to be known. On general-purpose processors, tasks are composed of imperative statements that contain many branches such that the *Worst-Case Execution Time* (WCET) of a task is difficult to determine. Additionally, tasks have shared resources such that sophisticated operating systems with predictable schedulers are required to schedule tasks deterministically. Consequently, WCETs are often over-approximated, which leads to pessimistic estimates and thus require more powerful (micro-)processors.

In contrast to processors that have a fixed architecture, FPGAs contain an array of thousands of programmable logic blocks connected to reconfigurable interconnects that can be used to create digital circuits inside the chip. Designers have more control over the implementation such that determinism is easier to guarantee but inherently requires more development effort. The *Computer Architecture for Embedded Systems (CAES)* group of the University of Twente (Enschede, The Netherlands) is developing a modular hard real-time framework for FPGAs using the functional hardware description language CλaSH to reduce the design time for FPGAs [5]. CλaSH borrows its syntax and semantics from the purely functional programming language Haskell which enables the use of high-level mathematical descriptions with a high degree of abstraction. The goal of the framework is to offer a highly analysable system with deterministic behaviour such that static schedules can be derived at design time that guarantees hard-real-time performance. This architecture deals with challenges such as modularity and multiple clock domains that are required to create modular cyber-physical systems.

On-die memory on an FPGA is scarce such that applications with high memory demands in both bandwidth and memory size require large and high-speed external memories such as DDR *Synchronous DRAM* (SDRAM). Due to their construction, they have high access latencies that depend on previous requests. Moreover, DRAM must be periodically refreshed to maintain its state which causes the memory to be temporarily unavailable. Sophisticated memory controllers manage the DRAM and translate memory requests in sequences of DRAM commands. Hence, the command scheduler has a significant influence on the latency of requests. To increase the performance of requests, caches buffer small portions of data in on-die storage and thus reduce the number of requests to the external memory. Due to the limited size, cache misses are unavoidable such that data has to be requested from the external memory with high latency as a result. The fundamental problem is that a cache buffers data based on previous requests and is unaware of future requests of the application. Scratchpad memories are also small high-speed memories that can be used to store application data temporarily but must be managed by the application. When the memory access pattern of the application is statically known at design time, it can be used to buffer the correct data at the correct time and thus create deterministic storage.

1.2 Problem Description

This section focusses on the scope of the project. In this section, the problem statement, objective and limitations will be defined.

1.2.1 Problem Statement

The hard real-time framework in CλaSH lacks a high-performance deterministic memory interface for external Dynamic RAM. Commercial-Off-The-Shelf memory controllers are optimised for high bandwidth and have pessimistic, or even unbounded, real-time latency bounds. Traditional caches reduce the number of memory requests by temporarily buffering memory data in high-speed on-die storage. Caches buffer memory data based on previous requests and are unaware of future memory requests of the application with increased worst-case latency as a result.

The research question is formulated as follows:

> How can design-time knowledge of memory access patterns be used to generate an optimised real-time memory interface using a framework in CλaSH?

1.2.2 Objective

The objective is to create a framework in CλaSH that can generate an optimised real-time memory interface for external SDRAM that exploits design-time knowledge of the memory access pattern of the application and the known behaviour of a real-time memory controller. The memory interface should be simulatable in CλaSH such that the behaviour of the complete can be verified during development. Furthermore, a dataflow model of the interface must exist such that the real-time performance can be analysed.

1.3 Approach and Outline

There exist various memory data access patterns with each a different behaviour. This thesis demonstrates an image processing use-case, by creating a real-time memory buffer that uses knowledge about the data access pattern of a 2D convolution algorithm. The convolution of an image with a kernel is a fundamental operation in image processing and is used in among others filtering (e.g. Gaussian blur) and edge detection (differentiation). The convolution algorithm exhibits a simple memory access pattern but requires specific buffering to prevent repeated access to the same data. A naive approach will be used in which images are first streamed to the memory and after which the image is read from the external memory and processed by the image convolution algorithm. The image is assumed to be received as a grey scale bitmap such that both storage and processing is performed linearly.

The thesis starts with background information and related work in Chapter 2 and Chapter 3, respectively. Chapter 4 contains a conceptual buffer architecture based on choices made in a Design Space Exploration. A realisation of the image processing buffer is described in Chapter 5. The image processing use-case is described and implemented in Chapter 5.2. The thesis is concluded in Chapter 6.

Chapter 2

Background

In hard real-time systems, tasks must finish within a specified time window, and exceeding the deadline is considered as a system failure with possibly catastrophic consequences. Therefore, it is important that tasks have a bounded worst-case execution time such that a deterministic schedule can be guaranteed. The external memory has a high variable latency that depends on the state of the memory and previous accesses. Memory controllers manage the external memory and translate requests from the application to sequences of requests to the external memory. The memory controller has a significant influence on the latency by scheduling commands to the external memory in a specific order. To increase the performance of memory requests, caches are often used to temporarily buffer memory data in fast on-die memory and thus reducing the number of requests to external memory. Caches buffer data based on previous requests and are unaware of future requests by the application to memory. When the access pattern of the application is known at design time, this knowledge can be used to prefetch data needed by the application and to determine the worst-case latency of combinations of memory requests.

This chapter focusses on the background of DRAM, memory controllers and caches to understand the difficulties in current memory components and where latencies arise. Furthermore, this chapter highlights properties of the hardware description language CλaSH and dataflow analysis that can be used to derive analyse systems. The chapter starts with the fundamentals of DRAM in Section 2.1, followed by how the memory controllers manage the external memory in Section 2.2, and how caches increase the memory performance by buffering data in Section 2.3. After that, CλaSH and dataflow analysis are described in Section 2.4 and Section 2.5, respectively.

2.1 DDR SDRAM

Many FPGAs have support for high-speed external memories such as DDR2, DDR3 and DDR4, which are *Double Data Rate* (DDR) *Synchronous DRAMs* (SDRAMs). Compared to DRAM, the DDR DRAM uses the so-called double pumping technique to transfer data on both clock edges. It doubles the data bus bandwidth without increasing the signal frequency and thus has a lower signal integrity.

This section focusses on the internals of a memory chip and where major latencies arise.

2.1.1 DRAM Cell

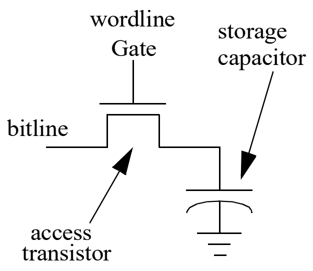


Figure 2.1: DRAM Cell [2]

DRAM is a type of RAM that stores the state of a bit in a tiny capacitor that can be charged or discharged to represent either a logic ‘0’ or a logic ‘1’ as shown in Figure 2.1 [6]. Furthermore, a memory cell contains a transistor as a switch to connect or disconnect the capacitor to the read and write logic. Due to self-discharge caused by the leakage current of the capacitor, it needs to be regularly refreshed to maintain its state [2]. During a refresh, the memory is unavailable for read and write-requests and thus has a significant effect on the access latency. A longer refresh interval results in a higher bandwidth but may corrupt date as the state cannot be reliably read. *Static RAMs* (SRAMs) do not require a refresh as a memory cell contains a flip-flop to store the state of a bit, which makes SRAMs faster than DRAMs [7]. However, DRAMs need fewer components and thus allows a denser and thus cheaper memory. Nonetheless, both memories are volatile as their state is lost after power loss.

2.1.2 DRAM Architecture

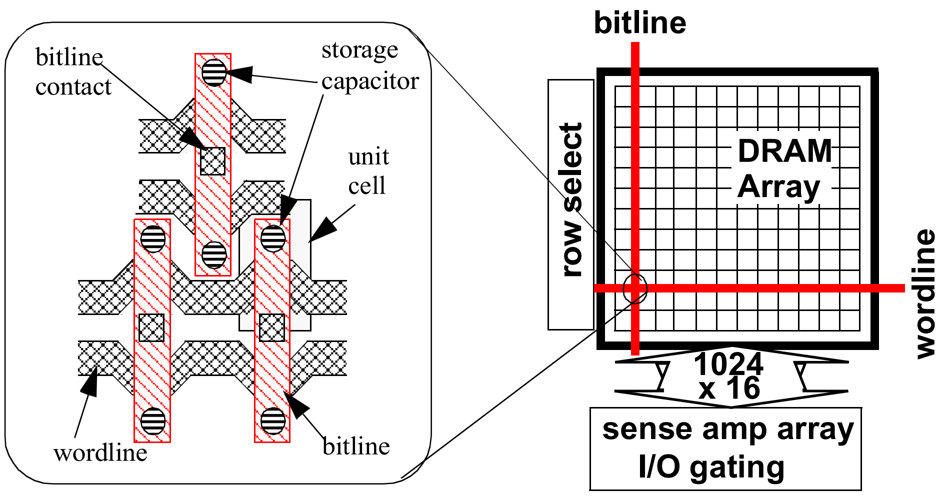


Figure 2.2: DRAM Array with row buffer [2]

Millions of memory cells are arranged in a two-dimensional grid array of rows and columns combined with a row buffer, as shown in Figure 2.2. The memory chips on the DDR3 module used in the Xilinx AC701 evaluation kit, for example, has over 130 million memory cells in a single array to create a memory bank of 16 MB. Memory access is based on row access in which a whole row is loaded in the row buffer.

On selection of a row, the corresponding *wordline* is activated which connects all cells in that row to their *bitline*. Therefore, the memory array is physically limited to maximal one active wordline at a time. Nevertheless, on activation of a wordline, thousands of cells are activated concurrently. The memory cells are as small as possible such that the capacity of the wordlines and bitlines is significantly larger than the capacitance of individual memory cells. When a wordline is activated, the capacitance of memory cells results in a small voltage change such that sophisticated amplifiers are required to restore the cell's state [2]. Memory arrays use a folded bitline architecture in which pairs of memory cells are alternating connected to a pair of bitlines. As a pair of bitlines are closely matched, the unconnected bitline can be used as a reference voltage for the sense amplifier. Another advantage is that a folded bitline structure has a better common-mode noise rejection compared to a single bitline. Before a wordline *activation*, the bitlines have to be *precharged* to the reference voltage, which is commonly half the supply voltage. On *activation*, the capacitors are connected to their bitline, and the charge from the capacitors is placed on the bitline. The differential sense amplifiers measure the voltage swing and drive the lines to full voltage levels such that logic levels can be read. Consequentially, the capacitors in that row are refreshed. A refresh of the memory row is performed by activating each memory row. When data has to be written to the memory cells, the bitlines are enforced to the new state. After the state of the capacitors is restored by the sense amplifiers, the row is activated, and the sense amplifiers likewise act as a row buffer. After that, the it allows fast read and write access to this buffer. Column access is achieved by multiplexers that select the column bits within the row. As precharge and activate is slow, the precharge and activation times contribute significantly to the access latency of DRAM.

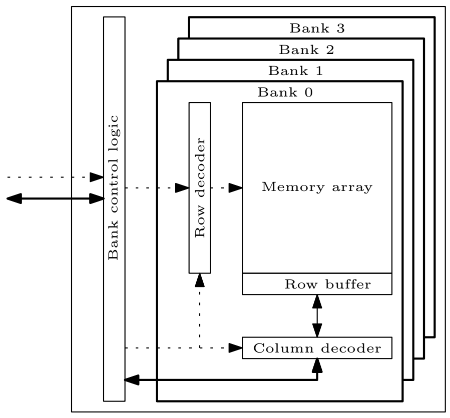


Figure 2.3: DRAM memory chip with 4 banks

As the speed of memory arrays degrades with the size, memory chips contain multiple memory arrays called *banks* to minimise the degradation, as shown in Figure 2.3. Each memory array contains a dedicated row buffer, row decoder and column-decoder, and operates therefore independently from other memory arrays. However, communication to the memory is constrained to a single command and data bus. The command and data bus operate semi-independent such that commands to a bank can be sent during data transmission of another bank. Therefore, the order of commands significantly influences the efficiency of the data bus. Besides increased memory capacity, multiple independent banks allow interleaved access to different banks such that idle banks can be accessed while other banks can be busy with precharging or activating. Memory modules are often composed of multiple memory chips with a shared command and address bus to extend the data bus width. Such a group of memory chips is called a rank, and each chip contains a part of a memory word. Multiple ranks can be used to allow interleaved access to different ranks such that one rank can be accessed while another bank is being refreshed.

2.1.3 Timing Constraints

As described in the previous section, memory is accessed by read and write requests to the row buffer. To access a memory address of a bank, the corresponding row has to be loaded in the row buffer with an *Activate* command after which *Read* and *Write* commands can be issued to access the data. When a memory address does not match the loaded row, another row has to be activated by a *Precharge Activate* command sequence. Access to a *Closed* row is called a *row miss* and access to an *Open* row is called a *row hit*. The *Read* and *Write* commands are also referred to as *Column-Address Strobe* (CAS) commands and are always executed in bursts with a fixed length. In DDR3 and DDR4 the typical burst length is 8, and due to the double data rate, a burst takes 4 clock cycles.

Table 2.1: Timing constraints in data bus clock cycles [4]

Constraint	Description	*DDR3-1066F*	*DDR3-1866M*
t_{CK}	Clock unit	1.875 ns	1.07 ns
t_{RCD}	*Activate* to *Read*/*Write* delay	7	13
t_{RP}	*Precharge* to *Activate* delay	7	13
t_{RC}	*Activate* to *Activate* delay (same bank)	27	45
t_{RAS}	*Activate* to *Precharge* delay	20	32
t_{WL}	*Write* to data bus transfer delay	6	9
t_{RL}	*Read* to data bus transfer delay	7	13
t_{RTP}	*Read* to *Precharge* delay	4	7
t_{WR}	End of *Write* data transfer to *Precharge* delay	8	14
t_{RRD}	Activate to *Activate* delay (different bank)	4	6
t_{FAW}	Four *Activates* window	20	33
t_{WTR}	End of *Write* data transfer to *Read* delay	4	7
t_{RTW}	*Read* to *Write* delay	9	10
t_{BURST}	Data bus transfer	4	4
t_{CCD}	*Read* to *Read* or *Write* to *Write* delay	4	4

Besides the intra-bank timing parameters, memory chips also have inter-bank constraints as the banks share a common bus interface. This includes among others a minimum time between two activates to different banks, a minimum time between a write followed by a read and vice versa as the direction of the bus needs to be switched, and a window in which maximum four activates can be issued. Therefore, the latency of a memory access does not only depended on the state of the bank itself but also on requests of other banks. The important timing constraints for two bus specifications are listed in Table 2.1. Figure 2.4 illustrates the timing constraints and conflicts of different requests to different memory banks. The diagram shows sequences of DRAM commands to certain banks and the data bus in different rows. In the diagram, a read request is performed to *Bank 0*, *Bank 1*, *Bank 3* and *Bank 4*, and a write request is performed to *Bank 2*. The requests are scheduled in a consecutive fashion such that the write request degrades the performance significantly.

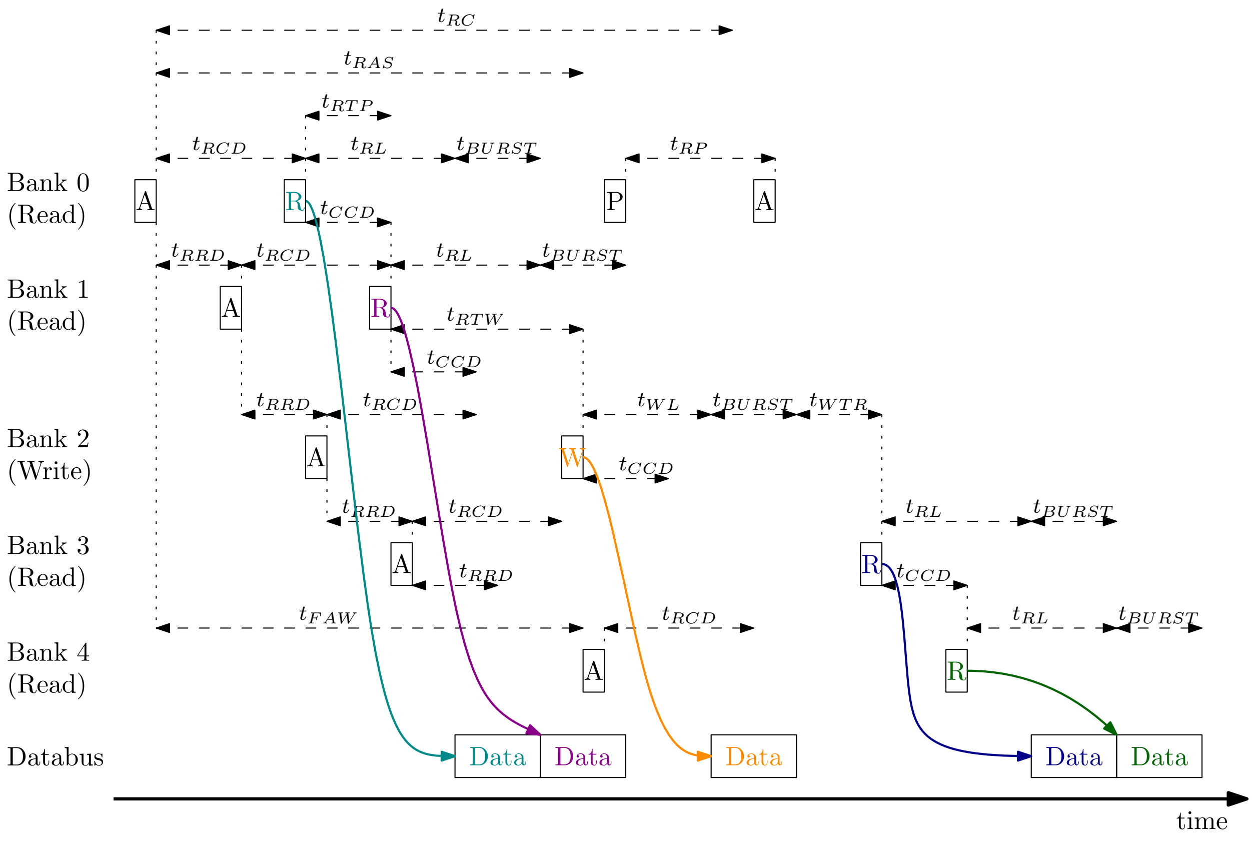


Figure 2.4: Memory access diagram showing timing constraints for five banks performing memory requests

In the diagram in Figure 2.4, *Bank 0* first issues an activate command followed by a read command. *Bank 1* also performs an activate and read, but the activate is delayed as *Bank 0* is executed first. The data transfers can be executed back-to-back as both requests are of the same type. In contrast to *Bank 0* and *Bank 1*, *Bank 2* executes a write sequence. The activate command of *Bank 2* is delayed by the activation of *Bank 0* and *Bank 1*. The write command is delayed as minimum time between a read and write has to be maintained. The same applies to *Bank 3*, which issues a read sequence after *Bank 2* and therefore experiences a write to read delay. *Bank 4* issues the fifth activate command and experiences an additional delay as a maximum of four activates can be issued within a window and overlaps with the write command of *Bank 2* in which the write command is prioritised. The read command is not extra delayed as the previous CAS command is also a read. The delay of the activate command does not have a direct influence on the read command as the read command is delayed more than the activate. Therefore, the data can be transmitted back-to-back with the data from *Bank 3*. Furthermore, *Bank 0* can execute a precharge and activate command before the read commands of *Bank 3* and *Bank 4*, but the precharge is delayed as it would collide with commands from other banks. As the diagram shows, bank interleaving allows a high degree of parallelism, but the scheduling of commands significantly influences its performance if no special scheduler is used.

