TECHNISCHE UNIVERSITÄT
CHEMNITZ

MASTER THESIS

On-Board Memory Extension on Reconfigurable Integrated Circuits using External DDR3 Memory

Submitted by

Bhaveen Lodaya

(353988)

for the fulfilment of the academic degree

Master of Science in Automotive Software Engineering
Technische Universität Chemnitz

Supervisor:

Prof. Dr. Wolfram Hardt
Department of Computer Science,
Technische Universität Chemnitz, Germany.

Advisor:

Dipl.-Ing. Stephan Blokzyl
Department of Computer Science,
Technische Universität Chemnitz, Germany.

TECHNISCHE UNIVERSITÄT
CHEMNITZ

Aufgabenstellung

zur

Abschlussarbeit
im Studiengang Master Automotive Software Engineering

für

Herrn Bhaveen Lodaya
geb. am 14. Mai 1990 in Dombivli, Thane

zum Thema

On-board Memory Extension on Reconfigurable Integrated Circuits using External DDR3 Memory

Betreuer/ Prüfer:	Prof. Dr. Wolfram Hardt
Ausgabedatum:	09.06.2016
Abgabedatum:	17.11.2016
Tag der Abgabe:	

Unterschrift: .....

Prof. Dr. F. Hamker
Vorsitzender des Prüfungsausschusses

Acknowledgement

I would like to extend my sincere gratitude to all the people who were involved in helping me make this master thesis a success.

I am very thankful to Prof. Dr. Wolfram Hardt for providing me with an opportunity to work on master thesis under his professorship.

A special gratitude I give to my advisor and mentor, Dipl.-Ing. Stephan Blokzyl, whose contribution in the form of his involvement in long discussions, experienced suggestions, constant encouragement and guidance has helped me get through my problems and difficulties.

Furthermore, I would also like to acknowledge the help provided by the other staff members of the Computer Science department by providing me with the tools, equipment and a quiet and peaceful place to work.

I take this opportunity to express my appreciation to my family and friends who have provided me with continuous moral support during the course of the thesis.

Abstract

User-programmable, integrated circuits (ICs) e.g. Field Programmable Gate Arrays (FPGAs) are increasingly popular for embedded, high-performance data exploitation. They combine the parallelization capability and processing power of application specific integrated circuits (ASICs) with the flexibility, scalability and adaptability of software-based processing solutions. FPGAs provide powerful processing resources due to an optimal adaptation to the target application and a well-balanced ratio of performance, efficiency and parallelization.

One drawback of FPGA-based data exploitation is the limited memory capacity of reconfigurable integrated circuits. Large-scale Digital Signal Processor (DSP) FPGAs provide approximately 4MB on-board random access memory (RAM) which is not sufficient to buffer the broadband sensor and result data. Hence, additional external memory is connected to the FPGA to increase on-board storage capacities.

External memory devices like double data rate three synchronous dynamic random access memories (DDR3-SDRAM) provide very fast and wide bandwidth interfaces that represent a bottleneck when used in highly parallelized processing architectures. Independent processing modules are demanding concurrent read and write access.

Within the master thesis, a concept for the integration of an external DDR3-SDRAM into an FPGA-based parallelized processing architecture is developed and implemented. The solution realizes time division multiple access (TDMA) to the external memory and virtual, low-latency memory extension to the on-board buffer capabilities. The integration of the external RAM does not change the way how on-board buffers are used (control, data-flow).

Contents

List of Abbreviations

Abbreviation	Full form
ALU	**A**rithmetic and **L**ogic **U**nit
AMBA	**A**dvanced **M**icrocontroller **B**us **A**rchitecture
ARM	**A**corn **R**ISC **M**achine
ASIC	**A**pplication **S**pecific **I**ntegrated **C**ircuit
AXI	**A**dvanced **E**xtensible **I**nterface
C-Link	**C**amera **Link**
CPU	**C**entral **P**rocessing **U**nit
DDR3	**D**ouble **D**ata **R**ate **3**
DFC	**D**ata-**f**low **C**ontroller
DFM	**D**ata-**f**low **M**anager
DRAM	**D**ynamic **RAM**
DSP	**D**igital **S**ignal **P**rocessor
FIFO	**F**irst **I**n **F**irst **O**ut
FPGA	**F**ield **P**rogrammable **G**ate **A**rray
FSM	**F**inite **S**tate **M**achine
HDL	**H**ardware **D**escription **L**anguage
IC	**I**ntegrated **C**ircuit
I^2C	**I**nter-**i**ntegrated **C**ircuit
IP	**I**ntellectual **P**roperty
ISE	**I**ntegrated **S**ynthesis **E**nvironment
ISim	**ISE Sim**ulator
LUT	**L**ookup **T**able
MIG	**M**emory **I**nterface **G**enerator
MMU	**M**emory **M**anagement **U**nit
OS	**O**perating **S**ystem
RAM	**R**andom **A**ccess **M**emory
RISC	**R**educed **I**nstruction **S**et **C**omputer
RPU	**R**econfigurable **P**rocessing **U**nit
SDRAM	**S**ynchronous **DRAM**
SRAM	**S**tatic **RAM**
TDMA	**T**ime **D**ivision **M**ultiple **A**access
TIFF	**T**agged **I**mage **F**ile **F**ormat
UART	**U**niversal **A**synchronous **R**eceiver **T**ransmitter
USB	**U**niversal **S**erial **B**us

VHDL	**V**HSIC **H**ardware **D**escription **L**anguage
VHSIC	**V**ery **H**igh **S**peed **I**ntegrated **C**ircuit
WRR	**W**eighted **R**ound-**R**obin

List of Figures

List of Tables

1. Introduction

1.1. Motivation

FPGAs are user-programmable integrated circuits which are extensively being used along with the traditional microprocessors in various fields of applications. FPGAs contain an array of logic blocks, lookup tables (LUT), etc. which are used to implement complex functional blocks. It can be used to implement a hardware functional block with a single purpose (like ASIC) and also have the added feature of re-programmability (like Microprocessor). FPGA combines the main advantages of both ASIC and Microprocessor.