2.1.4 Refresh

As mentioned earlier, every row in a DRAM should be periodically refreshed to guarantee retention of the data in a typical interval of 64 milliseconds. As the memory is unavailable during a refresh, it is divided in many small periodic refresh cycles of t_{RFC} in which only a single row is refreshed in which the memory is short unavailable. The average refresh interval t_{REFI} is equal to the retention interval divided by the number of rows and thus inverse proportional with the number of rows. The worst-case access latency caused by the memory unavailability is reduced, but more requests experience interference as the row buffer is closed after each refresh. Due to the asynchronous behaviour, memory requests can be interrupted by the refresh and postponed until after the refresh is complete. Moreover, a request to an open row will experience a row-miss after the refresh.

Table 2.2: Refresh cycle count for various module sizes [4] for *DDR3-1066F*

Module size (Gb)	t_{RFC} (data bus clock cycles)
1	59
2	86
4	139
8	187

For DDR3 DRAM modules with a refresh count of 8K, the average refresh interval t_{REFI} is 7,8 µs. As shown in Table 2.2, the refresh cycle time t_{RFC} increases with the size of the module. With growing DRAM sizes and thus increasing number of rows, the average refresh interval reduces significantly since the interval is inverse proportional to the number of rows. For large DRAMs such as a 1 Gb module, a dual refresh cycle is introduced in which two rows are consecutively refreshed [8]. Therefore, the refresh interval is doubled at the cost of a doubled t_{RFC}.

To keep track of which rows have to be refreshed, the DRAM features an internal hardware refresh counter and refreshes successive rows until all rows have been refreshed. The memory controller only has to periodically send a sufficient number of refresh commands at a t_{REFI} interval to maintain data retention. This refresh method is called auto-refresh or *CAS-before-RAS*. The memory controller can also provide a row address to be refreshed. In this mode, the memory controller has to track the state of each row and is often referred to as *RAS Only Refresh*. A common issue is that besides the unavailability of the memory, also the row has been closed as a result of the refresh. Therefore, a request after the refresh that would have hit the same row requires to activate the row first. As t_{REFI} interval is an average interval, the memory controller can send earlier refreshes or even multiple successive refresh commands in a burst such that only a single row-miss occurs after a refresh at the cost of a longer refresh period [9]. DDR3 DRAM, for example, has a maximum burst length of eight refresh commands. Burst refresh can also be beneficial in scheduling as a longer period is available without refreshes. As a refresh is performed to all banks at the same time, a beneficial early refresh for one bank can decrease the performance of other banks.

2.2 Memory Controller

The memory controller translates read and write requests in a sequence of DRAM commands and schedules them such that timing constraints are satisfied. *Commercial-Off-The-Shelf* (COTS) memory controllers typically use request reordering to optimise for maximum efficiency between rank-bank queues. Memory requests are reordered to exploit burst access, which can lead to unbounded latency as requests are delayed in favour of more efficient requests [3, 10, 11].

Besides request reordering, also the *row policy* has a significant influence on the performance of the memory controller. The row policy specifies if the memory controller leaves a row open (*open-row policy*) of closed (*close-row policy*) after a request. An *open-row* policy exploits the caching behaviour of the row buffers and only closes a row in case of a row-miss or refresh. When a request is performed to an open-row (*row-hit*), then only a CAS command is required. Otherwise, the row must be opened first (*row-miss*) in which an activate command and possibly a preceding precharge command must be issued before the CAS command can be performed. Therefore, the access latency of a request with an open-row policy depends on the previous request. In a *close-row* policy, the memory controller automatically precharges the row buffer when no other requests for the currently open row are found in the request queue. Each request can be seen as a row-miss and thus requires an activate-CAS command sequence. All request have the same latency since row-hits are treated as row-miss.

In real-time applications, open-row memory controllers are generally combined with bank privatisation in which a memory client has exclusive access to a bank and other clients cannot destroy the contents of the row buffer. Consequently, it limits the maximum number of memory clients to the number of banks and more storage or a higher bandwidth is achieved by allocating multiple banks for a single client. The bank-privatisation isolates each client such that scheduling is limited to individual clients. The downside is that the memory can be underutilised when a client cannot exploit the full storage capacity of the allocated bank(s) or does not require the potential bandwidth. Close-row controllers are often combined with interleaved access and require exclusive accesses to all banks to guarantee address-independent memory access at a given time slot. In contrast to the private-bank architecture, the number of clients is not restricted to the number of banks and higher bandwidth is achieved by allocating multiple time slots within a period. Additionally, all banks are combined to one storage space such that clients can allocate various memory areas without underutilisation. Consequently, the clients are not isolated such that a schedule must be found that satisfies all clients. Furthermore, interleaved access has a high peak throughput as a request is performed to all banks. For example, DDR3 memory has a 64-bit wide databus such that an individual request transfers 64 bytes and a combined transfer of 512 bytes in a single time slot. In comparison, eight consecutive requests must be performed to a single bank in a private bank architecture to transfer the same amount of data.

2.3 Cache

As external memories such as DRAM have high and variable access latencies, caches are often used to increase memory performance. Caches are small high-speed buffer memories that temporarily buffer portions of data from the external memory [12].

When a request to the memory is performed, the memory address is compared with the contents of a lookup table inside the cache. If the requested address is stored in the cache, a *cache hit* occurs in which the cache will be used to perform the memory access. If the requested address is not in the cache, a *cache miss* occurs, and the data will be requested from the external memory and stored in the cache such that the cache can complete the request. Ideally, the cache has a cache rate of 100% such that the cache can complete all memory requests on its own.

Besides the cached memory data, a cache entry contains a tag that identifies the data. The tag contains part of the memory address such that the data can be uniquely located in the external memory. A cache data line can contain data from multiple consecutive memory addresses in which only a subsection of the memory address is required to locate the data. When cache lines are larger than a memory data unit, multiple memory requests are required to fill the cache line on a cache miss. If only a single memory address is requested, the cache potentially leads to overhead and higher latencies.

As the cache is smaller than the external memory, cache entries have to be replaced by new ones when the cache is full. Commonly, the *Least Recently Used* policy is used as the least recently used item is least likely to be used in the near future. Besides replacement policy, also associativity has a significant influence on the cache contents. Associativity determines which cache locations can be used for certain entries. In a *Fully Associative Cache*, all cache entries can be mapped to all cache locations. A drawback of the fully associative cache is that all tags have to be compared with the requested address which can be slow for large caches. Furthermore, a fully associative cache is expensive in terms of hardware resources as it requires hardware for comparing and routing of all entries. In contrast to a fully associative cache, in a *direct-mapped* cache, a cache entry can only be mapped to a single location that is determined by the address. Therefore, a direct-mapped cache is faster and simpler than a fully associative cache as it requires less hardware to route the data. The disadvantage of a direct-mapped cache is that cache entries are replaced based on the address, which can increase the miss ratio in case two often requested addresses collide. A compromise is an N-way set-associative cache in which a cache entry can be mapped to N locations. Compared to a fully-associative cache the search space is reduced to N locations and thus simpler than a fully associative cache. A one-way set associative cache can also be seen as a direct-mapped cache and a fully associative cache as an N-way set-associative cache with N equal to the cache depth.

Another important parameter is the write policy. The cache contains a copy of data from the external memory and problems can arise when the cache contents are modified through write requests. The *write-back* policy marks entries as dirty when the contents have been modified and must write the contents to the external memory before the entry can be replaced. The *write-through* policy performs write requests to both the cache and external memory to ensure that the memories are synchronised. The advantage of the write-back policy is that write requests to the external memory are reduced when a memory location is modified multiple times before it is removed from the cache. When multiple caches are used, this strategy can lead to undesired behaviour as the caches can contain different data, and the external memories contents cannot be ensured. The *write-trough* can prevent this behaviour but has as a disadvantage that all write requests are forwarded to the external memory, which can lead to extra overhead and increased memory latency as each request can delay other requests.

2.4 CλaSH

Low-level hardware description languages like VHDL and Verilog demand significant engineering effort to design hardware architectures, especially when it comes to advanced algorithms. Many high-level synthesis tools such as *CatapultC* [13], *Vivado HLS* [14] and *Intel HLS* [15] have been developed to accelerate the development by generating a *Register Transfer Level* (RTL) description from programming languages such as C, C++ and SystemC. Those HLS tools mainly concentrate on imperative languages that are originally designed for *Von Neumann-architecture* like computer systems such that software developers can target FPGAs almost the same as they are used to program for a processor. Imperative languages are normally compiled to a list of statements that are sequentially executed by a processor. For an FPGA, the HLS tool has to parallelise the sequential instructions in hardware. Other High-Level languages such as *Chisel* (developed at UC Berkeley) [16] and *CλaSH* (originally developed at the University of Twente) use functional programming languages as their base. Functional languages are more suitable for hardware design as they describe relations rather than the behaviour of values over time.

CλaSH is a functional hardware description language based on the purely functional programming language *Haskell* and borrows both its syntax and semantics. The CλaSH compiler transforms high-level descriptions to low-level languages such as VHDL and Verilog [5] [17]. It features an interactive *Read–Eval–Print Loop* (REPL) interpreter that allows to quickly test functions before translation to RTL. As CλaSH inherits the strong typing system of Haskell, types are statically typed and have type inference in which they are determined at compile time by unifying bidirectionally. If they do not match, the compiler will throw an error which provides a solid base for well-typed code. In the FIFO example shown in Listing 2.2, it can be seen that the FIFO takes a vector of elements as initialisation of the FIFO's memory. As the vector has a fixed length and data type, the FIFO unifies its input and output accordingly. Furthermore, the size of the initialisation vector is used to define the ranges of the head and tail pointer.

```
1 data Fifo a =
2     FifoInput {
3       rReq        :: FifoRead a,  -- Read request
4       wReq        :: FifoWrite a  -- Write request
5     }
6   | FifoOutput {
7       rRes        :: FifoRead a,  -- Read response
8       wRes        :: FifoWrite a  -- Write response
9     }
```

Listing 2.1: FIFO input-output-pair in a record structure

```
fifo :: (
    HiddenClockResetEnable domain
  , KnownNat size, (1 <=? size) ~ 'True
  , NFDataX a
  )
  => Vec size a
  -> Signal domain (Fifo a)
  -> Signal domain (Fifo a)
```

Listing 2.2: FIFO interface type unifying

Part of the high-level abstraction, CλaSH emphasises on the availability of *higher-order* functions. Higher-order functions take a function or expression as a parameter and apply it on a list that, due to type inference, requires no parametrisation. As higher-order functions have a regular structure, they can be well mapped to regular structures in hardware. As CλaSH is based on Haskell, lists have a dynamic length which is impossible in hardware as a physical circuit is realised. Therefore, it is required that the length of a list is known at compile time. Some high-level functions are therefore not usable as they produce a list with a length depended on the input data. CλaSH extends the high-order function concept to *Mealy* and *Moore* state-machines that can be used to create synchronous sequential circuits. Functions in functional programming languages can be chained by function composition in which the result of one function is the input for another function.

Another aspect of functional programming languages is the use of *Algebraic data types*. Algebraic data types are shaped by the *'algebraic'* operations *'sum'* and *'product'* [18]. The 'sum' operator $A|B$ is alternation in which a type can have multiple shapes. The 'product' shape is the combination AB, in which a type has a combination of shapes. The 'sum' and 'product' operators can be composed to create larger shapes. With algebraic data types, an *algebraic language* can be defined in which pattern matching can be used to distinguish between shapes. The FIFO mentioned before, has an algebraic data type to define the inputs (*FifoInput*; line 2-5) and outputs (*FifoOutput*; line 6-9) of the FIFO, as shown in Listing 2.1. The FIFO has an input and output that are combined to an input-output-pair in a ‘sum’ like operation. As the FIFO has a read and write port, the inputs and outputs from both ports are combined in a ‘product’ like operation.

As CλaSH is built on Haskell, algorithms in Haskell can be relative easy transformed to CλaSH by mostly specifying list lengths and types. Besides the translation of algorithms from Haskell to CλaSH, Haskell can also supplement hardware development in various ways. One of the cases is compile-time meta-programming. The *Template Haskell* framework allows type-safe meta-programs to be evaluated at compile time in which the result is used in other Haskell or CλaSH code. For example, the filter coefficients of a FIR filter can be computed by a parametrised algorithm in Haskell and then used as constants in the CλaSH implementation of the filter. Another case is testing in which test frameworks in Haskell such as *HUnit*, *SmallCheck* and *QuickCheck* can be used test algorithms in CλaSH on functional correctness.

To accelerate hardware development, CλaSH can be used to generate sub-modules in low-level languages. Developers familiar with, i.e. VHDL, can describe high-level algorithms in CλaSH with challenging data types such as fixed-point integers and generate a black-box type of modules without the need to describe the whole system in CλaSH such that existing parts in VHDL can be reused. Nevertheless, low-level access to specific parts is also possible through templating.

2.5 Dataflow Analysis

Programming hard real-time applications on multi-core processor systems is often seen as difficult due to concurrent tasks and synchronization between them. Furthermore, it might be challenging to exploit the processing power available as tasks often depend on data from other tasks and thus might have to wait for data to be available. Multi-core processing systems are often seen as software running on CPUs with multiple execution units or multiple CPUs on a single board. FPGAs might be considered multi-core systems as well, since they often contain multiple processing blocks, e.g. algorithms implemented in hardware or softcores, that depend on each other. Dataflow analysis techniques can be used to describe such real-time applications in an abstract model and to derive the behaviour such as throughput and latency. Besides analysis of the performance, it can also be used to determine the minimum buffer sizes between tasks and to derive the number of concurrent instances of tasks to optimize the system.

Several dataflow models exists with each its own properties, such as *Homogeneous Synchronous Dataflow* (HSDF), *Synchronous Dataflow* (SDF) and *Cyclo-Static Dataflow* (CSDF), as shown in the Hasse diagram in Figure 2.5 [19]. While each model has each own properties and complexity, all are based on the same principle. The HSDF model is considered to be one of the simplest models as it has the lowest expressiveness and thus lower complexity. The model has strong analytical properties and can be analysed with several formal semantic models such as Max-Plus algebra and sets of linear constraints. Models with a higher expressiveness can describe more advanced algorithms but have less properties that can be derived from the model. From the models shown in the Hasse diagram, a periodic schedule can be derived for the HSDF, SDF and CSDF graphs and can be statically scheduled.

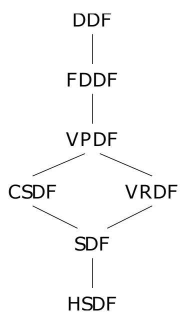


Figure 2.5: Hasse diagram of various dataflow models

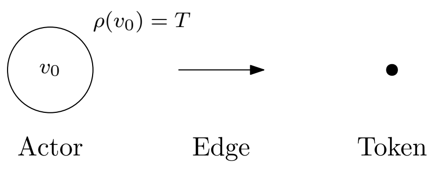


Figure 2.6: HSDF elements

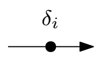
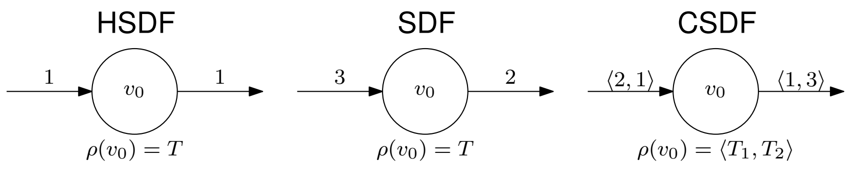


Figure 2.7: Dataflow edge with δ_i tokens

An HSDF model is a graph that contains *actors* represented by nodes that are connected by *edges* represented by arrows. Edges can store tokens represented by dots. The elements of an HSDF graph are shown in Figure 2.6. A visualization of actors in various dataflow models is shown in Table 2.3. When multiple tokens are present on a single edge, it can be shortened to a number next to a dot as shown in Figure 2.7. [19] defines the HSDF model as follows: *A HSDF is defined as a directed graph $G\left(V,E\right)$ that consists of actors $v \in V$ and directed edges $e \in E$ with $e = (v_i, v_j)$.*

Table 2.3: Visualization of various dataflow actors



An actor represents a task such as a computation and is started once there is sufficient data on the input(s) without having to wait for additional input data [20]. The start condition of an actor is called a firing rule and it defines the number of tokens that must be present on each input edge before an actor can fire. The firing rule of an HSDF actor requires at least one token to be present on each input edge. When an actor fires, it consumes the number of tokens defined by the firing rule from each input edge instantaneously and produces a token on each output edge when the actor finishes. In other dataflow models, the number of tokens produced can be more than one or even zero in some cases. The time an actor needs to finish is called the firing duration and is constant for an HSDF actor. As dataflow is an abstraction, a token can represent data, but also space available or for synchronization between tasks. An important feature of dataflow is that the execution of an actor is not exclusive. As long as the firing rule is satisfied, multiple instances of an actor can execute concurrently. This auto-concurrency can be limited to a certain number of instances by creating a so-called self-loop. Those self-loops often obfuscate the graph and are often omitted. An HSDF node with two input edged, a self-edge with δ_k tokens and firing duration T is shown in Figure 2.8.

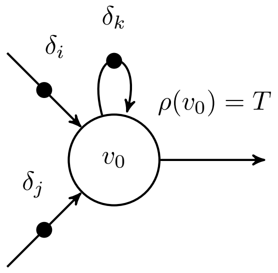


Figure 2.8: HSDF node v_0 with two input edges and a self-edge to reduce auto-concurrency to k instances

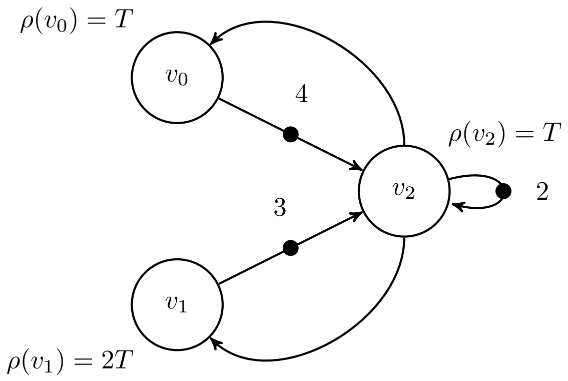


Figure 2.9: HSDF example graph

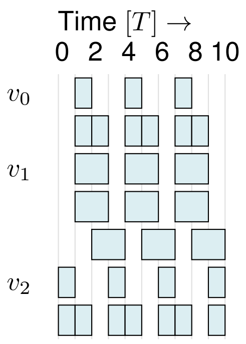


Figure 2.10: HSDF example schedules

An example of HSDF with three actors is shown in Figure 2.9 and the matching schedule is shown in Figure 2.10. Node v_2 has two inputs, one from node v_0 with 4 tokens on it and an edge from node v_1 with three tokens on it. Furthermore, node v_2 has a self-edge that limits its concurrency to two and return outputs to both node v_0 and node v_1. In the graph, node v_2 is the only node that can start firing and fires two times as the firing rule is twice satisfied. After time T, node v_2 finishes its firing and produces a token on each outgoing edge for every firing. This satisfies the firing rule for node v_2 once again as the self-loop contains two tokens, but only one token is available on the edge from node v_1 to v_2. Also, node v_0 and v_1 can fire (both can fire twice) as two tokens are available on their input edge. Node v_0 needs one T time instance to finish and node v_1 needs two

T time instances. Since v_1 needs two T time units and no self-edge is used, the third execution overlaps with the first two.

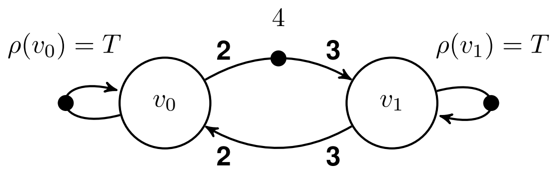


Figure 2.11: SDF example graph

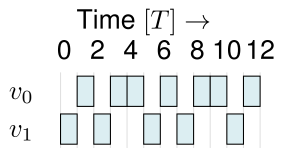


Figure 2.12: SDF example schedule

While HSDF actors consume and produce only one token per firing, an SDF actor consumes and produces one or more tokens per firing. The number of tokens produced or consumed is constant such that each firing consumes and produces the same number of tokens. A simple example is shown in Figure 2.11 with the schedule in Figure 2.12. The graph contains two actors: v_0 and v_1 with each a self-edge and a firing duration of T time units. Node v_0 consumes and produces two tokens from/to node v_1 and node v_1 produces and consumes three tokens from/to node v_0. When the graph starts with the execution, node v_1 can only fire and three tokens move from the top edge to the bottom edge. Node v_1 can now only fire once and two tokens move back to the top edge where there are then 3 tokens. Node v_1 can fire only once, but thereafter, all tokens are located at the bottom edge and node v_0 can fire twice. All tokens are then back at the top edge which is equal to the start condition.

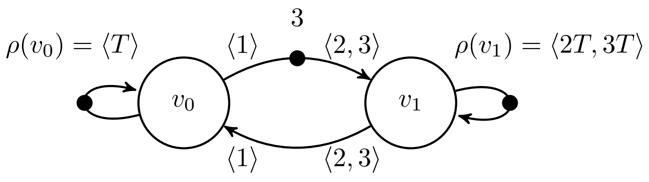


Figure 2.13: CSDF example graph

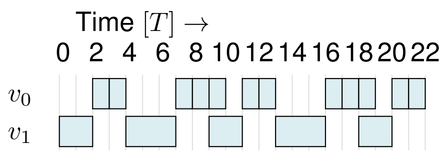


Figure 2.14: CSDF example schedule

The last dataflow model to be mentioned is the CSDF model. In contrast to HSDF and SDF where an actor has a only one firing duration with production/consumption of tokens, an CSDF actor can have multiple firing durations with for each a different production/consumption of tokens. For each firing, the duration and the production/consumption are equal, and firings are in a cyclic order. An example of an CSDF model is shown in Figure 2.13 with its schedule in Figure 2.14. The example contains two CSDF actors: v_0 and v_1. Both actors have a self-edge that prevents auto-concurrency. The self-edges have a production/consumption of one token for each firing and is hidden for clarity. Actor v_0 has only one firing with a duration of T time units and a production/consumption of one token. Actor v_1 has two firings: one firing with a duration of $2T$ with a production/consumption of 2 tokens and one firing with a duration of $3T$ with a production/consumption of 3 tokens. While the cyclic firings of each node can be replicated using the Least Common Multiple (LCM) for a common cycle, the cycles are not always in sync. As shown in the example, node v_0 must fire twice to match the first production and three times to match the second production. This can also be seen in the schedule of the example. After the first firing of node v_1 which takes two time units, node v_0 fires twice. Thereafter, node v_1 fires for tree time units and node v_0 can fire three times. After the second firing (at time 9), enough tokens are produced by v_0 such that v_1 can fire again. When no auto-concurrency is used, tokens might overtake each other as a later firing can finish earlier and thus the property of HSDF and SDF that tokens maintain their order is not guaranteed in CSDF.

While dataflow is a powerful tool that can help with the design of complex systems and derive performance parameters, not all behaviour can be described in dataflow and requires abstraction. For example, the often-used behaviour *choice* cannot be described with HSDF and SDF. It can be described by modelling each branch of the choice behaviour in a separate model or by abstraction that uses the worst-case scenario. As models become larger when describing more complex systems, abstraction can be used to reduce the size and complexity of those graphs. As result, those abstract models are easier to analyse, but are less accurate compared to the implementation.

Chapter 3

Related Work

In the previous chapter, in-depth background information on DRAMs, memory controllers and caches have been given to understand the various access latencies and unpredictability in DRAM components. The DRAM is highly stateful and has various timing constraints. The memory controller manages the DRAM, and schedule commands to the external memory on behalf of the application. Therefore, the command scheduler has a significant influence on the latency of memory requests. Within FPGAs, the memory controller and buffers are implemented in reconfigurable hardware such that the optimal combination of controller and architecture can be used. Extensive research has been performed on high-performance real-time memory controllers.

This chapter focusses on leading research on real-time DDR3 memory controllers and architectures for reconfigurable hardware. Research from various groups is summarised and compared with each other.

3.1 Memory Controller

Multiple Real-Time DDR3 memory controllers have been proposed for FPGAs with different approaches. This section highlights and compares key trends in state-of-the-art memory controllers.

Heithecker et al. proposed a multi-stream mixed-QoS SDRAM controller to achieve high bandwidth utilisation [21, 22]. While the paper targets SDRAM, it is stated that it can be easily adapted to DDR SDRAM. In their first paper, two architectures are described in which one uses request priority scheduling, and the other uses bank priority scheduling. The first variant adds requests from multiple inputs to the matching bank buffer. A priority-based round-robin scheduler is used to schedule bank buffer request. The second variant uses two queues per bank, one for each priority. A multiplexer selects the high priority request if available and otherwise the normal request. Furthermore, it is stated that their round-robin based scheduler has deterministic maximum latency and minimum throughput. In evaluation, the two-stage approach showed high flexibility and efficiency with a 90% memory bandwidth utilisation in simulation. In a second paper, an additional flow controller has been introduced to enhance the overall performance. The flow controller requires a minimum time between two memory read or write accesses by allowing a certain amount of transfers within a period. This allows a burst of multiple successive memory requests. The additional flow control, in combination with access priorities, increases the system performance predictability.

Akesson et al. proposed the Predator memory controller that provides a guaranteed minimum bandwidth and maximum latency bound to the requester for predictable SDRAM sharing [23]. Their approach combines elements from statically and dynamically memory schedulers by defining memory access groups corresponding to static SDRAM access sequences that are dynamically scheduled by a predictable arbiter at run-time. The static sequences have a known efficiency and latency, which leads to a minimum net bandwidth to be provided. The dynamic arbiter is a *Credit-Controlled Static-Priority* arbiter that uses a rate-regulator to isolate requests from different requesters and a static-priority scheduler to provide a maximum latency bound. The isolation of requests has the advantage that the required maximum latency of the request is guaranteed. The scheduler has a modular design and is tightly integrated into the network interface of a *Network-on-Chip* (NoC). Experiments pointed out that all requesters received their allocated bandwidth with a measured lower latency than the maximum latency bound.

Ecco et al. use bundling of read and write requests to reduce the bus turnaround overhead [3, 24–26]. The data bus in DDR memory is bidirectional and is either driven by the memory controller in case of write requests or by the memory chip in case of read requests. Therefore, the direction of the bus needs to be reversed when the type of request changes which requires a significant amount of idle cycles, especially at higher bus frequencies. The researchers propose a modular generic memory architecture that can be used with any private or bank-shared bank mapping, and close or open row buffer policy. In the paper, an open-row private-bank has been considered to exploit memory fast access with row hits. The architecture is based around bank schedulers with input queues to schedule inter-bank requests and a channel scheduler to schedule commands from the bank scheduler to the memory. Analytical comparison with a state-of-the-art memory controller showed up to 27% better timing bounds. Furthermore, the researchers provide detailed timing latency bounds.

Another open-row private-bank Real-Time memory controller has been proposed by Wu et al. [10]. The controller does not optimise for bus-turnaround, which results in higher analytical bound than [3] for high-speed modules but comparable simulated bound. Besides a higher analytical bound, also the bus utilisation is lower as the penalty for a bus turnaround increases with faster modules.

Bhat et al. researched the impact of DRAM refresh on the predictability of task execution times on microprocessors [27]. The DRAM refresh is typical triggered by the memory controller and thus unsynchronized with the tasks on the processor. In their work, the asynchronous auto-refresh is replaced by a periodic software task that performs refresh operations in burst mode. This approach allows the refresh to be modelled as a process and thus delays due to refreshes can be isolated from application execution. Furthermore, memory requests during hardware refresh are prevented. The refresh time is relative small compared to task periods such that scheduling a single refresh complicates the static schedulability. The burst refresh, as described in Section 2.1.4, increases the refresh interval and thus reduces the number of task preemptions.

The memory controllers of *Ecco et al.* and *Wu et al.* are open-row real-time memory controllers that offer high performance when a known static access pattern can exploit internal caching of the DRAM. The latency of a sequence of requests can be derived from the number of row-misses and row-hits of the access pattern independent from requests to other banks. The memory controllers of *Heithecker et al.* and *Akesson et al.* use a close-row policy in which burst requests to all banks are performed. The refresh method of Bhat et al. can be integrated with the memory controller and using hardware signals to synchronize the tasks with the memory refresh. The burst refresh execution time is known such that it can be used in the scheduling of requests in each task.

3.2 Memory Architecture

The CoRAM memory architecture provides a virtualised memory through distributed SRAMs that are actively managed by programmable Control Threads [28]. The Control Threads manage data transfers between the CoRAM and external memory through a standardised *Instruction Set Architecture* (ISA) on behalf of the application logic. The application logic and CoRAMs exchange tokens through two-way FIFO channels to indicate start and result of a transfer. In contrast to the rest of the architecture, Control Threads can be expressed in higher level languages such as C, and allow sophisticated memory architectures such as scratchpads, caches, and FIFOs that are tailored to memory patterns. The architecture targets implementation in fabric and arranges CoRAMs in a 2D-mesh that are connected through a NoC. This allows the transfer of data between the CoRAM memory interfaces such that multiple can be composed to create larger memories.

In contrast to the explicit user control of data transfers between on-chip SRAMs and external memory in the CoRAM architecture, LEAP scratchpads abstract the memory architecture from the application [29]. The LEAP scratchpads framework utilises on-chip RAM as a single memory which is partitioned and managed by a single scratchpad controller. Each application client has a dedicated scratchpad interface that connects to the scratchpad controller through a ring interconnect. To reduce the latency of the shared, high-latency scratchpad network, each client has its dedicated cache and marshaler to merge objects from different sizes in a single memory. The private caches in the LEAP architecture are direct mapped caches, but the caches can be replaced by other cache implementations and even by CoRAMs.

Yang et al. improved the performance of the LEAP architecture by adding prefetching to the private caches and improving the scratchpad memory network [30, 31]. Some applications, such as H.264 decoding, have a data-dependent or complex memory access pattern such that it is difficult to derive a static access pattern at compile time. Yang et a. propose a dynamic stride-based prefetching mechanism that learns the distance between the addresses of requests and fetches data based on the distance.

Papenfuss et al. presented a platform for high-level synthesis for memory-intensive image processing algorithms [32]. The platform is based on sliding window buffers in RAM and autonomous prefetching caches. The window buffers are used for algorithms that traverse deterministically and references multiple lines concurrently. In the platform, the buffers are used between two processing algorithms in order to reduce the number of external memory accesses. A cache controller uses private bank-interleaving in combination with a round-robin arbiter which maps each cache module to a separate memory bank. The cache modules are specially targeted for image processing by writing pixels in-order and reading pixel tiles. The tiles move deterministically through the image and allow small non-deterministic accesses within the tile. The cache prefetches two tiles in RAM and has run-time configurable start, stride and tile size in which the stride determines the displacement of the window. For non-deterministic memory accesses, multiple caches are used as ping-pong buffers and swap after each image frame.

Ma et al. propose the use of direct-mapped private caches per application port to minimise cache conflicts between ports [33]. Cache privatisation also allows optimising the cache for a specific application such as read/write -only and associativity.

The effect of cache parameters such as associativity and dimensions on the speed and area of an Altera Stratix FPGA has been analysed by Yiannacouras et al. [12]. In their work, a background analysis of caches is presented with a review of different cache associativities and their effects on implementation. Furthermore, caches have been evaluated on the occupied area and speed with different associativity, latency, cache depth, address width, and data width.

The CoRAM architecture exploits memory access patterns of an application to manage distributed SRAMs. The *Control Threads* are programmable in high-level languages and make use of a fixed ISA. The NoC complicates the real-time analysis, and both the architecture and control logic can be defined in CλaSH to create a dedicated active managed scratchpad. LEAP scratchpads use a single scratchpad with ring interconnect to connect multiple clients through private caches. The high-latency LEAP architecture depends on caches to reduce request latency, and the number of requests on the ring interconnect. Based on the LEAP framework, a real-time ring interconnect such as in [34] can be used in combination with a close-row memory controller as a scratchpad to connects multiple clients to the external memory. The techniques of the sliding window buffers and cache strategies presented by Papenfuss et al. can be used to create deterministic prefetch buffers.

Chapter 4

Interface Design

Traditional caches improve the performance of memory requests by temporarily buffering memory data in faster memory and thus reduce the number of requests to the external memory. They are unaware of the access pattern of the application such that only previous requests can be used to predict future requests. In case that the memory data access pattern of the application is static and known at design-time, this knowledge can be used to create an actively managed buffer that deterministically buffers memory data. Additionally, multiple memory requests can be combined in burst sequences that can be efficiently transferred. In combination with the known behaviour of the memory controller, the worst-case latency of memory request sequences can be determined such that a deterministic schedule can be derived. The objective is to create a framework in CλaSH that can generate an optimised real-time memory interface for external SDRAM. The framework contains parametrised actively managed buffers that exploit the design-time knowledge of the memory access pattern of the application.

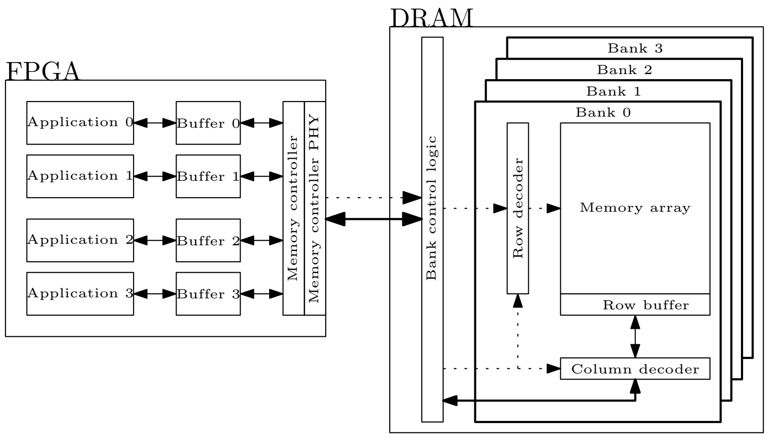


Figure 4.1: Schematic overview of FPGA with external DRAM

Previous chapters concentrated on the background information and leading research on DRAM, memory controllers and traditional caches, along with background information on CλaSH and available information in data access patterns. This chapter focusses on the design of a deterministic real-time memory interface that exploits the known data access pattern and the behaviour of the memory controller. A schematic overview of the FPGA within reconfigurable hardware, the memory controller and buffering, in combination with external DRAM is shown in Figure 4.1. The first part of this chapter focusses on the design decisions in the design of a memory architecture. The second part describes the design based on the choices of the *Design Space Exploration* (DSE). Finally, the design choices are summarised in the conclusion.

4.1 Design Space Exploration

Many FPGAs such as the FPGAs on the *Xilinx Artix-7 FPGA AC701 Evaluation Kit*, have a hardware PHY that embeds the necessary hardware to control DDRx memory device signals and thus can be used as a base for a memory controller. The remaining memory controller logic is implemented in reconfigurable hardware such that either the Xilinx generated *Memory Interface Generator* (MIG) or a modified/custom memory controller can be used. The row-policy of the memory controller has a significant impact on the type of memory access, and thus, the remaining memory interface architecture.

4.1.1 Memory Controller Policy

The memory controller manages the DRAM and translates memory requests in sequences of DRAM commands which are scheduled by a scheduler inside the memory controller. As DRAM is highly stateful, the order of commands has a significant influence on the performance of each memory request. COTS memory controllers are optimised for average throughput, with extensive memory re-ordering to schedule memory requests in the most efficient order. Consequently, less efficient requests are delayed in favour of more efficient requests, which results in pessimistic worst-case latencies of individual memory requests. Hence, real-time memory controllers use a strict order policy in which requests to a single bank are scheduled in the same order as the arrival.

The row-policy, as explained in Section 2.2, has a significant impact on the request latency and the type of memory access and the remaining memory architecture. A close-row policy requires a strategy that allows exclusive interleaved-bank accesses to all banks to guarantee address-independent memory access at a given time slot. Every request can be considered as a row-miss with a fixed latency bound independent from the previous request. Open-row memory controllers are more efficient when the caching behaviour of the row buffer can be exploited. A common access strategy is bank-privatisation that grants a client exclusive access to one or more banks. Another strategy is interleaved bank memory access but it requires multiple consecutive slots to benefit from the internal caching behaviour of the memory. A drawback of interleaved access is that a static deterministic schedule must be found that satisfies all clients. With the relative wide data bus of modern external DRAMs, a single burst request to all banks in a close-row policy transfers typically 512 bytes. An open-row policy benefits from requests to an open-row and consequently has a more gradually data transfer.

A trade-off between close and open-row with exclusive interleaved-bank and private-bank architecture is shown in Table 4.1. On FPGAs, the external memory is used purely for large application

Table 4.1: Memory controller policy trade-off

Row policy Access policy	Close-row Exclusive interleaved-bank	Open-row Exclusive interleaved-bank	Open-row Private-bank
Number of clients	+	+-	-
Access	Burst to all banks	Multiple burst to all banks	Burst to dedicated banks
Performance	-	-	+
Access schedule	TDMA	TDMA; multiple consecutive slots	Continuous burst sequences

data. An open-row memory controller with private-bank policy offers the highest potential performance as the row buffer can be exploited. Furthermore, bank-privatisation isolates memory requests to different banks such that memory access schedules of a client can be derived independently of other clients and thus simplifies the analysability of the overall system.

For the memory controller, the real-time open-row memory controllers for DDR3 of Ecco et al. [3] and Wu et al. [10] have been considered. The simulated performance found in [3] of both controllers is comparable, but the controller of Ecco et al. provides a lower analytical bound for DDR3 DRAM than the controller of Wu et al. As the analytical bound is used for scheduling in a real-time system, the controller of Ecco et al. has been chosen as it will provide higher performance. The Xilinx MIG generated memory controller features a close-row policy with strict re-ordering mode. The Verilog source code of the memory controller is available but requires reverse engineering to translate the memory controller to CλaSH. Furthermore, significant engineering effort is required to modify the controller to the scheduling implementation as described by Ecco et al. The implementation of an SDRAM controller is highly complex such that only a model will be used in CλaSH for analysing and simulation purpose. This model should behave as if realised on an FPGA.