The main advantage of a FPGA based system is its ability to perform the task at a high speed and efficiency. Hardware parallelism is the reason that a FPGA is able to achieve faster speeds and better efficiency thereby achieving higher throughput.

The advantages that FPGAs have over other types of data processing units are:

1. They are efficient at parallel data processing. This is a major advantage of FPGA over other reconfigurable ICs like Microprocessors which process data sequentially.

2. They are capable of processing large size of data at high speeds. Unlike Microprocessors, those are limited to data processing equivalent to its arithmetic and logic unit (ALU) size (e.g. 8bit, 16bit, 32bit, 64bit).

3. They are reliable. Since, FPGAs are configured using hardware description language (HDL), a developer has a complete control, up to the gate level which helps in designing time critical and reliable systems.

4. They have a variety of interface options like UART, I^2C, USB, Ethernet, etc. due to which FPGAs can be interfaced with different kinds of electronic devices.

5. They have a broad range of applications. FPGAs are being used in wide areas like Aerospace and Defence, Medical Electronics, Scientific Instruments, Consumer Electronics, Security Systems, Image Processing, Automotive sector, etc.

FPGAs also include some kind of data storage in the form of memory blocks which are used to temporarily buffer the data being processed and push the data from one functional block to another as and when required.

When FPGAs are used for high volume data processing (e.g. the data parallelization mentioned in the research article [SB12]), the amount of data required to be buffered is in the order of few MBs. The amount of memory available on the FPGA is around 2.5MB (combination of the block RAM and distributed RAM on Virtex-6 XC6VLX240T [Vir15]).

This shows that the amount of memory resources available on-board is very less as compared to the data required to be stored in the memory. This is one of the major disadvantages when using FPGA for high volume data processing.

Also, in real world applications when multiple functional entities are combined together to make a bigger entity and work together, the entire system might not yield the expected speed and throughput. One of the problems that the system designers face is where the systems combined together have a difference in the data processing speeds at an individual level.

For example, if there are two entities which are working together to achieve a particular task and the first entity has twice as higher speed of data processing as the second entity. In such a case, the first entity has to wait for the second entity to complete its processing before it can move onto the processing of the next available data input.

Asynchronous buffers (First In First Out - FIFO) are used to handle such problems. Asynchronous FIFOs reduce the amount of the stalling time and thereby reducing the latency of the overall system [Tal14]. But, the amount of FIFOs that can be implemented on an FPGA is also limited by the amount of on-board memory available. These problems further increase when there are multiple instances of such logic blocks which require high amount of data storage.

Within the scope of this master thesis a method has been proposed to increase the on-board memory with the help of external DDR3 SDRAM with an effort to reduce the overall latency of the system and a smooth functioning of all the logic blocks connected to one another.

1.2. Structure of the master thesis report

Chapter 2: Provides information about the state of the art which consists of different methods proposed by other researchers to solve the problem similar to the one mentioned above. It describes the idea of creating an arbiter so as to facilitate the flow of data from 2 separate entities to an external RAM. One of the methods presents a concept of a new SDRAM controller designed from scratch without the use of any already available memory controllers provided by Xilinx. Data caching techniques found in the modern processors are also considered to be an idea for the development the solution for this master thesis.

Chapter 3: Describes the concept involved behind the development of base solution for the memory management module i.e. Data Flow Manager (DFM). It also contains different approaches designed in order to improve the DFM and the reason behind selection of a particular approach and its implementation. The architecture of the DFM and all the modules required to develop the final entity are also described and explained in this chapter.

Chapter 4: Gives an overview of the hardware and tool chain used to realize the whole system. It explains in brief all the pre-built logic blocks (Intellectual Property Core - IPCore) used in order to implement the DFM.

Chapter 5: Involves the test environment created to evaluate and validate the system function. The architecture of the environment and the structure of all the modules used to create this environment are explained in brief. This chapter also includes the different test scenarios used to verify the working of the system and different results observed and discusses these results in detail.

Chapter 6: Concludes the master thesis, points out the shortcomings of the design. Recommendations for the removal of these shortcoming and bugs are explained and ideas for the future improvements of this topic are proposed.

2. State of the Art

2.1. Memory management to support multitasking on FPGA-based systems

Klaus Danne of University of Paderborn *"introduced a concept of Memory Management Unit (MMU). MMU is designed in a way that is is capable to handle the concurrent operations of multiple tasks to banks of a single external RAM"* in research article [Dan04a]. MMU is used to store the state of the currently ongoing task to be able to handle the interrupt from a new task.