4.1.2 Memory Controller Request Interconnect

The memory controller described by Ecco et al. is designed for a many-core system and features *Caching and Interconnect fabric* to connect each core to the memory controller, as shown in Figure 4.2. The *Bank Address Mapping* decodes the memory address and forwards the request to the matching *Bank Request Queue*. As a result, the interface of the memory controller is a shared resource and arbitration is required to provide guaranteed access. In a private-bank architecture, each bank has at most one client such that the it can be directly connected to its bank request queue, and the shared interconnect can be eliminated. Consequently, the *WRITE Data Buffer* has to be merged with the bank request queues as write data is part of the request. The *Channel Scheduler* can match the received data to the client such that the data can be routed accordingly. The data bus of the external memory is a shared bus such that a common data bus can be used for the received data. Each client can listen to a ready signal that indicates that its data is placed

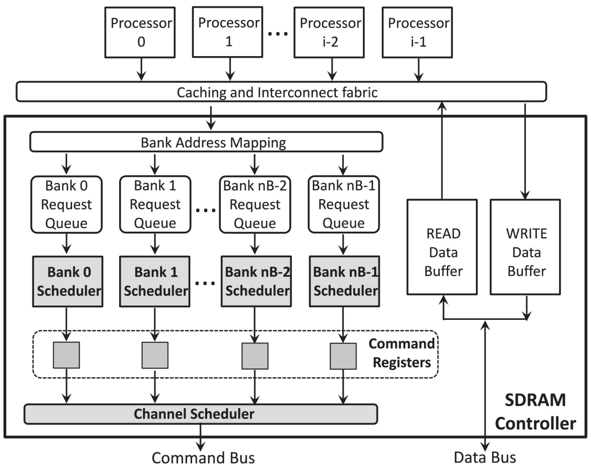


Figure 4.2: Memory controller by Ecco et al. [3]

on the bus and thus eliminate the *READ Data Buffer*.

Table 4.2: Buffer to memory controller connection

Hardwired connection	Crossbar Interconnect
Data exchange in application	Data exchange in memory
Private memory map	Shared memory map
Exclusive access fixed in architecture	Mutual exclusive access controlled by the application

The clients can be *hardwired* to the bank request queue or connected through a *crossbar interconnect*, as outlined in Table 4.2. With a hardwired connection, each buffer can shape its memory map, but data exchange with between buffers is performed in the application logic. With a crossbar interconnect, buffers can exchange data by reconfiguring the interconnect. Hence, a global memory map is required such that multiple banks can access the same data. As the banks are dedicated to the access pattern of the application and the storage is used accordingly, the hardwired connection is preferred. The crossbar can be easily added afterwards but requires a global memory map and control from the application.

4.1.3 Memory Refresh

The refresh mechanism has a significant influence on the static schedulability of requests. The open-row policy offers higher performance the longer the bank is open as the internal caching of the DRAM can be exploited. Besides memory unavailability, the refresh also closes the row buffers of all banks such that a row-hit to the previously open bank turns into a row-miss. Furthermore, an idle period between memory requests and the refresh can occur when a request takes longer than the time remaining before a refresh. As the refresh period is an average interval, an advanced or delayed refresh can improve the schedule for a single bank but inherently influences the schedule of other banks. This is less of a problem in a close-row interleaved-bank architecture as every request has the same length with no difference between row-hit and row-miss. An effective method in a private bank architecture to increase the performance is to use a periodic burst refresh. A burst refresh increases the continuous access period, but also increases the memory unavailability time. As the refresh is periodic and the buffers can synchronise with the memory controller, memory requests can be statically scheduled for an individual buffer. The burst length should be configurable such that the refresh interval can be configured at design time.

4.1.4 Buffer Data Storage

The external memory connected to reconfigurable hardware is purely used for application data. The data is mostly formatted such that it can be efficiently stored and transferred but is not always efficiently formatted for processing on an FPGA. Normal caches abstract the external memory and thus have a fixed memory architecture. Data access is based on memory requests such that only one memory address can be accessed at the same time. Accordingly, data that overlaps multiple memory addresses must be consecutively requested and stored locally. The actively managed buffers are not bound to the storage method of the cache such that the on-die memory of the FPGA can be used to store the data such that the application can efficiently access it for processing. A buffer can have multiple tasks with each a different access pattern and storage requirements such that each can have dedicated storage. Consequently, translation is required to translate between the local and external storage format. Translation can take multiple clock cycles such that additional FIFOs are required to temporarily store the data between translation and transfer to or from the external memory. The buffers are tailored to the data access pattern of the application and hence require higher development effort on the buffers but a lower effort on the application itself. Multiple tasks of the application must use mutual exclusivity to access the external memory data to avoid coherency conflicts as the translation adds extra latency. A task that reads data from the external memory must wait until the data is guaranteed to be stored.

Table 4.3: Memory architecture storage trade-off

Memory format	Application format
Translation in application	Translation in buffer
Simpler versatile buffer	Buffer tailored to the application
Possibly multiple clock cycles required to access data	Efficient data access for application
Data only stored in data storage	Buffering between translation required
Wide data format hard to store in BlockRAM when depth cannot be exploited	Smaller data format easier to store
Access based on the memory address	Data abstracted from address

As outlined in Table 4.3, the buffered memory data can be stored in memory format like a cache or in an application specific format that is efficient to process. Individual data units often have a different size than a single data transfer such that storage in application format is preferred. For example, modern DRAMs have a 64-bit wide data bus combined with a burst size of 8 transfers such that 512 bits are transferred for a single request.

4.1.5 Conclusion

An open-row real-time memory controller, in combination with bank-privatisation, has been chosen with a gradual data transfer. Efficient sequences can exploit the caching behaviour of the external memory with higher efficiency of the data transfer as a result. The bank-privatisation allocates dedicated banks to a buffer and thus isolates the behaviour of each bank. The shared interconnect of the selected memory controller is eliminated such that buffers can directly access the bank request queue(s). This avoids arbitration in the interconnect and thus simplifies the architecture. Consequently, transfer between buffers must be performed in application logic. The actively managed buffers store the data locally in a format that can be efficiently accessed by the application. This requires additional translation between the external memory and local memory but allows to use the on-die memory efficiently. When a buffer has multiple tasks, mutually exclusive access to memory regions inside the external memory must be used to avoid cache coherency conflicts.

4.2 Buffer Architecture Design

Based on the choices made in the design-space-exploration, this section describes the basic buffer architecture design.

The actively managed buffer is located between the memory controller and application, and exploits the design-time knowledge about the data access pattern to transfer data. It is synchronised with both the memory controller and application such that it can transfer the correct data. With the synchronisation and known access pattern, the buffer provides a virtual memory space to the application. Each buffer is tailored to the application but all buffers share a basic architecture, as shown in Figure 4.3. The architecture features a master-slave controller design to separate the management of application data and memory data in which the front controller is the master and the back controller the slave. A buffer can have multiple tasks with each a different access pattern and storage requirements. The buffer shown in the figure contains two tasks: a write task and read task, that can be used by separate tasks of the application.

The front controller manages the buffer storage and is synchronised with the memory controller and the application. The synchronisation with the application is performed through the tasks such that the application can access each task through its interface. After a refresh from the memory controller, it consecutively schedules for each task a request sequence which are encoded in the controller. In order to provide a virtual memory space to the application, it keeps track of the state of each task and the memory address of the external memory. The back controller performs the requests to the memory bank on behalf of the front controller and signals to the front controller when the sequence is finished. The buffer stores data in the application format and consequently requires application-specific translation between application data and memory data. Each task contains dedicated storage and translation, and contains a small FIFO to buffer between translation and memory transfer as translation can take multiple clock cycles. The FIFOs maintain the order of the translated data such that the back controller requires a start memory address and the number of requests in a sequence to initiate a burst data transfer to the memory. Depending on the

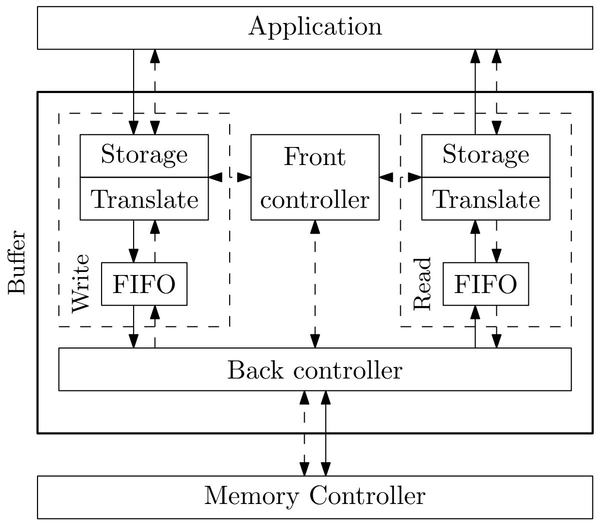


Figure 4.3: Basic buffer architecture with write and read tasks

direction of the transfer, the back controller reads data from the write FIFO or pushes data in the read FIFO. The memory regions of the tasks must be mutually exclusive to avoid data coherency issues as the on-die storage is separated.

Since the buffers are implemented on reconfigurable hardware and the buffer is designed in the same language as the rest of the system, the buffers are highly customizable and allow for tight integration with the application. Furthermore, the high-level abstraction of the CλaSH language allows for reuse and a high degree of automatic parametrisation through the type system. Each task of the buffer has its specific storage and translation, and the front controller contains logic to schedule the request sequences for each task. The back controller and FIFOs are generic such that they can be reused in different buffers.

Chapter 5

Realisation

In real-time systems, a deterministic schedule must exist that guarantees that a task finishes within a constraint time window. Therefore, it is crucial that the latency of memory requests is known such that the schedule can be guaranteed. The external DRAM is managed by a memory controller and subsequently requires a controller that is designed for real-time systems of which the latency of a memory request is bound. COTS memory controllers tend to be optimised for high bandwidth and re-order memory requests to schedule memory requests in the most efficient order with pessimistic worst-case latencies as a result.

In the previous chapter, a real-time memory controller is chosen that features an open-row policy in combination with a private bank architecture that limits the number of requesters of a memory bank to one requester. This architecture benefits consecutive requests to an open memory row such that row-misses can be eliminated. Thus, the worst-case latency of a known memory request sequence is presumably lower than the combined worst-case latency of an unknown requests. The memory buffer fetches memory data based on the memory access pattern of the algorithm such that it requests the correct data at the correct time. This buffer stores the data, preferably in a data format that is efficiently accessible by the application.

This chapter highlights the realisation of the memory controller model in CλaSH and compares simulations of the implemented model with a dataflow model. Furthermore, a buffer based on a 2D convolution algorithm for image processing is implemented as use case for the buffer mechanism.

5.1 Memory Controller

Memory controllers are highly sophisticated due to the high statefulness and different clock domains between logic on the FPGA and DRAM control and data bus signals. Therefore, only an abstract model of the memory controller, as described by Ecco et al. [3], is implemented such that the system can be simulated in CλaSH. The model contains a memory array for memory storage such that the data can be stored and retrieved.

5.1.1 CλaSH Model Behaviour

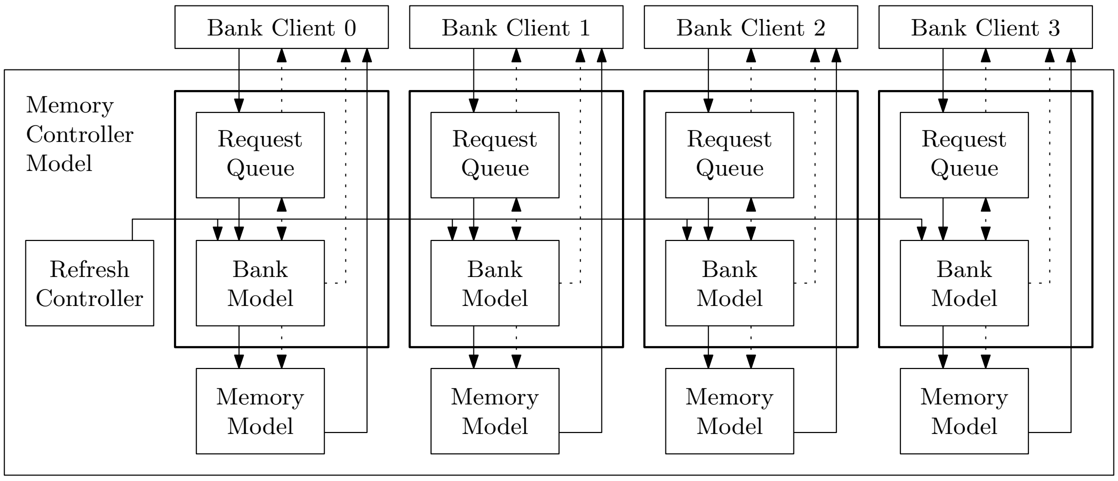


Figure 5.1: DRAM memory controller model with four banks

The model contains a bank-independent bank-model in combination with a private request-queue for each bank with a central controller for synchronised refreshes and initialisation, as schematically shown in Figure 5.1. As described Section 4.1.2 of the DSE, the shared interconnect of the memory controller proposed by Ecco et al. is replaced by hardwired connections with the clients.

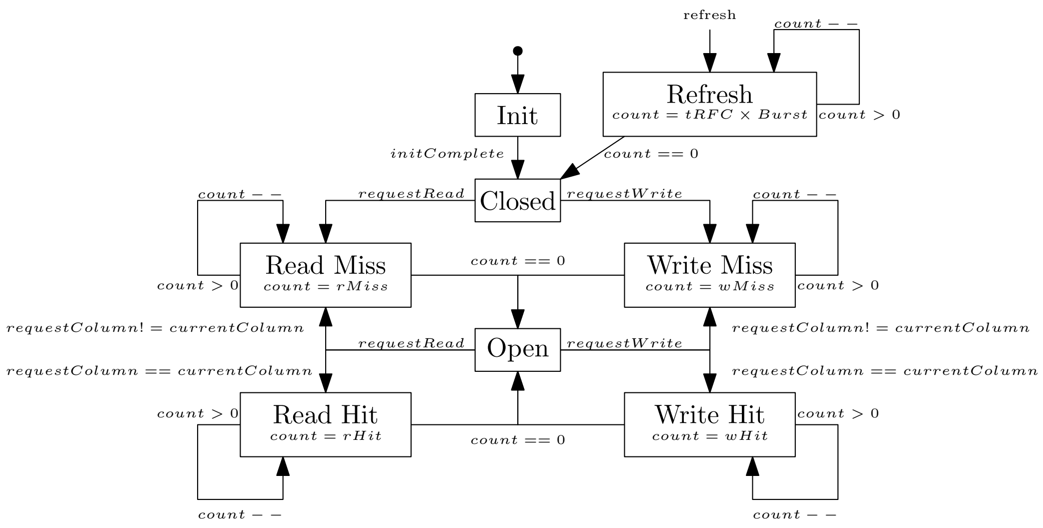


Figure 5.2: Memory bank state diagram with an asynchronous refresh

The bank model is based on a state machine for processing read and write-requests, which uses precomputed maximum latencies for row-hits and row-misses, as shown in Figure 5.2. The model starts in the *initialisation* state, which simulates the initialisation phase of the memory controller. After that, the row buffer is *closed*, and requests in the request-queue can be processed. After a request from the *closed* state, the row buffer remains open such that requests to the same row can be either a row-hit or miss. The model has a common refresh counter with refresh burst multiplier for all banks. On overflow of the counter, all banks asynchronously switch to the *refresh* state. When there are unfinished requests at the start of the refresh, the request is restarted when the refresh is finished. Consequently, all rows are closed such that the first request after a refresh is always a row-miss. When the refresh is finished, each bank signals a refresh ready such that the buffer can synchronise with the memory controller.

The model is connected to a 2D-matrix of rows and columns in which each element can store data from a single request such that data can be accessed in a single clock cycle. As the model uses the worst-case latency for each request, multiple banks can receive data at the same time. Therefore, each bank requires a data receive bus while in reality, a common data bus is sufficient as the DRAM has a single data bus. Request latencies are often less than the worst-case such that requests sequences can finish earlier than the worst-case estimation. Designers must accommodate sufficient buffers to prevent stalling of the burst access. The high abstraction level allows modelling other real-time memory controllers with known latencies.

5.1.2 Memory Request Timing

With a known static memory access pattern and the known behaviour of the memory controller, sequences of memory requests can be statically scheduled. The chosen memory controller, as described by Ecco et al. in [3], features an open-row policy such that the worst-case latency of a row-miss request is significantly larger than a row-hit request. Because of the known data access pattern, it is possible to determine the number of row-hits and row-misses within a sequence of requests during design time.

With the method described in the paper, the read and write latencies for a row-miss and row-hit are calculated for the *DRR3-1066F* standard. This standard is the fastest possible on the Xilinx AC701 evaluation board [35] as the bus frequency of the FPGA is limited to 533 MHz. The resulting latencies are shown in Table 5.1. With the known latency of a request and the known sequence of requests, the ideal maximum latency of a task without refresh can be calculated as the sum of the products of each request type and the number of occurrences of that type in the sequence. The task latency is formalised in Equation 5.1.

Table 5.1: Request latency in clock cycles for DDR3-1066F at data bus frequency

Notation	Description	Command sequence	Latency (clock cycles)
L_{Req}^{RM}	Read Miss	Precharge-Activate-Read	134
L_{Req}^{RH}	Read Hit	Read	75
L_{Req}^{WM}	Write Miss	Precharge-Activate-Write	134
L_{Req}^{WH}	Write Hit	Write	75

$$L_{Task}^{Reqs} = N_{Task}^{RM} L_{Req}^{RM} + N_{Task}^{RH} L_{Req}^{RH} + N_{Task}^{WM} L_{Req}^{WM} + N_{Task}^{WH} L_{Req}^{WH} \tag{5.1}$$

Although the DRAM data bus has a frequency of 533 MHz, most designs for reconfigurable hardware use a lower frequency such as 50 MHz or 100 MHz. With the assumption that the embedded memory controller runs internally at a higher frequency and no additional latencies are introduced, the latencies found in Table 5.1 are linearly scaled to a frequency of 100 MHz. When the memory controller has additional latencies, they can be added to the calculated ones. Furthermore, as the read and write latencies are equal for the used bus standard, the latencies can be reduced to a hit and a miss latency. The hit and miss latency at 100 MHz are 26 and 15 clock cycles, respectively. With combined read and write latency, also the equation for the task latency can be simplified, as shown in Equation 5.2. Similarly, the refresh interval and refresh cycle count can be scaled. At a 100 MHz clock, the refresh interval is 780 clock cycles with a refresh count of 12 clock cycles. The scaled timing parameters are summarised in Table 5.2.

Table 5.2: Scaled timing parameters for a 100 MHz clock

Parameter	Clock cycles
${L_{Req}^{M}}'\ (t_M)$	26
${L_{Req}^{H}}'\ (t_H)$	15
${t_{RFC}}'$	12
${t_{REFI}}'$	780

$$ {L_{Task}^{Reqs}}' = N_{Task}^{M} {L_{Req}^{M}}' + N_{Task}^{H} {L_{Req}^{H}}' \tag{5.2} $$

5.1.3 CλaSH Model Simulation

To compare the CλaSH implementation with the expected behaviour, the CλaSH memory controller has been simulated with several requests that highlight various situations. Compared to the timings found in subsection 5.1.2, most of the timings used in the simulation, as shown in Table 5.3, have been reduced to improve the readability of the simulation result. The row-miss (t_M) and row-hit (t_H) access times have been reduced to 7 and 4 clock cycles, respectively. Accordingly, the refresh has been configured with a refresh period t_{REFI} of 15 clock cycles and a refresh time t_{RFC} of 2 clock cycles. A refresh multiplier of two has been used that effectively increases the period to 30 clock cycles and refresh time of 4 clock cycles. The initialisation time has been set to one clock cycle as the model requires no initialisation. The queue size has been configured with a size of 3.

Table 5.3: Simulation parameters

Parameter	Value	Units
rows	64	
columns	32	
data-width	8	bits
t_{REFI}	15	clock cycles
t_{RFC}	2	clock cycles
refMultiplier	2	
t_M	7	clock cycles
t_H	4	clock cycles
initDelay	1	clock cycles
queue depth	3	

The simulation is divided into two situations. The first situation (Listing 5.1) shows the typical start-up followed by two write requests and corresponding read requests. The second situation

(Listing 5.2) shows the memory refresh in combination with a write request that is postponed until after the refresh. Each simulation result has four primary columns; the clock cycle count (*CLK*), the bank request (*Bank Request*), the bank response (*Bank Response*) and the internal state of the bank (*Bank State*). Each row of the table represents the time in clock cycles.

```
CLK | Bank Request  | Bank Response                 | Bank State
    |               | Init Refresh Sync Ready Data   | State   Row Queue Request
--- | ------------- | ------------------------------ | --------------------------
0   |               | BUSY                    0x0000 | Init
1   | W 0x00 0x0001 |              SYNC       0x0000 | Closed
2   | W 0x01 0x0002 |                         0x0000 | Closed      Read
3   |               |                         0x0000 | Busy    0         W( 0, 0)
4   |               |                         0x0000 | Busy    0         W( 0, 0)
5   |               |                         0x0000 | Busy    0         W( 0, 0)
6   |               |                         0x0000 | Busy    0         W( 0, 0)
7   |               |                         0x0000 | Busy    0         W( 0, 0)
8   |               |                         0x0000 | Busy    0         W( 0, 0)
9   |               |                         0x0000 | Busy    0   Read  W( 0, 0)
10  |               |                   READY 0x0000 | Busy    0         W( 0, 1)
11  |               |                         0x0000 | Busy    0         W( 0, 1)
12  |               |                         0x0000 | Busy    0         W( 0, 1)
13  |               |                         0x0000 | Busy    0         W( 0, 1)
14  | R 0x01        |                   READY 0x0000 | Busy    0
15  | R 0x00        |                         0x0000 | Idle    0   Read
16  |               |                         0x0000 | Busy    0         R( 0, 1)
17  |               |                         0x0000 | Busy    0         R( 0, 1)
18  |               |                         0x0000 | Busy    0         R( 0, 1)
19  |               |                         0x0000 | Busy    0   Read  R( 0, 1)
20  |               |                   READY 0x0002 | Busy    0         R( 0, 0)
21  |               |                         0x0000 | Busy    0         R( 0, 0)
22  |               |                         0x0000 | Busy    0         R( 0, 0)
23  |               |                         0x0000 | Busy    0         R( 0, 0)
24  |               |                   READY 0x0001 | Busy    0
```

Listing 5.1: Memory controller simulation - Write Read

In the first situation, as shown in Listing 5.1, the memory controller is first initialised after which write and read requests are performed. The initialisation takes one clock cycle as indicated by *BUSY* in the bank response column. After that, the memory controller signals a *Sync* at $t = 1$ such that the client can synchronise with the refresh. At $t = 1$, the client inserts a write request to address 0 (value 1) in the request queue, followed by another write request to address 1 (value 2) at $t = 2$. One clock cycle after the insertion of the first request, a queue read is initialised (indicated by the *Read* in the *Queue* column) with arrival of the request data one clock cycle later. On arrival of the request data from the queue, it is translated to a memory request, as shown in the *Request* column of the bank state. The first write request accesses address 0 and is translated to a write request to row 0 and column 0. The bank is closed before the first request such that the first request is a row-miss and thus completes at $t = 10$, as indicated by *READY* in bank response column. The second request accesses the already open row such that a row-hit (t_H) is expected. Furthermore, it can be seen that the second request is retrieved during the first request.

Accordingly, the second write request is finished at $t = 14$ which is $t_H = 4$ clock cycles after the first request finishes. At completion of the last write request ($t = 14$), a read request to address 1 is inserted in the queue, followed by a read request to address 0. Like the first write request, the first read request experiences additional delay caused by the retrieval of the request from the queue, but experiences only $t_H = 4$ clock cycles latency due to the row-hit. The second read request finishes $t_H = 4$ cycles after the first request is finished as expected. The data received by the read requests at $t = 20$ and $t = 24$ match the data stored by the write requests.

```
CLK | Bank Request  | Bank Response                  | Bank State
    |               | Init Refresh Sync Ready Data   | State   Row Queue Request
--- | ------------- | ------------------------------ | --------------------------
25  | W 0x00 0x0001 |                         0x0000 | Idle    0
26  |               |                         0x0000 | Idle    0   Read
27  |               |                         0x0000 | Refresh 0         W( 0, 0)
28  |               |      REFRESH            0x0000 | Refresh           W( 0, 0)
29  |               |      REFRESH            0x0000 | Refresh           W( 0, 0)
30  |               |      REFRESH            0x0000 | Refresh           W( 0, 0)
31  |               |      REFRESH SYNC       0x0000 | Busy              W( 0, 0)
32  |               |                         0x0000 | Busy    0         W( 0, 0)
33  |               |                         0x0000 | Busy    0         W( 0, 0)
34  |               |                         0x0000 | Busy    0         W( 0, 0)
35  |               |                         0x0000 | Busy    0         W( 0, 0)
36  |               |                         0x0000 | Busy    0         W( 0, 0)
37  |               |                         0x0000 | Busy    0         W( 0, 0)
38  |               |                   READY 0x0000 | Busy    0
```

Listing 5.2: Memory controller simulation - Refresh

The second situation, as shown in Listing 5.2, highlights the memory refresh. At $t = 25$, a write command to address 0 is inserted to the request queue. The request targets the already open row, but does not finish before the start of the refresh at $t = 28$. The refresh is finished at $t = 35$ as expected with a t_{RFC} of two and a refresh multiplier of two and thus an effective t_{RFC} of four clock cycles. After the refresh, the row is closed such that the request experiences a memory access latency of $t_M = 7$ clock cycles after the refresh is finished.

```
CLK | Queue
    | Write   Full | Empy Read Data
--- | -----------------------------
1   |              | E         0x0000
2   | 0x0001       | E         0x0000
3   |              |      R    0x0000
4   |              | E         0x0001
5   | 0x0002       | E         0x0000
6   | 0x0003       |           0x0000
7   | 0x0004       |           0x0002
8   |         F    |           0x0002
9   |         F    |      R    0x0002
10  |              |      R    0x0002
11  |              |           0x0003
```

Listing 5.3: Queue simulation

In the simulation, the arrival of the first request at the bank controller is two clock cycles after the insertion of the request. This behaviour is highlighted in a simulation of the queue, as shown in Listing 5.3. In the simulation, data is inserted in the queue at $t = 2$. At $t = 3$, the empty flag signals the non empty state such that a read can be started. At $t = 4$, the data inserted at $t = 2$ is received. In contrast to most modules which are based on a *Mealy* state machine, the queue is based on a *Moore* state machine in which the output is derived from the current state. Thus, a state change is visible in the next clock cycle. This includes status flags, such that the non empty state is visible a clock cycle delayed. When the empty flag is updated in the same cycle as insertion, a combinatorial loop can occur when a module reads the queue directly. In that case, the empty flag will be updated accordingly, and the module will stop reading and thus creates a loop. Xilinx user guide *UG743* describes the use of dedicated logic embedded in BlockRAM that enables the implementation of synchronous FIFOs [36]. Besides using dedicated storage in BlockRAM rather than using logic blocks, the FIFO supports the *First Word Fall Through* (FWFT) operation mode. In this mode, the first item inserted in an empty queue appears automatically at the output of the FIFO. A drawback of the BlockRAM based FIFO is the deassertion latency of three clock cycles such that data is available one clock cycle later than the current implementation.

5.1.4 Dataflow Model

The behaviour of the memory controller depends on the access pattern and thus the dataflow graph would be data depended. With a known access pattern and the assumption that a request sequence always starts with a row miss, the memory controller access can be modelled using a CSDF graph. Furthermore, it is important that the memory controller access is synchronised with the refresh and no access is performed during a refresh.

The memory controller is schematically depicted in Figure 5.3 for an expected pattern of four consecutive requests to the same memory row. In the diagram, node S executes four consecutive requests which are buffered in the $Queue$. The buffer has N_M places and node S has to wait with inserting an item in the buffer until a slot is available. When a request is available in the queue,

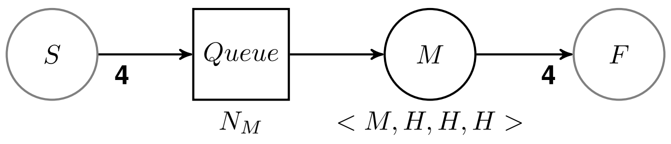


Figure 5.3: Schematic overview of the memory controller using four consecutive requests to the same row

node M fetches the request and executes it. As the requests are access the same row consecutively, the first request is expected to be a row miss as it cannot be guaranteed that the row is already open and the remaining requests will be a row hit. Node F fires when all four requests are finished.

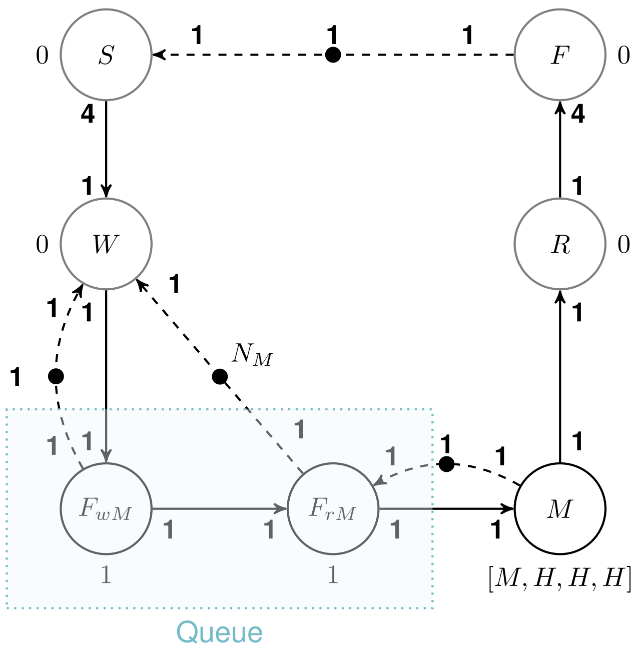


Figure 5.4: Unoptimized Memory Controller CSDF graph using a sequence of four requests

The schematic diagram shown in Figure 5.3 has been elaborated in the CSDF graph as shown in Figure 5.4. Like the schematic diagram, Node S generates a sequence of four consecutive requests to the same row. Those requests are received by helper node W which writes the requests one-by-one to the queue. A write to the queue takes one clock cycle in hardware and is represented by node F_{wM}. Like writing to the queue, also reading takes one clock cycle for the data to be available and is represented by node F_{rM}. The requests are thus stored on the edge from node F_{wM} to node F_{rM}. The edge from node F_{wM} to node W prevents multiple writes at the same time and the edge from node F_{rM} to F_{wM} limits the number of requests that can be stored in the queue to N_M requests. Node M represents the memory request being processed by the memory controller and node R fires each time a request is finished. When all four requests are finished, node F fires, upon which node S can start again and repeat the sequence. The edge from node F to node S prevents that a new sequence is started before the current sequence is finished.

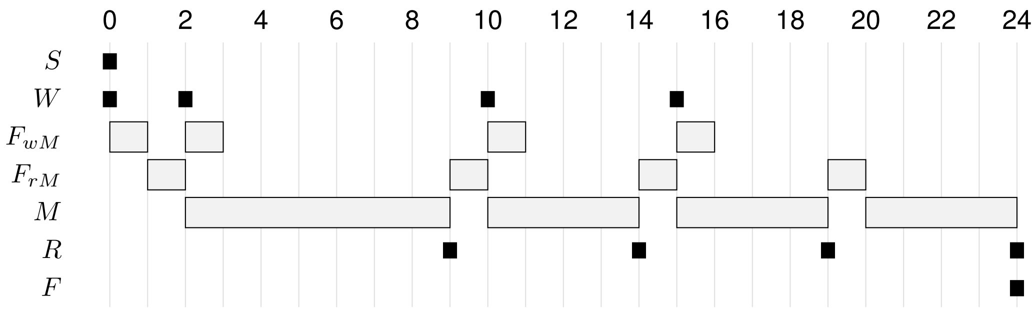


Figure 5.5: Unoptimized Memory Controller self-timed schedule ($N_M = 1; t_M = 7; t_H = 4$)

The firings of the nodes is clarified by the self-timed schedule, as shown in Figure 5.5, using a queue size of 1; a miss latency of 7 and a hit latency of 4 clock cycles. In the schedule, it can be seen that node S generates the requests at $t = 0$. The first request can be directly inserted in the buffer as the buffer is empty. At $t = 1$, the first request is written to the buffer and thus the buffer is non-empty. The request is directly fetched from the buffer, as seen by the firing of node F_{rM} at $t = 1$, upon which it can be executed by the memory. At $t = 2$, the request arrives at node M and takes 7 clock cycles to finish as the first request is a row miss. Since the request has been read from the queue at $t = 2$ and thus the queue is non-full, the second request can be stored in the queue as seen by the firings of nodes W and F_{wM}. The first request is finished at $t = 9$ upon which node R fires and also the second request is fetched by the buffer as seen by the firing of node F_{rM}. The second request is a row hit and thus the execution time of the second firing of node M is four clock cycles. The third and fourth requests are performed the same as the second request. At $t = 21$, all four requests are finished and node F fires.

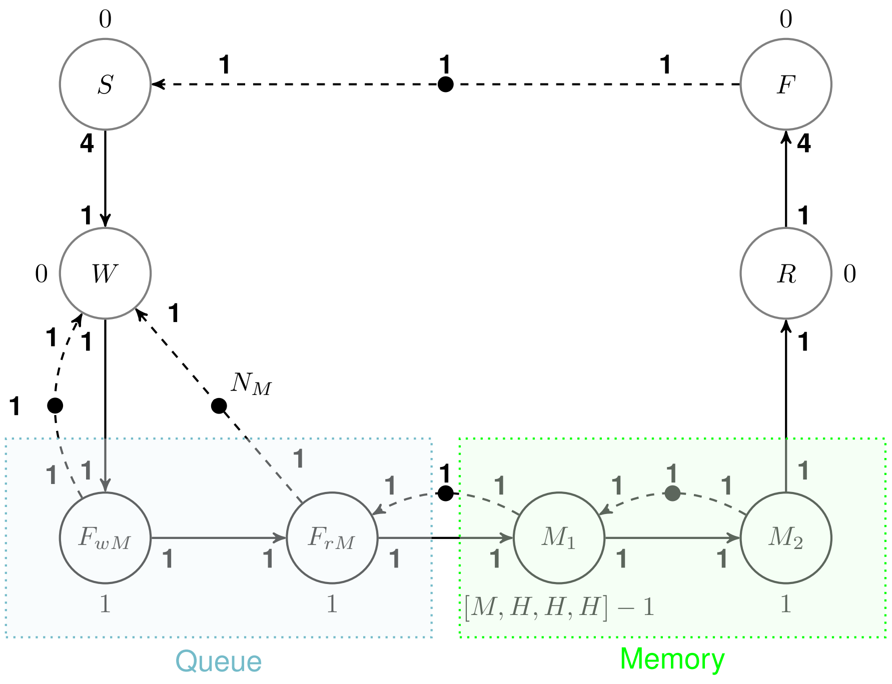


Figure 5.6: Memory-controller CSDF graph using a sequence of four requests

In the schedule, it can be seen that the execution of the memory request is delayed due to the fetch of the request from the buffer. By using two nodes instead of one to represent the memory, as shown in Figure 5.6, this one clock cycle delay can be hidden. Node M_1 represents the memory access latency, except it finishes one clock cycle before the end of a memory request, with node M_2 representing the last clock cycle of the request. Due to the edge from node M_1 to M_2, node M_2 can only fire after node M_1 has finished while the edge from node M_2 to M_1 prevents that the next request start before the current request is finished. Since node next node M_1 finishes one clock cycle earlier, the read of the next requests from the queue starts one cycle before the finish time of the current request and thus the next request is fetched when the current request is finished. The effect of this change can be seen in the self-timed schedule in Figure 5.7. In the schedule, M' represents the combined execution time of M_1 and M_2. Furthermore, it can be seen that the fetch from the queue (represented by F_{rM}) is finished at the same time the request is finished and thus the next memory request starts subsequently to the preceding request.

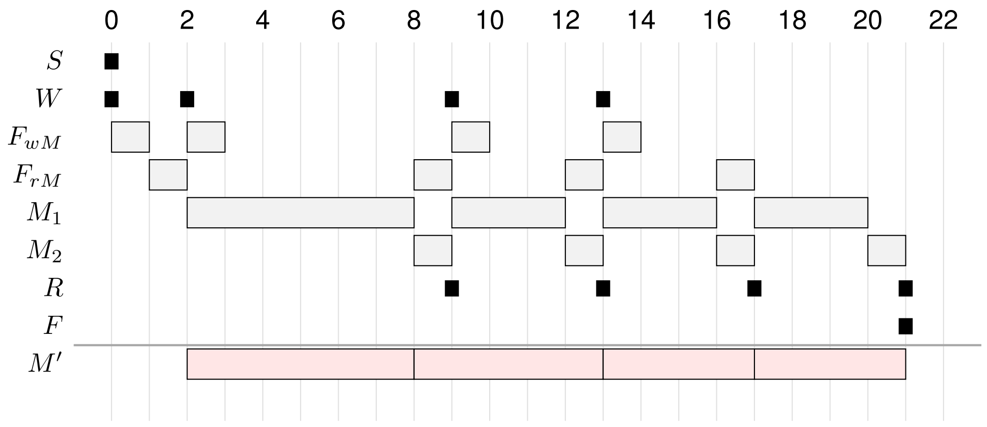


Figure 5.7: Memory Controller self-timed schedule ($N_M = 1; t_M = 7; t_H = 4$)

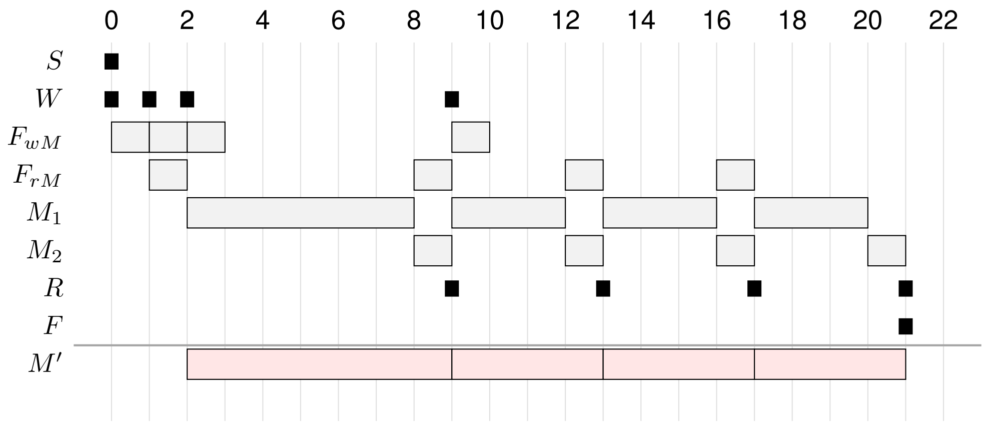


Figure 5.8: Memory Controller self-timed schedule ($N_M = 2; t_M = 7; t_H = 4$)

The effect of the request queue size on the schedule is shown in Figure 5.8. Compared to the schedule in Figure 5.7, the queue has a size of two instead of one. After the first request has been inserted in the buffer the queue has one more empty place such that the second request can be directly inserted after the first. At $t = 2$, two requests have been stored in the buffer, but also one request has been fetched. Therefore, there is one place free upon which the third request is written to the queue. The fourth and last request is written to the queue at $t = 9$. The larger queue size has no effect on the execution of the memory requests, but allows the requester to create more requests in advance.