The development of a MMU is required to provide abstraction and resource management for the tasks, which are normally provided by the Operating System (OS). In normal case of a computer microprocessor and the OS, the memory management is done with the help of support from the OS and dedicated MMU of the processor. However, in case of the FPGA, MMU is not a dedicated unit but a module created on the FPGA itself to handle the memory management.

The features required for the MMU are [Dan04a]:

1. virtual addressing
2. management of the external dedicated RAM
3. memory caching not needed
4. resolution of the task conflicts when multiple tasks require to access the same memory bank
5. access to the multiple memory banks in parallel
6. tasks are supposed to handle the memory access delays

2.1.1. Virtual addressing

The virtual addressing is required to make the memory addressing independent of the other tasks that are running. Therefore, every task gets a memory slice address starting from 0x00...0 and ends at a value depending upon the size of the memory slice. If all the information regarding the tasks is available beforehand

then the virtual address can be converted to the physical address. This virtual addressing is required, as the number of tasks is not known at the design time.

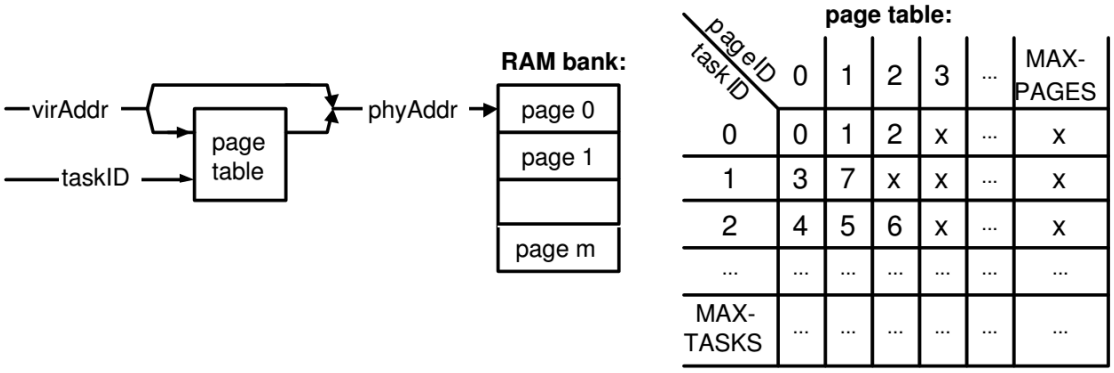

Figure 2.1.: Mapping virtual address to the physical address using a page table [Dan04a]

To implement the address translation at runtime, information regarding the address mapping is needed to be stored in the MMU. Figure 2.1 show an example of the address mapping technique. Here, the page table information is stored in the MMU. The high-word of the virtual address along with the task ID is used to address the page table and the output of the page table is used as the high-word of the physical address. The low-word of the virtual address is used directly as the low-word of the physical address. The width of the higher part of the word and the lower part of the work depend completely on the value of the chosen page size. Due to such an easy virtual address mapping technique the page size is limited to be a power of two.

There are two additional features of using such a simple method [Dan04a]:

1. **Memory protection:** Due to presence of the page table, the MMU is aware if a particular task is trying to access a memory location outside its address bounds. This helps in avoiding errors created due to faulty tasks.

2. **Inter task communication:** Inter task communication is possible via access to the same physical memory locations. This can be done by setting the page table entry to same value for two different tasks. Additional handshake signal mechanisms have to be implemented to avoid both the tasks trying to read the same memory location at the same time.

2.1.2. Dynamic memory allocation

Dynamic memory management is required when new tasks enter a system. MMU allows a task to allocate or de-allocate memory dynamically at runtime with the

help of 'requestPage(taskID)' function and 'releasePage(taskID)' function. Figure 2.2 shows the function snippet along with the free page stack, which stores the information regarding the allocated or de-allocated pages.

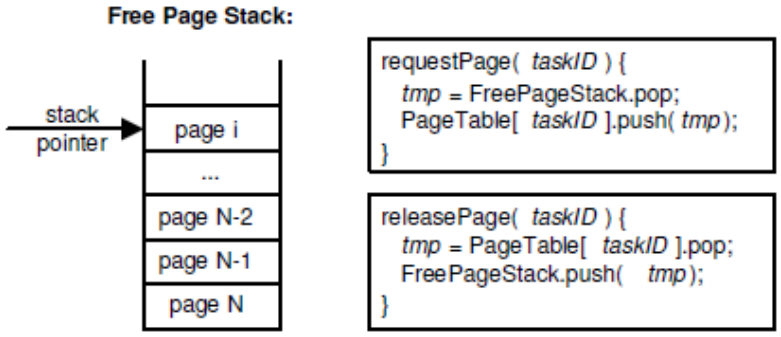

Figure 2.2.: Allocation and de-allocation of memory pages [Dan04a]

2.1.3. Priority based scheduled memory access

As discussed previously, inter task communication is possible via the access to the same physical memory location. For this to happen, the address and data bus of the memory have to be multiplexed. Figure 2.3 show the physical architecture of this mechanism.