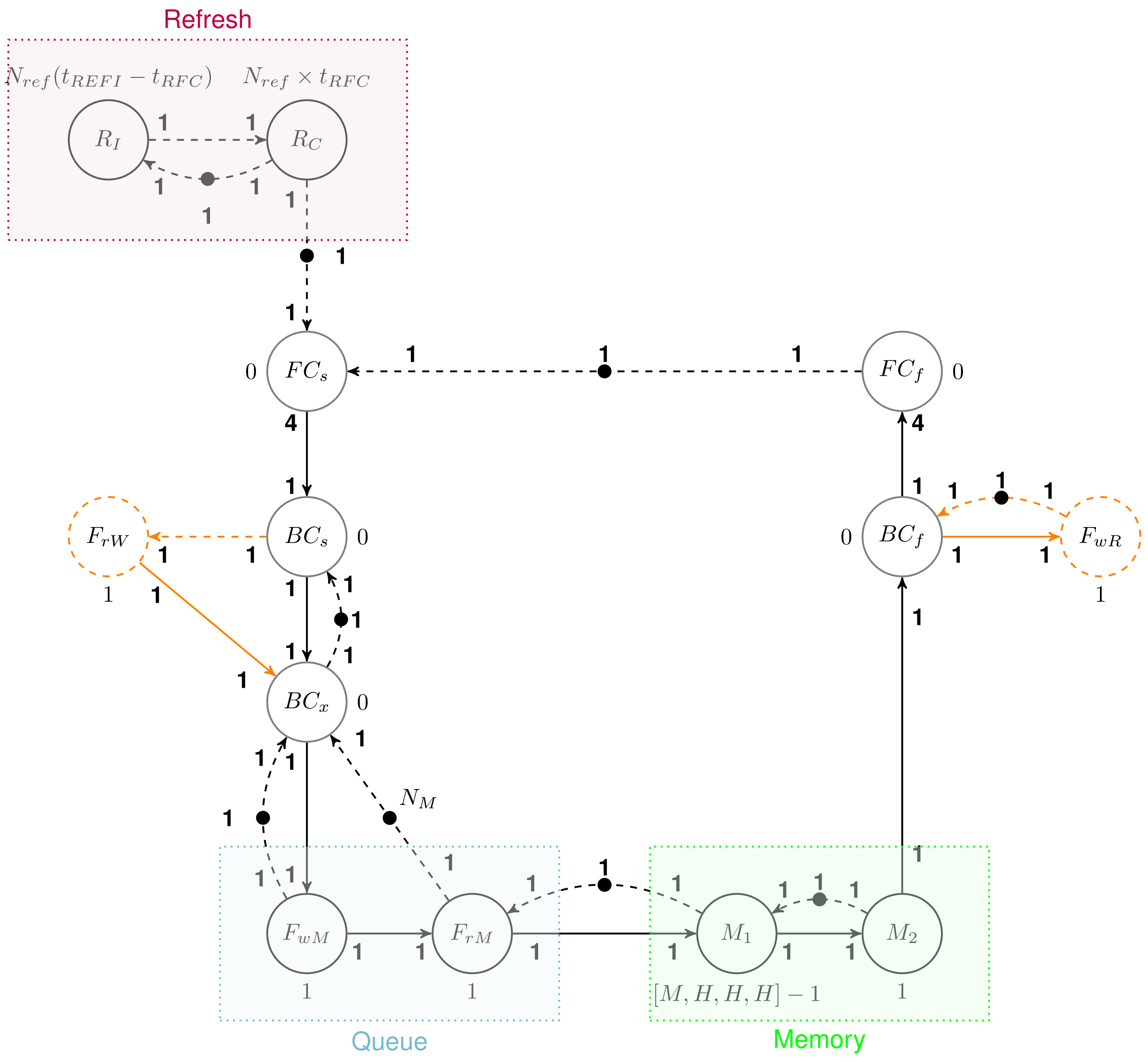


Figure 5.9: Buffer architecture CSDF graph using a sequence of four requests

The memory controller diagram in the CSDF dataflow graph in Figure 5.6 has been extended with the Front-Controller, Back-Controller and Refresh, as shown in Figure 5.9. The refresh is integrated in the memory controller, but it is assumed that the synchronisation is used between Front-Controller and Memory Controller and thus the refresh is modelled as such. The refresh contains node R_I that represents the interval between two refreshes and node R_C that represents the memory unavailability due to refresh. When the memory controller is initialised, there is no data yet in the memory, such that the Back-Controller can start directly without the need of a refresh. The edge from node R_C to node R_I enforces that both nodes are executed alternately such that there is a delay between two refreshes. Helper 'nodes' S and F in Figure 5.6 have been replaced by Front Controller nodes (FC) and nodes W and R have been replaced by Back Controller nodes (BC). Furthermore, the two nodes F_{rW} and F_{wR} represent the write data buffer read and read data buffer write respectively. If the requests are write requests, F_{wR} should be ignored and vice versa. FC_s generates a sequence of memory requests when enabled that are send to BC_s. Node W is replaced by two nodes BC_s and BC_x which are mutual exclusive. When a write request is issued, BC_s triggers a read from the write data buffer. BC_x waits for this data and forwards te request together with the data to the buffer of the memory controller. On the other hand, when the data is a read request, the data received by BC_f from the memory controller is written to the receive buffer represented by F_{wR}. Like the request buffer of the memory controller, both the read and write buffer require a one clock cycle to process the data. When writing to the memory, this causes the memory request to be delayed one clock cycle as node BC_x must wait for the data before the request can be inserted in the request buffer of the memory controller. When reading from the memory, the memory request scheduling is not affected as long as the read buffer has space available. A scheduling read example with two requests is shown in Figure 5.10. The memory refresh starts at $t = n - 4$ and is finished at $t = n$, at which also the front controller starts the request sequence. Since the request can be directly stored in the queue, the next request can start parallel. The write example, shown in Figure 5.10, shows that the read from the data buffer causes the memory request to start one clock cycle later. For the second request this is no longer a problem as the data is fetched during the first request.

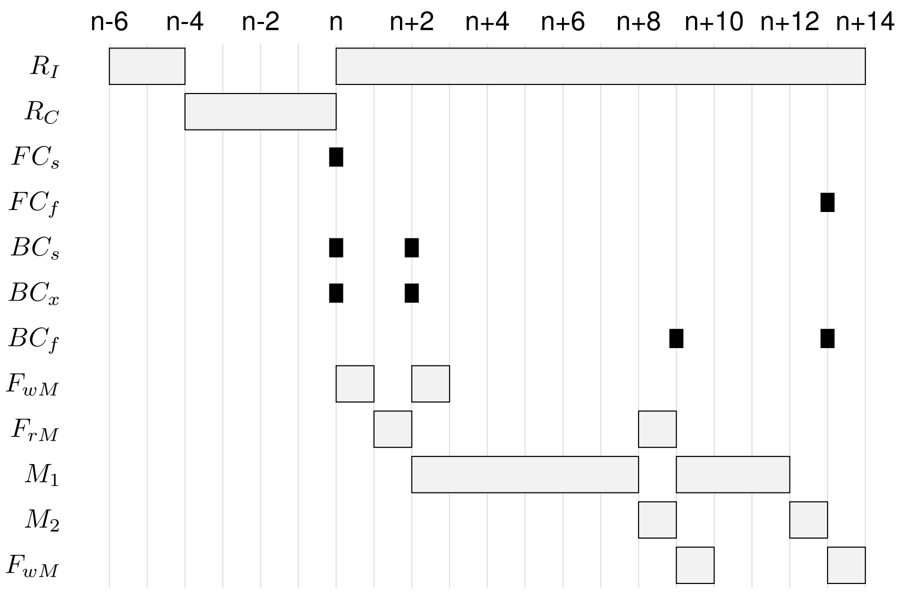


Figure 5.10: Basic buffer architecture with two read requests

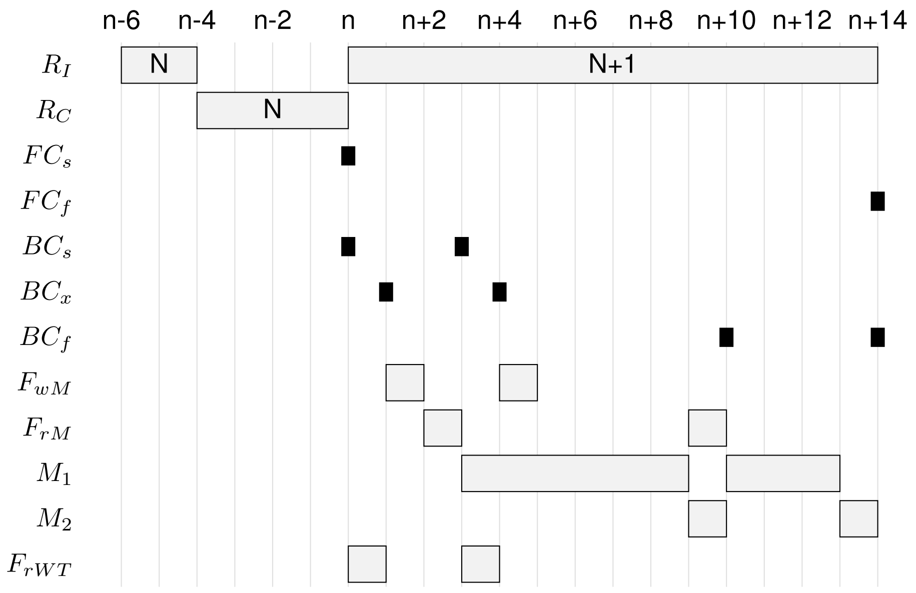


Figure 5.11: Basic buffer architecture with two write requests

5.1.5 Conclusion

Based on the memory controller chosen in the previous chapter, an abstract CλaSH model is implemented and simulated. This model uses the worst-case access latencies for several access types of the found memory controller as access latencies for row hit and row miss latencies. When requests are not interrupted by memory refreshes, the CλaSH model can be described by a CSDF dataflow model as in Figure 5.6 using the row hit-miss pattern. Based on the memory controller dataflow model, a dataflow model has been described that models the buffer architecture. Depending on a read or write access, the anchor points are shown where data request buffers can be connected to. When a write sequence is performed, the memory request is one clock cycle delayed due to the read from the buffer.

5.2 Use Case

To demonstrate the actively managed buffer, a use case is implemented based on the 2D convolution used in image processing. This section starts with a description of the access pattern followed by a description of the request sequence. Thereafter, a CλaSH model is implemented and simulated. The implementation is then compared to a dataflow model of the buffer.

5.2.1 Access Pattern

The image processing use-case assumes a source, such as a camera, that linearly streams an image to the external memory such that the 2D convolution algorithm can read it at another time. The image is stored as a bitmap in the external memory such that various patterns can be used to access the data. The stream write task and 2D convolution read task operate semi-parallel by consecutive burst requests to minimise row-misses within a sequence without the need to buffer complete images.

The linear storage of images allows chunks of pixels to be directly stored in the memory. To collect chunks of pixels, the translator of the write task must ‘deserialise’ the stream of pixels and forward the collected chunks to the buffer’s FIFO. The collected chunks in the FIFO can be transferred in a burst sequence and act consequently as storage for the write task. Deserialisation is based on *Serial-in Parallel-out* shift registers in which pixels are shifted as a single data unit. When sufficient pixels are loaded in the registers, all registers are parallel read as a single bit vector and forwarded to the FIFO.

The convolution algorithm is based on a sliding window that traverses an image row by row. The pixel data from adjacent image rows and columns are required to calculate the value of a single pixel within a window. As images are stored as bitmap linearly in the memory, data from neighbouring pixels in the same row is either at the same memory address or an adjacent address. Pixels on multiple image rows in the same column are further distributed in the memory such that multiple image rows have to be stored in memory for fast access. Image rows can be stored in individual line buffers such that pixels at the same column address can be accessed in parallel. With a kernel size K, a minimum of $K+1$ line buffers is required such that the application can process K rows and another line can be prefetched while the application is busy. As pixel data is small and an image row is deep, the on-die BlockRAMs are particularly suitable for storage of lines efficiently. When the application finishes processing a row, and the next row is fetched, the lines have to be shifted one row. As the BlockRAMs have limited access ports, the BlockRAMs can be virtually shifted by reconnecting the BlockRAMs in a circular fashion, without the need to copy data between BlockRAMs.

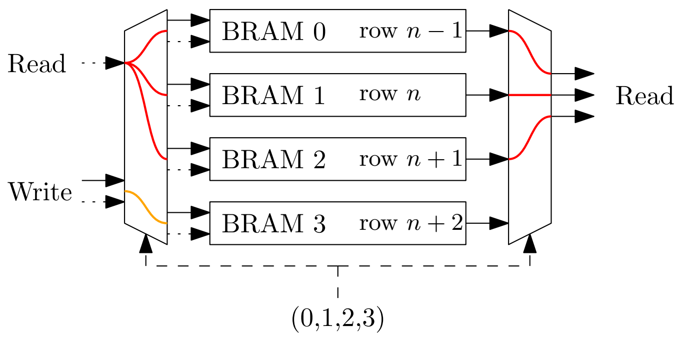


Figure 5.12: Line buffer initial state for kernel size of 3. Read (Red); Write (Orange)

The virtual rotation of BlockRAMs in the initial state and virtually rotated one row for a kernel size of $K = 3$ is shown in Figure 5.12 and Figure 5.13, respectively. The application reads K buffers at the same address in parallel such that the control signals for the line buffers being read can be replicated. However, the outputs of the buffers need to be demultiplexed in the correct order. The buffer being used for fetching data from memory requires control and data inputs, but the output data is irrelevant. The inputs require a multiplexer to switch between read or write, and the outputs

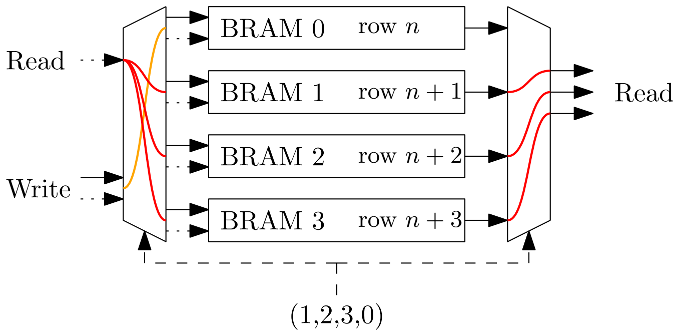


Figure 5.13: Line buffer virtual rotated one row for kernel size of 3. Read (Red); Write (Orange)

require a crossbar with 4 inputs and 3 outputs. In the initial state as shown in Figure 5.12, BlockRAMs 0-2 are connected to the read control and demultiplexer's outputs and the bottom BlockRAM to the write control and data input. In Figure 5.13, the line buffers are virtually rotated one position such that the write controls are connected to the top BlockRAM and the read controls and data to BlockRAMs 1-3.

The pixels received from the external memory are received in chunks such that the read translator requires to serialise the chunks in a stream of pixels. Serialisation is based on *Parallel-in Serial-out* shift registers in which the memory data from the FIFO is parallel loaded in registers and shifted out in individual pixels. For N pixels in a memory data unit, serialising received data to pixels requires N clock cycles. As the BlockRAMs on the FPGA are *True Dual Port* BlockRAMs, the translation can be accelerated by utilising both ports of the BlockRAM. Furthermore, the pixels need to be serialising in groups of two such that serialising takes $N/2$ clock cycles. Serialisation can also be accelerated by storing $stride$ number of pixels at an address such that $N/stride$ clock cycles are required. The data width increases with stride times the pixel size while the required depth is divided by the stride. Besides increased translation, the stride factor can be used to increase the processing performance as multiple windows are overlapped in the data.

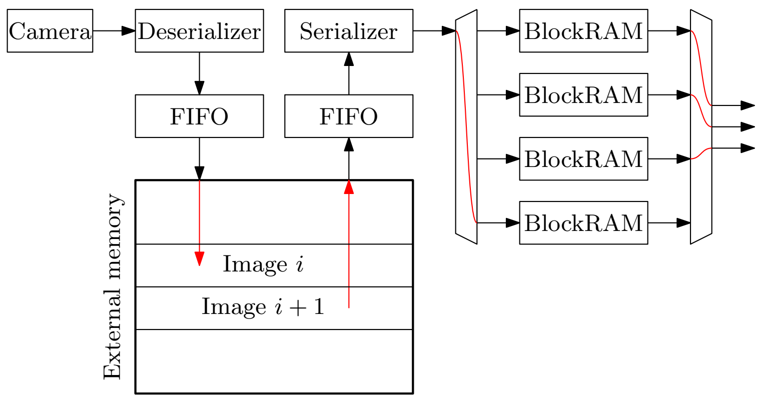


Figure 5.14: Image convolution buffer data flow

The data flow in the buffer is schematically shown in Figure 5.14. The data from the image source is streamed linearly to the external memory. The data can be linearly read from memory, but multiple image rows need to be stored in the memory for processing.

5.2.2 Request Sequences

With the access pattern as described above and information about memory structure, it can be determined how many row-misses and row-hits occur within a request sequence. An 1 GB DDR3 SODIMM module has eight DRAM chips with each an 8-bit data bus for a combined data bus of 64-bits. Each memory bank has 16 384 rows and 128 addressable columns. Each memory location contains 64 bits wide data which is accessed in 8 consecutive requests. Consecutively, the eight chips have together 8 192 bytes per row and transfer 512 bits per request. With an 8-bits pixel size, each transfer contains 64 pixels. Since images are stored linearly, an image row can be transferred with a sequence of consecutive requests.

Table 5.4: Request latency of image formats

Standard	QVGA	VGA	HD	FHD	Custom				
Columns	320	640	1280	1920	128	256	512	1024	2048
Rows	240	480	720	1080	128	256	512	1024	2048
Requests per image row	5	10	20	30	2	4	8	16	32
Image rows per memory row	25.6	12.8	6.4	$4\frac{4}{15}$	64	32	16	8	4
Misses	2	2	2	2	1	1	1	1	1
Hits	3	8	18	28	1	3	7	15	31
Image row latency	97	172	322	472	41	71	131	251	491

The number of requests to access an image row for several image sizes is shown in Table 5.4. The table shows the number of image rows that can be stored on a single memory row. As a buffer can have multiple tasks that operate semi-parallel, it cannot be guaranteed that both tasks operate in the same memory row. Furthermore, the row buffer is closed after a refresh such that the first request in a sequence is assumed to be a row-miss. It can be seen that for conventional image sizes, the amount of image rows that can be stored in a single memory row is a fraction. This means that there exist image rows that are distributed over multiple memory rows and thus the worst-case access contains an extra row miss. The custom square images have an integer number of image rows per memory row and thus do not contain an extra row miss. Using the number of hits and misses, the *Image row request latency* is calculated using Equation 5.2 and the request latencies as calculated earlier. In a streaming application, it can be beneficial to divide a sequence in multiple subsequences such that tasks can alternate faster and potentially reduce the required buffer size. For example, an image row of 1024 pixels can be transferred as one single sequence of 16 requests or two sequences of 8 requests each. Due to the row-miss at the beginning of a sequence, the combined time of the two smaller sequences is $131 + 131 = 262$ clock cycles while one larger sequence requires 251 clock cycles.

5.2.3 CλaSH Model Behaviour

A CλaSH model has been implemented based on Figure 5.14. Besides the application specific parts such as the (de)serializers and linebuffers, the *Front Controller (FC)* and *Back Controller (BC)* are implemented. While the FC is specifically implemented for the use case, the BC is more general as it executes the requests generated by the FC.

Front Controller

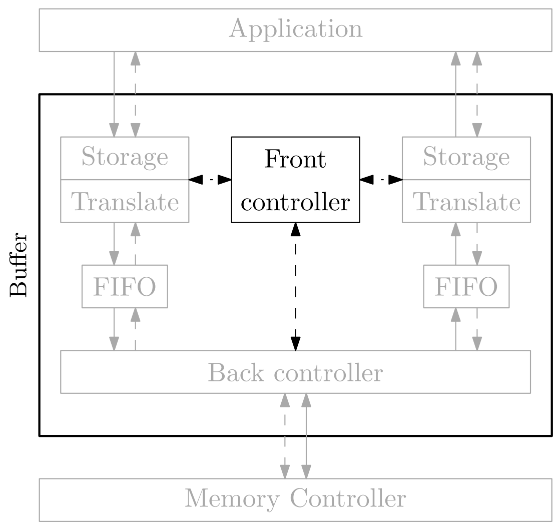


Figure 5.15: Buffer front controller connections

The FC is the central controller of the memory buffer, as shown in Figure 5.15. It schedules request sequences to the memory controller that the BC executes. Furthermore, it synchronises with the application such that the correct data is transferred at the right time.

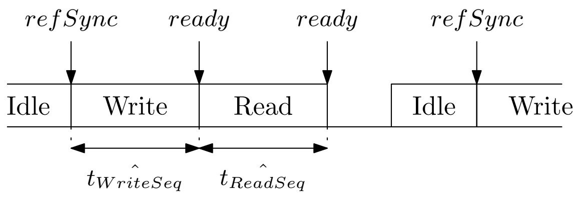


Figure 5.16: Buffer front controller sequence scheduling

The control is based on a state machine that waits for the *refSync* signal to synchronise with the refresh of the external memory, as shown in Figure 5.16. After the synchronisation, the controller processes the sequences encoded in the state machine. When a sequence is enabled, the corresponding request is send to the BC and waits until the sequence is finished to continue with the next sequence. The controller processes the request sequences in a best-effort fashion, such that when the data path corresponding to the request sequence is not ready, the request sequence will be skipped in that round. This problem occurs when the first sequence finishes earlier or is not enabled, such that the start of the second sequence moves forward. A solution to this problem would be to wait for the data path to be ready since the data path should be real-time and thus finish before the worst-case start of the request sequence. Currently, each sequence is only once executed after a refresh while there might be time remaining for more and thus increase the data transfer throughput. A possibility is to configure the number of sequences that can be executed between two refreshes and repeating scheduling sequences alternatingly.

Back Controller

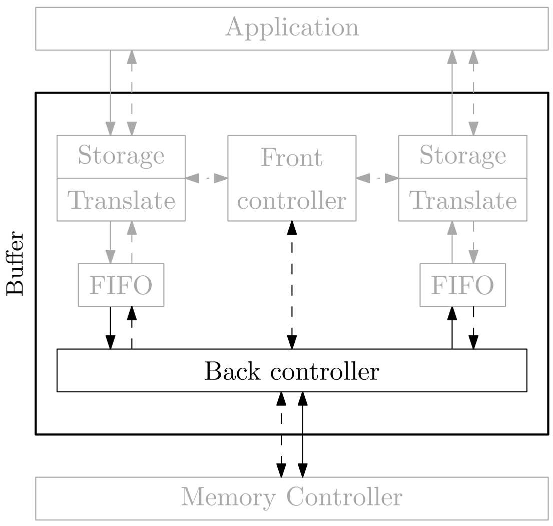


Figure 5.17: Buffer back controller connections

The BC of the memory buffer performs requests to the memory controller on behalf of the FC and is accordingly located between the front controller and memory controller. Furthermore, it is connected to the read and write FIFOs, as shown in Figure 5.17. The FC is concerned with scheduling of request sequences, and the BC is concerned with transferring the request sequence to the memory controller. As the data is mainly stored in an application-specific format, translation of stored data to memory data and vice versa is required. The translation might take longer than a single memory request such that FIFOs temporarily buffer data. The BC depends on the FIFOs for storage such that they should be sufficiently dimensioned to not stall the request sequence.

5.2.4 CλaSH Model Simulation

The buffer is simulated with fixed data input for the write task and a simple application that reads the line buffers. The access latencies are equal to the ones used in the memory controller simulation, but the refresh has been changed to a refresh period of 50 clock cycles in combination with a refresh time of 4 clock cycles. An initialisation delay of 10 clock cycles is used in which the first image line is streamed to the buffer and thus can be processed in the first round. The write task is configured with a stride of 2, and a FIFO depth of 4 as the FIFO is used as buffer storage.

Table 5.5: Buffer simulation parameters

Parameter	Value	Units
rows	64	
columns	8	
tREFI	50	clock cycles
tRFC	4	clock cycles
refMultiplier	1	
initDelay	10	clock cycles
wStride	2	pixels
wFifoDepth	4	
rStride	1	pixels
rFifoDepth	1	
reqs	4	requests
kernel	3	
hpixels	4	pixels
wpixels	16	pixels

The read task has a stride of 1 but translates effectively with a stride of 2 due to the dual port storage line buffers. Therefore, translating the received data from the memory controller takes 1 clock cycle which is shorter than the row hit latency of a single request. Accordingly, the receive buffer only needs a size of 1. The read and write sequences have a length of 4 such that 16 pixels can be transferred in a sequence. The kernel has a size of 3 with an image size of 16 pixels wide and 4 pixels high.

The simulation results are separated in write and read task. In the write task, as shown in Listing 5.4, four image lines will be transferred to the external memory. The read task, as shown in Listing 5.5, starts reading the first line after the line has been stored in the memory by the write task. The simulation has six main columns; the clock count (*CLK*), the front controller state (*Front State*), the back controller request and response (*Back*), the input write request (*Write*), the output of the read task (*Output*) and as last the memory request (*Mem Req*). The *Front State* column is subdivided in the sequence state (*S*), image index (*WI* and *RI*) and memory address (*WA* and *RA*). The *Back* column contains the refresh synchronisation signal (*S*ync), request from the front controller (*Req*) and the sequence ready signal (*Res*). The *Output* contains the line read ready (*R*), line shift (*S*), read column index (*RI*) and the data from the row storage at that column (*Data*).

```
CLK | Front State   | Back           | Write   | Output                  | Mem Req
    | S WI WA RI RA | Sync Req   Res | Request | R S RI Data             |
--- | ------------- | -------------- | ------- | ----------------------- | -------
0   | I    0     0  |                |         |     0  <<0>,<0>,<0>>    |
1   | I    0     0  |                | <1,2>   |     0  <<0>,<0>,<0>>    |
2   | I    0     0  |                | <3,4>   |     0  <<0>,<0>,<0>>    |
3   | I    0     0  |                | <5,6>   |     0  <<0>,<0>,<0>>    |
4   | I    0     0  |                | <7,8>   |     0  <<0>,<0>,<0>>    |
5   | I    0     0  |                | <9,10>  |     0  <<0>,<0>,<0>>    |
6   | I    0     0  |                | <11,12> |     0  <<0>,<0>,<0>>    |
7   | I    0     0  |                | <13,14> |     0  <<0>,<0>,<0>>    |
8   | I 1  0  1  0  |                | <15,16> |     0  <<0>,<0>,<0>>    |
9   | I 1  0  1  0  |                |         |     0  <<0>,<0>,<0>>    |
10  | W 1  4  1  0  | SYNC  16W4     |         |     0  <<0>,<0>,<0>>    |
11  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    | W  16
12  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
13  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
14  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    | W  17
15  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
16  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
17  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
18  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
19  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
20  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
21  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    | W  18
22  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
23  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
24  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
25  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    | W  19
26  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
27  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
28  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
29  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
30  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
31  | W 1  4  1  0  |                |         |     0  <<0>,<0>,<0>>    |
32  | R 1  4  1  4  |       16R4 OK  |         |     0  <<0>,<0>,<0>>    | R  16
```

Listing 5.4: Convolution buffer simulation - Write line

In the write task, as shown in Listing 5.4, eight groups of two pixels are received by the buffer from $t = 1$ to $t = 7$. At $t = 10$, the memory controller is initialised and signals the refresh synchronisation to the front controller. The front controller starts processing the write sequence and sends the corresponding command accordingly to the back controller. It can be seen that the back controller receives the request at $t = 10$, but the first memory request of the sequence is added to the bank's request queue one clock cycle later at $t = 11$. As the data is part of the bank write request, the back controller must read the data from the FIFO before it can send the request to the bank's request queue. Due to the FIFO implementation, the data is one clock cycle delayed such that a write sequence experiences a cycle delay. The first request is expected to be a row-miss such that the remaining requests are expected to be a row-hit as they access the same memory row. With the additional delay caused by the bank request queue as explained in the previous section, the access latency of the memory controller is expected to be $1 \times t_H + 3 \times t_M + 2 = 21$ clock cycles. With the clock cycle delay caused by the back controller, the total latency of the sequence is 22 cycles as seen by the request ready signal (*OK*) from the back controller at $t = 32$.

```
CLK | Front State   | Back           | Write   | Output                 | Mem Req
    | S WI WA RI RA | Sync Req   Res | Request | R S RI Data            |
--- | ------------- | -------------- | ------- | ---------------------- | -------
32  | R 1  4  1  4  |        16R4 OK |         |     0  <<0>,<0>,<0>>   | R  16
33  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
34  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   | R  17
35  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
36  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
37  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
38  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   | R  18
39  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
40  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
41  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
42  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   | R  19
43  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
44  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
45  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
46  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
47  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
48  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
49  | R 1  4  1  4  |                |         |     0  <<0>,<0>,<0>>   |
50  | I 1  4  1  4  |             OK | <17,18> |     0  <<0>,<0>,<0>>   |
51  | I 1  4  1  4  |                | <19,20> |     0  <<0>,<0>,<0>>   |
52  | I 1  4  1  4  |                | <21,22> |     0  <<0>,<0>,<0>>   |
53  | I 1  4  1  4  |                | <23,24> | R   0  <<0>,<0>,<0>>   |
54  | I 1  4  1  4  |                | <25,26> | R S 0  <<0>,<0>,<0>>   |
55  | I 1  4  1  4  |                | <27,28> |     1  <<0>,<0>,<1>>   |
56  | I 1  4  1  4  |                | <29,30> |     2  <<0>,<0>,<2>>   |
57  | I 1  4  1  4  |                | <31,32> |     3  <<0>,<0>,<3>>   |
58  | I 1  4  1  4  |                |         |     4  <<0>,<0>,<4>>   |
59  | I 1  4  1  4  |                |         |     5  <<0>,<0>,<5>>   |
60  | W 1  8  1  4  | SYNC  20W4     |         |     6  <<0>,<0>,<6>>   |
61  | W 1  8  1  4  |                |         |     7  <<0>,<0>,<7>>   | W  20
62  | W 1  8  1  4  |                |         |     8  <<0>,<0>,<8>>   |
63  | W 1  8  1  4  |                |         |     9  <<0>,<0>,<9>>   |
64  | W 1  8  1  4  |                |         |     10 <<0>,<0>,<10>>  | W  21
65  | W 1  8  1  4  |                |         |     11 <<0>,<0>,<11>>  |
66  | W 1  8  1  4  |                |         |     12 <<0>,<0>,<12>>  |
67  | W 1  8  1  4  |                |         |     13 <<0>,<0>,<13>>  |
68  | W 1  8  1  4  |                |         |     14 <<0>,<0>,<14>>  |
69  | W 1  8  1  4  |                |         |     15 <<0>,<0>,<15>>  |
70  | W 1  8  1  4  |                |         |     0  <<0>,<0>,<16>>  |
```

Listing 5.5: Convolution buffer simulation - Read line

After the write sequence, the read sequence is started, as shown in Listing 5.5. In contrast to the write sequence, the read sequence does not require data from the FIFO such that the first memory request of the sequence is started at the same time as the arrival of the command. The read task accesses the already open row, such that the total access latency is expected to be $2 + 4 \times \times t_H = 18$ clock cycles which corresponds to the completion at $t = 50$ in the simulation. Since the write and read task are not guaranteed to access the same row, the worst-case read latency is $t_M - t_H = 3$ clock cycles longer. The read translator de-serializes the read data in a stream of pixels and outputs stride number of pixels each clock cycle. Each received data unit contains 4 pixels which are translated in two stages due to the dual port line buffers. The output of the first stage is ready at the same cycle as the arrival of the data such that translation requires effectively one clock cycle. Due to the buffer between the memory controller and read translator, the output of the translator is two clock cycles delayed after the data is received from the memory controller. In the simulation, the read translator is finished (indicated by the *R* in the *R* column of the *Output* column) 3 clock cycles after the back controller received the last data from the memory controller. The application starts reading the line from the buffer one clock cycle later. The data output of the line buffers is one clock cycle delayed compared to the address due to the clocked output of the BlockRAM. The output data is a vector of buffer outputs in which the first element represents the top row buffer and the last element the bottom buffer. Due to the read stride of one, each buffer output is a singleton value. While the application reads the line pixel-by-pixel, the next write sequence is started after the refresh is completed at $t = 60$.

```
CLK | Front State   | Back           | Write   | Output                 | Mem Req
    | S WI WA RI RA | Sync Req   Res | Request | R S RI Data            |
--- | ------------- | -------------- | ------- | ---------------------- | -------
310 | W 1  8  2  0  | SYNC  20W4     |         |    0  <<17>,<33>,<49>> |
311 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> | W  20
312 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
313 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
314 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> | W  21
315 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
316 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
317 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
318 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
319 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
320 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
321 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> | W  22
322 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
323 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
324 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
325 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> | W  23
326 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
327 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
328 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
329 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
330 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
331 | W 1  8  2  0  |                |         |    0  <<17>,<33>,<49>> |
332 | R 1  8  2  4  |       32R4 OK  |         |    0  <<17>,<33>,<49>> | R  32
333 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
334 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> | R  33
335 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
336 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
337 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
338 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
339 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
340 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
341 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> | R  34
342 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
343 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
344 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
345 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> | R  35
346 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
347 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
348 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
349 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
350 | R 1  8  2  4  |                |         |    0  <<17>,<33>,<49>> |
351 | R 1  8  2  4  |                | <33,34> |    0  <<17>,<33>,<49>> |
352 | R 1  8  2  4  |                | <35,36> |    0  <<17>,<33>,<49>> |
353 | I 1  8  2  4  |            OK  | <37,38> |    0  <<17>,<33>,<49>> |
```

Listing 5.6: Convolution buffer simulation - Write and Read at different locations

A different case is shown in Listing 5.6. In this case, the write and read location are different such that both sequences experience an initial row miss. The write sequence must fetch the data first from the data buffer and accordingly it requires $1 + 2 + t_M + 3 \times t_M = 22$ clock cycles to finish. The read sequence does not need to fetch the data from a buffer before the request can start such that it requires $2 + t_M + 3 \times t_M = 21$ clock cycles. As the line buffer is not cleared after the first case, the data from the line buffer can be ignored.

```
CLK | Front State    | Back           | Write    | Output                     | Mem Req
    | S WI WA RI RA  | Sync Req   Res | Request  | R S RI Data                |
--- | -------------  | -------------- | -------  | -------------------------- | -------
460 | W     16 2  12 | SYNC           |          |      3  <<0>,<0>,<0>>      |
461 | R     16 2  16 |        44R4    |          |      4  <<0>,<0>,<0>>      | R  44
462 | R     16 2  16 |                |          |      5  <<0>,<0>,<0>>      |
```

Listing 5.7: Convolution buffer simulation - Read without write

When no write sequence is executed, as shown in Listing 5.7, the read sequence will start one clock cycle after the synchronization pulse. This one clock cycle delay is introduced by the front controller that checks whether a sequence can be executed or not.

The bank request queue has been configured with a size of 3 such that the back controller can insert 3 requests in the queue while a request is being processed. Therefore, all four requests of the sequence can be inserted in the queue one after the other, as also seen in the memory request queue of the simulation. The request contain the data to be written to the memory in case of a write request such that significant storage is required for the queue. The buffer can store the data such that the request queue adds redundant memory. As no request reordering is used in the bank scheduler, the queue can be ideally reduced to one request such that requests can seamlessly processed without the need for extra memory. Due to the clocked outputs, the *queue full* status flag is updated one clock cycle after the insertion of a request. Additionally, the back controller has to read the data from the write queue of which the data output is one clock cycle delayed. In the current implementation the back controller starts to read the data from the write FIFO at the same time the succeeding request is inserted in the bank request queue. The status flags of the insertion of the succeeding request are updated at the same time as the data arrival such that the back controller cannot store the received data. Currently the request queue must be able to store the pending requests. In the future, this behaviour can be prevented by using for example the FWFT operation mode.

5.2.5 Dataflow Model

The CSDF graph of the read translator has been shown in Figure 5.18. To limit the figure size, only the connections to the basic buffer architecture, as shown in Figure 5.9, instead of the full graph. When data is received by node BC_f the BC from the memory, it is written to the FIFO buffer of the translator. Nodes F_{wRT} and F_{rRT} represent this data and have the same behaviour as the FIFO in the memory controller and N_{RT} represents the buffer size. Assuming 512 bits per request and a pixel size of 8 bits, each request contains 64 pixels and the serializer takes two times the stride (S_r) amount of pixels per cycle due to the double ported BlockRAM in the line buffers. To keep the FIFO read action and seralizer synchronized, the same amount of tokens consumed from F_{rRT} by

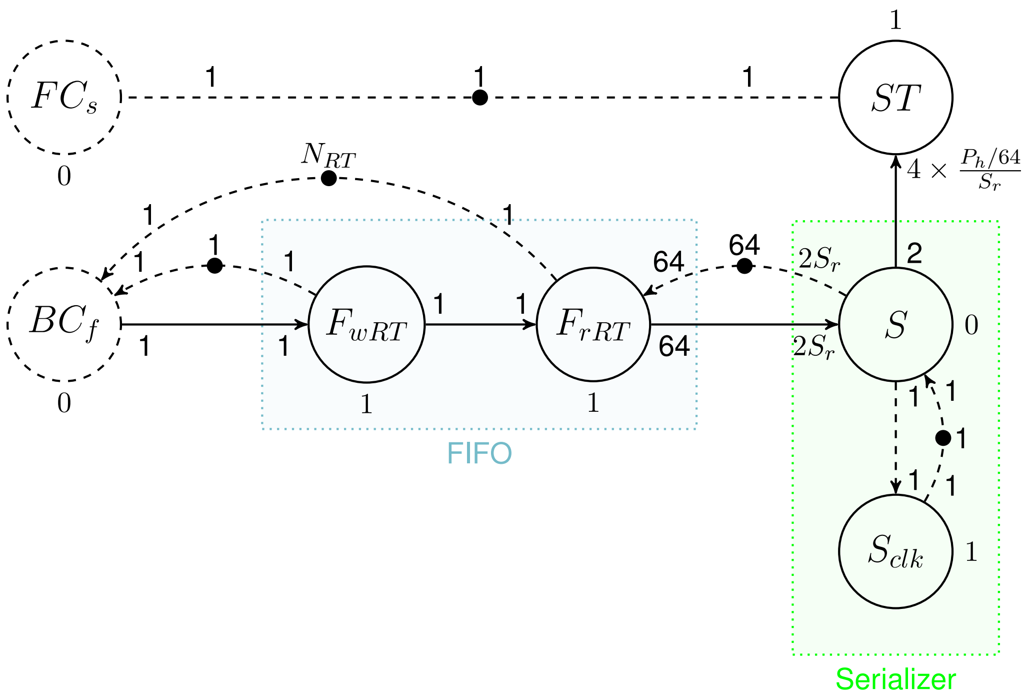


Figure 5.18: Read translator SDF graph

S is also produced and vice versa. The serializer processes the first part of the request data in the same clock cycle as the data is available, but processes the next part in the next clock cycle. Node S_{clk} represents this behaviour by slowing down the firing of node S to a minimum of 1 clock cycle. The serialized data is forwarded to the line buffer represented by ST. Each image line contains P_h horizontal pixels, but due to the number of requests per sequence only a part of the line is read per sequence. Due to the stride factor, the line buffer only contains $\frac{P_h}{S_r}$ entries. The edge from node ST to FC_s prevents that the next read sequence starts before the current sequence is fully processed.