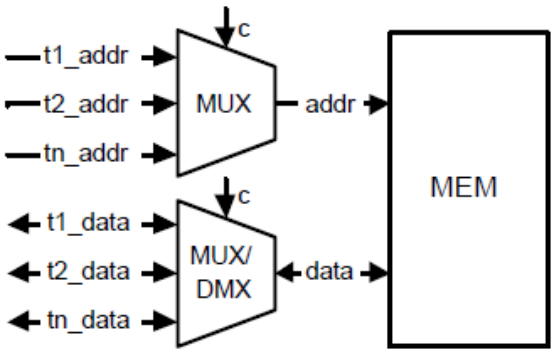

Figure 2.3.: Multiplexed access to one physical memory [Dan04a]

A control signal is used to manipulate the access given to memory for a particular task. If a single task requests access then the MMU switches the multiplexers so that the task is able to complete its write/read operation. If multiple tasks require access to the memory at the same time, a priority based scheduling of the memory access is done to guarantee error free operation that is well within its timing constraints.

2.2. Memory management abstraction for self-reconfigurable video processing

Kurt Franz Ackermann of Darmstadt University of Technology *"presents a concept for an SDRAM controller targeting video processing platforms with dynamically reconfigurable processing units (RPUs)"* in research article [AHIG09]. Multiple modules present on the FPGA for the task of video processing require the data to be stored and read from the external DDR3 SDRAM.

Figure 2.4 represents the Frame grabber structure and the data flow required for the reconfigurable video processing platform. The Camera-Link (C-Link) acts as an interface between the frame grabber and the camera. The frame received from the C-Link is first written to the external RAM via the memory controller. Here, the memory controller acts as the main communication centre for all the different modules. There are n different RPUs which work on the video frames buffered in the memory. There is Gigabit Ethernet interface which connects the whole system to the computer. When all the RPUs complete processing the data, the result is then transferred to the computer via the Ethernet. This whole cycle then repeats for the next video frame.

The memory controller developed for the task requires being complex so that it is capable to provide data to all the random data requests generated by the complex video processing algorithms. Also, the complexity of the controller increases as the RAM clients are unaware of the data organization in the memory.

The proposed memory controller architecture is presented in the figure 2.5. The functionality provided by the memory controller is [AHIG09]:

1. Support for multiple RAM clients
2. Priority arbitration
3. Support for read/write data bursts
4. Memory partitioning
5. Frame-based ring-buffers
6. Support for variable frame dimensions
7. Support for high-level addressing
8. Providing high-level status information

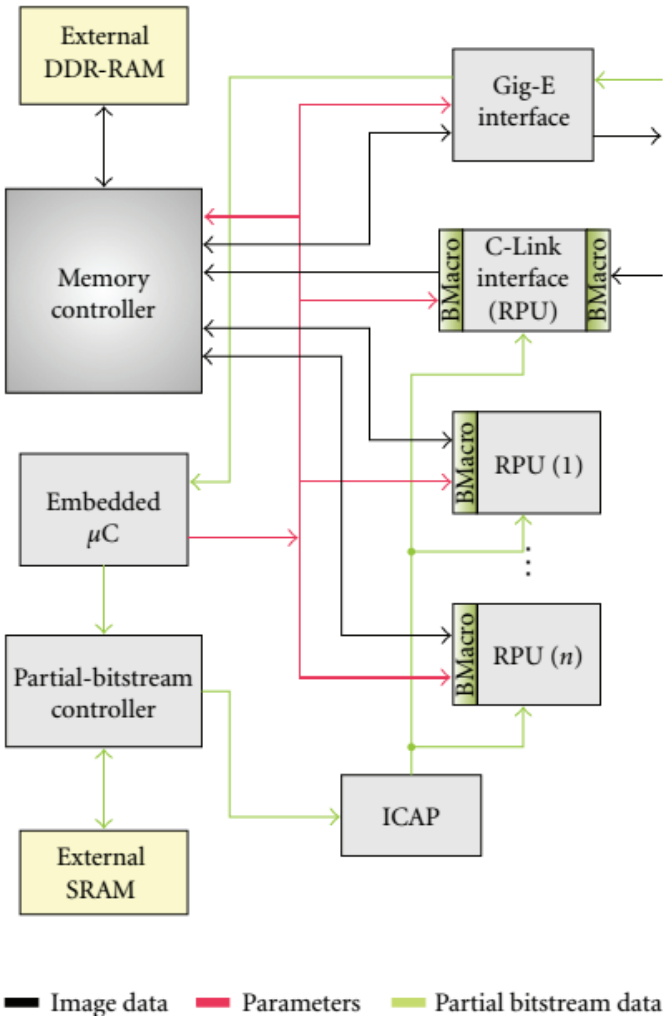

Figure 2.4.: Frame grabber structure [AHIG09]

2.2.1. Arbiter

The main advantage of the FPGA based system is ability of its different modules to work in parallel with respect to each other. In such a case, there is possibility of these different modules (RPUs) request data from the RAM at the same time. An arbiter needs to be implemented which can take care of the requests and satisfy the quality of the system.

The arbiter works on a weighted round-robin (WRR) algorithm which provides the RPUs with the chance to complete its data operation depending upon the clients' priorities.