The schedule in Figure 5.19 shows the behaviour of the read translator in more detail. A stride factor of four in combination with a line width of 512 pixels has been used. This combination requires $64/(4 \times 2) = 8$ serialization steps per memory request and two memory requests to fill a line. The Front Controller, Back Controller and thus also the memory behave the same as in the schedule of the basic buffer architecture, shown in Figure 5.10. At $t = 9$ and $t = 13$, the data read from the memory is inserted in the read buffer of the read translator. At $t = 11$, the first item is read from the buffer and the serializer starts processing the data. As the serialization takes longer than the memory request, the second item is stored in the buffer until $t = 18$. The last serialized data is available at $t = 26$ and all data is stored in the storage. Since the last data request is finished at $t = 13$, the back controller and front controller are finished with the sequence and can start processing a new sequence.

Multiple sequences can be chained as shown in the dataflow diagram in Figure 5.20. In the diagram, only the front controller is shown in relation to the refresh. After a refresh, a write sequence of 4 requests is performed followed by the read sequence as described above. The refresh can

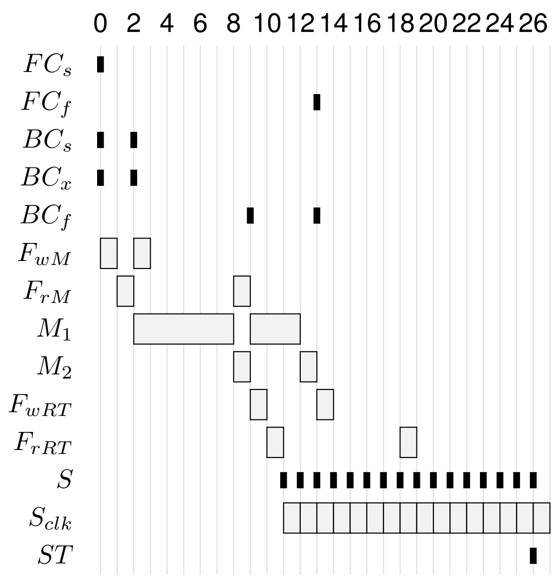


Figure 5.19: Read Translator schedule with two requests

be only executed after the memory transactions of the read sequence are finished, but the edge from the last FC_f should not slow down the system.

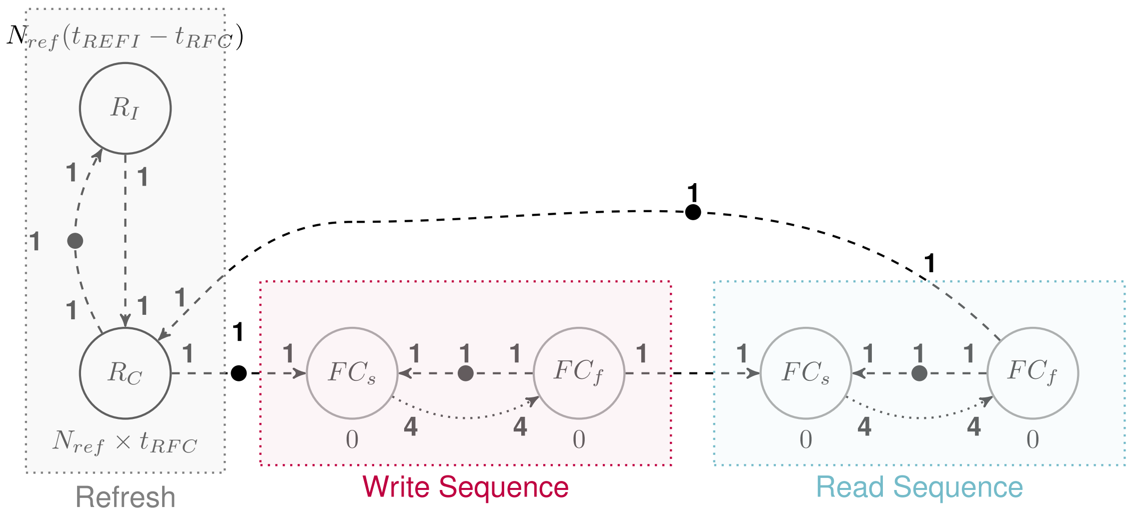


Figure 5.20: Chaining of two request sequences

The schedule in Figure 5.21 shows the chaining in greater detail. After the refresh at $t = 0$, a simple write sequence with 4 requests is performed. As expected, this write sequence requires $1+2+t_M+3\times t_H = 22$ clock cycles. When the write sequence is finished, a read sequence starts that forwards the read data to the read translator. Due to the queue implementation, the read request starts two clock cycles after the write request is finished. As expected, the read sequence is finished after $2+t_M+3\times t_H = 21$ clock cycles. In the example schedule, the next refresh starts three clock cycles after the read sequence is finished after which the process repeats. Since the data read from the memory is temporarily stored in a buffer, the read translator can process the data while the memory is refreshed. Moreover, the write sequence can start directly after the refresh as it is not depended on the read translator. The read translator finishes before the end of the 3rd write request and a new read sequence can start directly after the write sequence is finished. There could be cases that there is no data to be stored or fetched such that sequences have to be skipped, as also shown in the simulation of the buffer. Since the system lacks a fixed scheduler, the sequence after the skipped sequence will be performed potentially earlier. This also happens when expected row-misses turn into row-hits. In cases that an advanced sequence is problematic, back pressure can be used to prevent this. In the use case, the edge from the storage to the front controller start prevents that a specific sequence restarts before the previous sequence is completed.

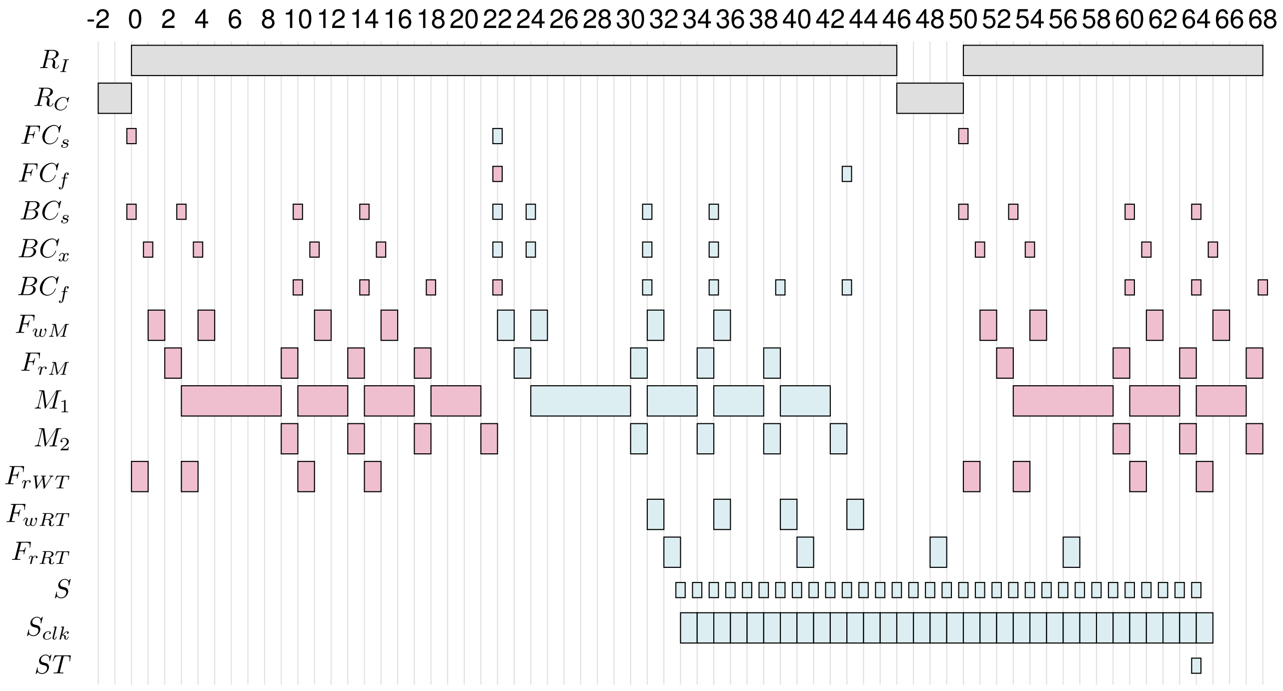


Figure 5.21: Combined Write (purple) and Read (blue) sequence schedule

5.2.6 Conclusion

The 2D image convolution buffer is capable of storing images in the external memory and buffer multiple image lines when reading the image. The multiple image lines can be read parallel such that an image convolution external can traverse over the image. The write and read are performed in an alternating fashion which allows for example read data to be translated while other data is written to the memory. In the current implementation, a write sequence is performed when the memory refresh is finished. This write sequence writes four data units from the write data buffer to the external memory. The write sequence is followed by a read sequence with four read requests. The data read from the external memory is stored in the data buffer of the read translator which will process the data one by one. The data in the buffer is serialized and than stored in the line buffers.

The CλaSH model simulation and dataflow model behave as expected. Both show the same execution of the data requests but have a small difference. While the dataflow model always assumes a row miss when starting a new sequence, the CλaSH model can have a row hit when both the write and read sequence access the same memory row. Furthermore, the simulation was configured such that only two serialization steps where required and the dataflow model schedule has been made using eight serialization steps. When changing the parameters for the dataflow model, the translation schedule will match the simulation. The CλaSH model can also skip a sequence when no data needs to be transferred while the dataflow model cannot. This can be solved by implementing back-pressure from the application, but this is not implemented in the CλaSH model.

Chapter 6

Conclusions

This thesis aimed to create a framework in CλaSH that can generate a real-time memory interface for external DRAM. When the memory access pattern of the application is known at design-time, the knowledge can be exploited to tailor the interface to that specific application. The interface should be simulatable in CλaSH such that the behaviour of the complete system can be verified during design time and a dataflow model of the CλaSH implementation should exist such that real-time performance can be guaranteed. Accordingly, the following research question has been defined at the start of the thesis:

> How can design-time knowledge of memory access patterns be used to generate an optimised real-time memory interface using a framework in CλaSH?

Inspired by the CoRAM architecture proposed by Chung et al. [28], this thesis proposes deterministic actively managed buffers that exploit the design-time knowledge of the memory access pattern of the application to store the correct data at the correct time. The buffers are connected in a private-bank architecture to an open-row real-time memory controller such that each buffer has isolated memory access with bounded access latency. This allows the buffers to operate independently of other buffers and thus also to be analysed independently. By exploiting the open-row policy of the memory controller, efficient sequences of requests can be composed with a lower worst-case latency bound than a sequence of unknown requests such that a guaranteed number of requests can be transferred.

An implementation has been realized for the image convolution use-case. Simulations show that this buffer is able to periodically transfer burst sequences of data with a bound latency and that the implementation can be modelled using a CSDF model. The buffer is synchronised with the memory refresh such that memory requests during memory unavailability are eliminated. The realized buffer can be tailored to the application, but for applications with other memory access patterns, designers have to implement the request scheduler and dedicated memory storage.

Chapter 7

Discussion and Future Work

This chapter focusses on the limitations of the proposed concept which can be further researched in future work.

The current implementation contains a model of the memory controller that uses the worst-case row-hit and row-miss latency bounds as access latency for each request. Those latencies assume the maximum interference from other requests, while in practice not every request will experience this worst-case situation. This causes requests to finish earlier than their worst-case deadline. Currently, requests are processed in a best-effort fashion such that the earlier finish accumulates during the sequence. The front controller does not use back-pressure such that sequences are skipped if the matching data-path is not completed. The data-path is included in the dataflow model and should meet the real-time expectations. Therefore, a future improvement should implement back-pressure based on the data-path of the sequence as the data-path can be safely used as back-pressure. As the buffer translates memory data internally, this might be slower than a memory request such that memory data must be temporarily buffered. When the translation is faster than the earliest finish of an individual request, the translation can also be used to temporarily stall the sequence. When the translation is slower than the worst-case, the translation can be clocked at a higher frequency. More testing is required to validate those methods and to eliminate the use of the FIFO storage.

Due to the current bank request queue implementation, the first request in a sequence experiences two clock cycles delay due to the empty buffer. Ideally, one request is stored such that succeeding requests can be seamlessly processed. By changing the queue implementation, this delay can be avoided and the back controller can generate a new request when buffer is empty.

The front controller is tasked with the scheduling of the request sequences that the back controller executes. The current controller implementation is tailored to the image convolution algorithm such that other access patterns require to reimplement the controller. In future implementations, the controller can be loosely coupled such that it can be passed to the buffer implementation as function parameter.

In contrast to low-level hardware description languages such as VHDL, CλaSH allows to combine multiple signals in a single data type in a tuple or record like structure. Furthermore, signals can be wrapped in an *Maybe* type that adds an enable-like signal to data and thus eliminates the need of an extra enable signal. The interface of each module is based on a request-response record pair such that groups of signals can be access based on element name in the record. At *Signal* level, each connection must be lifted to this domain which is in practice cumber-stone as it requires extra code and records cannot be composed based on element name. For debug purposes, the internal signals of functions might be interesting to watch. This requires to add the internal signal to the output of the function. In a hierarchical design, each module that contains a submodule must forward the debug output data of the submodule to its own output. This pollutes the module as the function type must be adapted accordingly.

Bibliography

[1] Xilinx, “7 series FPGAs data sheet: Overview (DS180),” DS180 (v2.6).

[2] D. T. Wang, “Modern DRAM memory systems: Performance analysis and a high performance, power-constrained DRAM scheduling algorithm,” p. 247.

[3] L. Ecco and R. Ernst, “Tackling the bus turnaround overhead in real-time SDRAM controllers,” vol. 66, no. 11, pp. 1961–1974.

[4] Micron Technology, Inc., “DDR3 SDRAM,” part number: MT41J128M8 Revision: Rev O.

[5] C. Baaij. Clash. [Online]. Available: https://clash-lang.org/

[6] E. Lakis, “FPGA implementation of a time predictable memory controller for a chip-multiprocessor system,” p. 125.

[7] K. Akesson, “Predictable and composable system-on-chip memory controllers.” [Online]. Available: http://repository.tue.nl/658012

[8] Micron Technology, Inc., “Designing for 1gb DDR SDRAM,” TN-46-09, Rev. B 11/09 EN. [Online]. Available: https://www.micron.com/-/media/documents/products/technical-note/dram/tn4609.pdf

[9] Micron Technology, Inc, “Various methods of DRAM refresh,” TN-04-30 DT30.p65 – Rev. 2/99.

[10] Z. P. Wu, Y. Krish, and R. Pellizzoni, “Worst case analysis of DRAM latency in multi-requestor systems,” in *2013 IEEE 34th Real-Time Systems Symposium*, pp. 372–383.

[11] Xilinx, “Zynq-7000 AP SoC and 7 series devices memory interface solutions v4.2.”

[12] P. Yiannacouras and J. Rose, “A parameterized automatic cache generator for FPGAs,” in *Proceedings. 2003 IEEE International Conference on Field-Programmable Technology (FPT) (IEEE Cat. No.03EX798)*, pp. 324–327.

[13] Mentor. Catapult high-level synthesis. [Online]. Available: https://www.mentor.com/hls-lp/catapult-high-level-synthesis/

[14] Xilinx. Vivado high-level synthesis. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[15] Intel. High-level synthesis compiler - intel® HLS compiler. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

[16] Chisel: Constructing hardware in an scala embedded language. [Online]. Available: https://chisel.eecs.berkeley.edu/

[17] R. Appel, H. Folmer, J. Kuper, R. Wester, and J. Broenink, "Design-time improvement using a functional approach to specify GraphSLAM with deterministic performance on an FPGA," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, pp. 797–803. [Online]. Available: http://ieeexplore.ieee.org/document/8202241/

[18] HaskellWiki. Algebraic data type. [Online]. Available: https://wiki.haskell.org/Algebraic_data_type

[19] Marco Bekooij, *Dataflow Analysis for Real-Time Multiprocessor Systems*.

[20] M. Bekooij, R. Hoes, O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. D. Mol, S. Stuijk, V. Gheorghita, and J. van Meerbergen, "Dataflow analysis for real-time embedded multiprocessor system design," in *Dynamic and Robust Streaming in and between Connected Consumer-Electronic Devices*, ser. Philips Research, P. van der Stok, Ed. Springer Netherlands, pp. 81–108. [Online]. Available: https://doi.org/10.1007/1-4020-3454-7_4

[21] S. Heithecker, A. d. C. Lucas, and R. Ernst, "A mixed QoS SDRAM controller for FPGA-based high-end image processing," in *2003 IEEE Workshop on Signal Processing Systems (IEEE Cat. No.03TH8682)*, pp. 322–327.

[22] S. Heithecker and R. Ernst, "Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements," in *Proceedings. 42nd Design Automation Conference, 2005.*, pp. 575–578.

[23] B. Akesson, K. Goossens, and M. Ringhofer, "Predator: A predictable SDRAM memory controller," in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 251–256.

[24] L. Ecco and R. Ernst, "Improved DRAM timing bounds for real-time DRAM controllers with read/write bundling," in *2015 IEEE Real-Time Systems Symposium*, pp. 53–64.

[25] L. Ecco, A. Kostrzewa, and R. Ernst, "Minimizing DRAM rank switching overhead for improved timing bounds and performance," in *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pp. 3–13.

[26] L. Ecco and R. Ernst, "Architecting high-speed command schedulers for open-row real-time SDRAM controllers," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, pp. 626–629.

[27] B. Bhat and F. Mueller, "Making DRAM refresh predictable," in *2010 22nd Euromicro Conference on Real-Time Systems*, pp. 145–154.

[28] E. S. Chung, J. C. Hoe, and K. Mai, “CoRAM: an in-fabric memory architecture for FPGA-based computing,” in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA ’11*. ACM Press, p. 97. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1950413.1950435

[29] M. Adler, K. E. Fleming, A. Parashar, M. Pellauer, and J. Emer, “Leap scratchpads: automatic memory and cache management for reconfigurable logic,” in *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays - FPGA ’11*. ACM Press, p. 25. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1950413.1950421

[30] H. Yang, K. Fleming, M. Adler, and J. Emer, “Optimizing under abstraction: Using prefetching to improve FPGA performance,” in *2013 23rd International Conference on Field programmable Logic and Applications*, pp. 1–8.

[31] H.-J. Yang, K. Fleming, F. Winterstein, A. I. Chen, M. Adler, and J. Emer, “Automatic construction of program-optimized FPGA memory networks,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA ’17*. ACM Press, pp. 125–134. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3020078.3021748

[32] T. Papenfuss and H. Michel, “A platform for high level synthesis of memory-intensive image processing algorithms,” in *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA ’11. ACM, pp. 75–78. [Online]. Available: http://doi.acm.org/10.1145/1950413.1950430

[33] L. Ma, L. Lavagno, M. T. Lazarescu, and A. Arif, “Acceleration by inline cache for memory-intensive algorithms on FPGA via high-level synthesis,” vol. 5, pp. 18 953–18 974.

[34] B. H. J. Dekens, M. J. G. Bekooij, and G. J. M. Smit, “Real-time multiprocessor architecture for sharing stream processing accelerators,” in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 81–89.

[35] Xilinx, “AC701 evaluation board for the artix-7 FPGA user guide (UG952),” p. 108.

[36] ——, “7 series FPGAs memory resources user guide,” p. 88, UG473 (v1.13).