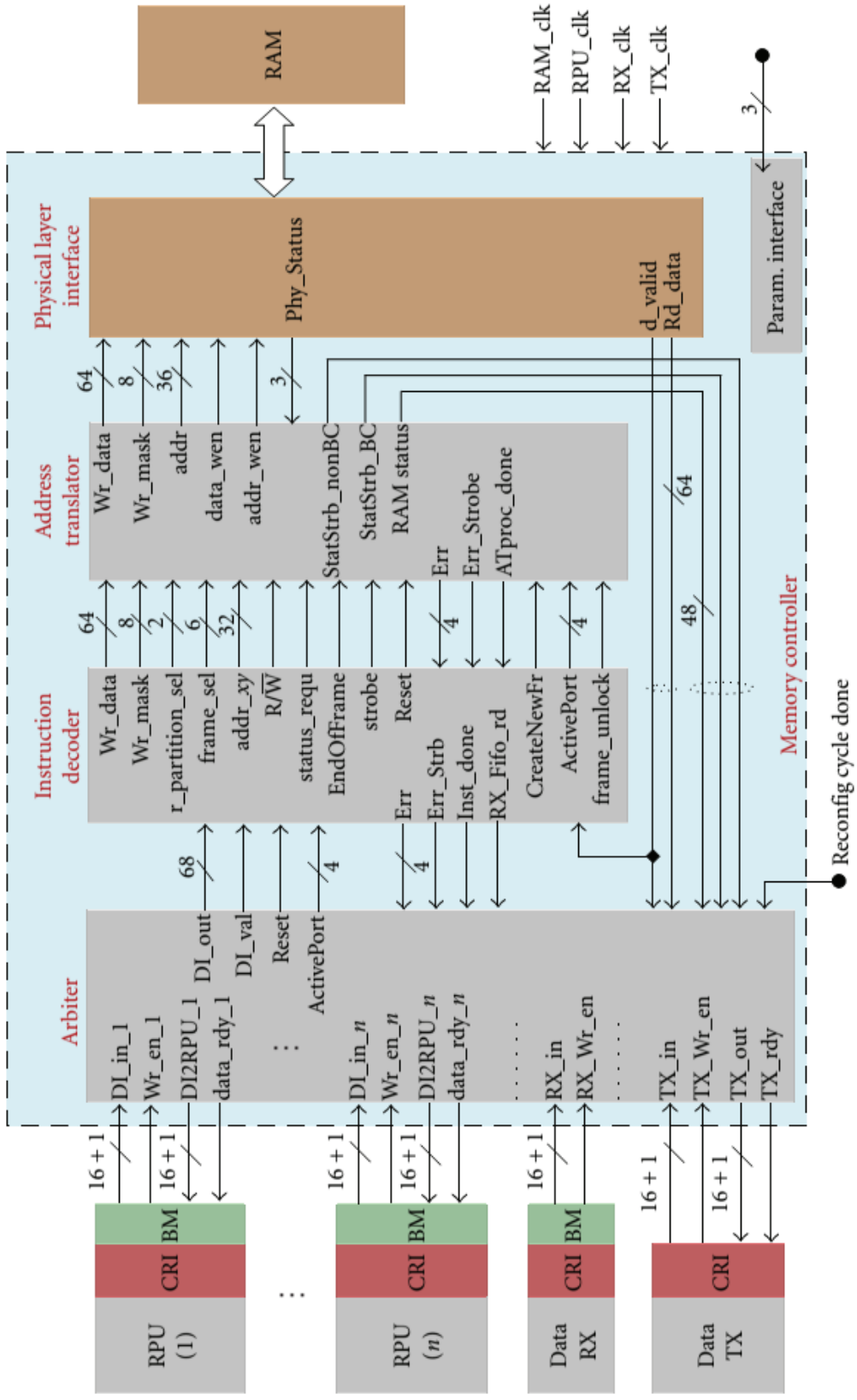

Figure 2.5.: Architecture of memory controller [AHIG09]

2.2.2. Instruction decoder

The Instruction decoder works on the commands received from the different clients. It decodes the instructions like *"create-new-frame, unlock, end-of-frame"* [AHIG09] and does the required pre-processing for the instructions. The decoder acts as a bridge between the arbiter and the address translator. Its allow the valid instructions for the address translation while invalid instructions generate appropriate error codes which are sent back to the arbiter.

2.2.3. Address translator

The Address translator handles the job of the maintaining the data in the physical RAM in suitable memory structures. It provides the required abstraction of the memory to the clients. The data addressed by the clients is in the units of pixels, lines and frames. Address translator partitions the data to avoid data corruption. The partitions are organized as frame ring-buffers which give access to the latest frames given by the clients.

2.3. FPGA design for DDR3 memory

Laura Fischer and Yura Pyatnychko of Worcester Polytechnic Institute *"presents a memory arbiter system capable of sanctioning two systems to interact with a single DDR3 SDRAM memory"* in their bachelor thesis [FP12]. The DDR3 memory controller developed by Xilinx supports communication between one system and memory. In real word FPGA-based applications, there are chances of more than one systems trying to communicate with the memory. If two such systems try to communicate with the memory simultaneously, there is a possibility of data corruption in case of data writing and false data reception in case of reading.
Hence, a traffic controller which keeps check on such multiple simultaneous requests and provides the response in a manner such that it avoids the data corruption is required. The arbiter not only avoids the requests collision but also maintains their order to ensure the data available in the memory is up-to-date. The requirements for such memory arbiter are listed as follows [FP12]:

1. Arbiter should take into consideration the memory's refresh rate
2. Arbiter must maintain the requests from the two systems in order
3. Arbiter must keep a check on the amount of time a system uses the memory

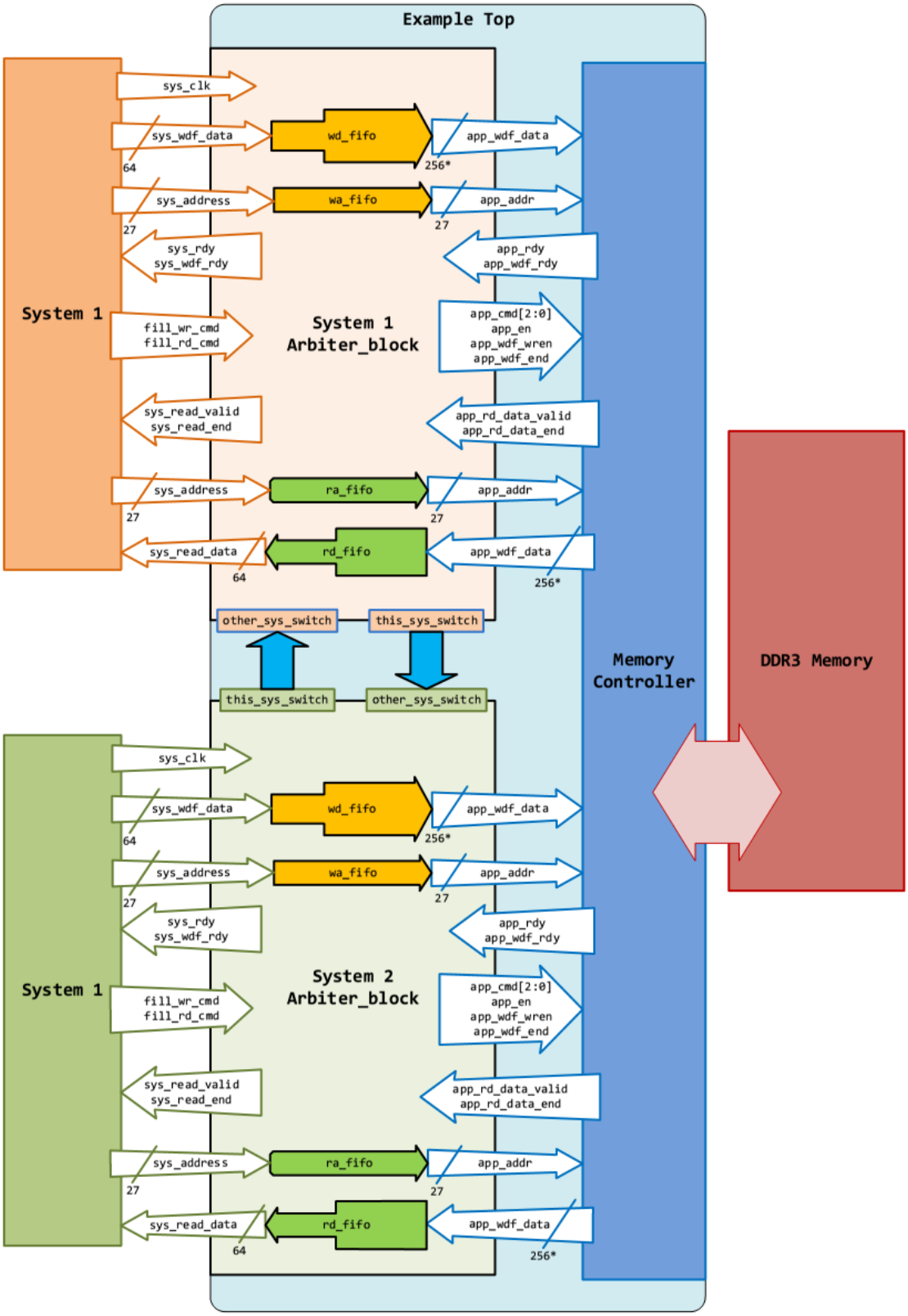

Figure 2.6.: Two systems memory arbiter block diagram [FP12]

The figure 2.6 contains the design of two systems connected to the memory via an Arbiter block. The arbiter contains all the necessary FIFOs required to buffer the read and write commands and the write data before they are transferred to the memory controller. There are two address FIFOs to store the addresses of the read and write commands. There is a 64-to-512 FIFO which is used to store the data to be written to the memory. A 512-to-64 FIFO is also available to store the data read back from the memory. The arbiter uses two signals: "other_sys_switch" and "this_sys_switch" which check the current system that is using the memory controller and limits the other one from using it. There are two copies of arbiter running simultaneously with in accordance to each other. The inter communication of these two arbiter blocks allows the two systems to maintain an error free data communication with the common external memory module.

2.4. Caching techniques in x86 processors

The Central Processing Unit (CPU) of a computer runs at a higher clock speed as compared to that of a memory module. The speed at which the data is provided to the CPU from the RAM creates a bottleneck in the operation. To avoid such bottlenecks and to hide the memory access latencies from the CPU the concept of data caching came into existence.

Caching consists of storing the most frequently used data in a memory as near to the CPU as possible. This cache memory is much faster as compared to the RAM. Along with being faster, this cache is also expensive and hence available in smaller quantities in comparison to that of a RAM. At the system reset, the cache is completely empty. At the first request, the data is read from the main memory and passed on to the CPU. A copy of this data is also maintained in the cache for further use. The initial read from the main memory costs high latency but it cannot be avoided. But all subsequent accesses to the same data are met with minimum latency possible as the data is already present in the cache memory.

Since the amount of cache is very small and is not capable of incorporating lot of data, a new level of cache is introduced which is slower than the previous cache but bigger in size as compared to the previous cache. The cache nearer to the CPU is called level 1 (L1) cache and the cache between L1 and the RAM is called level 2 (L2) cache.

L1 and L2 caches are available on the CPU chip itself and that is the reason why their size is very limited. To overcome the size limitations, the motherboards on which the CPU is connected also contain a level 3 (L3) cache. This leads to a memory hierarchy.

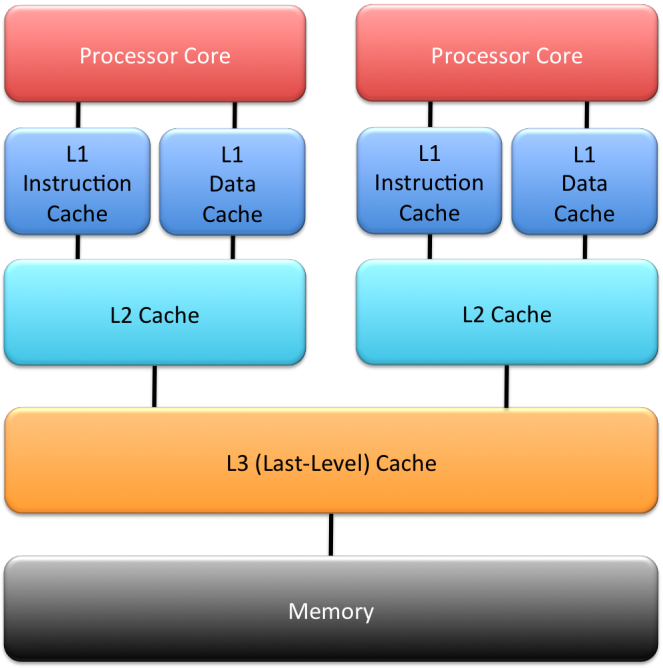

Figure 2.7.: Memory hierarchy [Hei15]

Figure 2.7 show the memory hierarchy found in the modern day computer. The size of the cache memories for the Intel's Sky Lake Micro-architecture is as follows [Ark]:

1. L1 Cache (Instruction) – 64KB per core
2. L1 Cache (Data) – 64KB per core
3. L2 Cache – 256KB per core
4. L3 Cache – 8192KB shared

2.5. Feature requirements for the on-board memory extension

In the previous sections of this chapter, different ideas regarding the implementation of the memory controller to increase the amount of memory available on-board with the help of external RAM have been discussed. There are different advantages and disadvantages of these methods with respect to the solution required for the problem mentioned in the chapter 1.

Based on the ideas and concepts used above, the features required or the concepts that can possibly be useful for the development of the memory extension are:

1. Virtual addressing: So that the different clients are not dependent on one another for the addressing.
2. Memory partitioning: To divide the memory equally between different clients.
3. Data caching: To satisfy data requests immediately whenever possible.
4. Scheduling algorithm: To ensure that every client gets a chance for data operation without collision with respect to another.
5. Memory protection: To ensure data written to a particular memory partition belongs to the same client.
6. Strict data ordering: To maintain data order when data appears in a long continuous stream.
7. Scalable architecture: To incorporate the variable number of clients.

3. Concept, Design and Implementation

As discussed in the previous chapter, an entity which reflects the working similar to that of the caching technique in case of computer processors needs to be implemented for the FPGA. Along with the low latency data-flow it also needs to be capable of implementing data-flow not only from one input entity to one output entity but it needs to be capable of managing the data-flow from multiple input entities to corresponding output entities.

Data-flow manager (DFM) is the solution to our problems which has the facility to transfer data from multiple input sources to the corresponding destinations directly if destination is ready to accept the data or via the external RAM as and when required.

3.1. Single-channel data-flow manager

The figure 3.1 portrays the architecture of a single channel DFM. WRITE_0 and READ_0 denote the level 0 buffers and WRITE_1 and READ_1 denote the level 1 buffers. The input and output data widths of each buffer are as mentioned in the table 3.1.

Buffer	**Input data width (bits)**	**Output data width (bits)**	**Memory location size (bits)**
WRITE_0	8	64	64
READ_0	64	8	64
WRITE_1	64	512	512
READ_1	512	64	512

Table 3.1.: Port width and memory location width of buffers

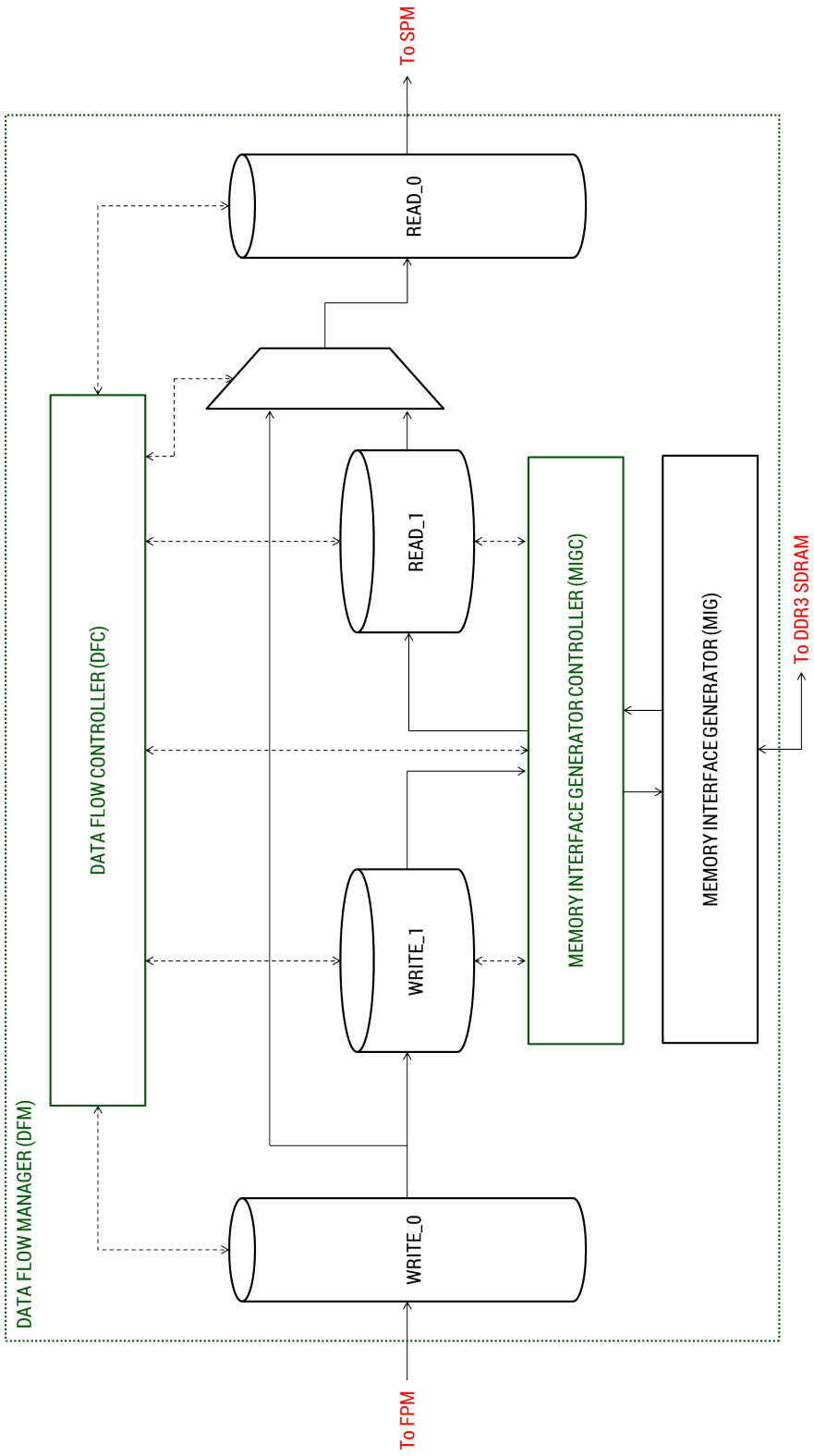

Figure 3.1.: Single-channel data-flow manager

Here the data-flow occurs via two different paths:

1. From WRITE_0 to READ_0 directly.
2. From WRITE_0 to READ_0 via WRITE_1,DDR3 SDRAM and READ_1.

The data is directly transferred from WRITE_0 to READ_0 if there is free memory location in the READ_0 buffer and there is no data present in any other buffers or external RAM. When the buffer READ_0 runs full then the data from WRITE_0 is transferred to WRITE_1 and it follows the second data path mentioned above.

The decision regarding the path that the data will flow through is being taken by Data Flow Controller (DFC). DFC keeps track of the empty and full signals of all the buffers and also on signals from the Memory Interface Generator Controller (MIGC) which helps in deciding if there is empty memory location in READ_0 and absence of data at any other buffer level or in RAM so as to allow a direct data transfer from WRITE_0 to READ_0 or not.

DFC not only makes the decision but also provides all the necessary signals to all the others entities which are required to initiate the data transfer.

The job of the MIGC is to read data from the WRITE_1 and generate appropriate write address required to write the data to RAM and provides this information along with write command and enable signals to the Memory Interface Generator (MIG). It also checks if there is empty memory location in READ_1 and generates appropriate read address. It provides the read address, read command and enable signal to MIG so as to initiate a read from the RAM. The data read from the RAM is then written to READ_1.

MIG is an Intellectual Property Core (IPCore) provided by Xilinx which helps in easy data write/read to/from external DDR3 SDRAM. Chapter 4 contains detailed information regarding the working of the MIG.

3.2. Multi-channel data-flow manager

Working of multi-channel DFM is such that local data transfers occur in each and every channel in parallel with respect to one another. Whenever the READ_0 of any channel gets full, there arises a need for the data to be stored in RAM. This data is first transferred to the next level WRITE_1 buffer. Now there are multiple WRITE_1 buffers which expect to write the data to RAM. A basic round robin manner is used where each and every channel gets a chance to write data to the RAM. Similarly, reading from RAM also happens in a round robin manner. There are few variations to a multi-channel DFM which are mentioned below which help

us in achieving the desired goal of low latency data-flow along with limited amount of on-board resource usage.

3.2.1. Super-scalar approach

Figure 3.2 depicts the very basic and logical version to implement a multi-channel DFM from a single-channel DFM. Here, the multiple channels are created by the replication of the modules present in the single channel without any extra data buffers.

The advantage to this design is that it is very easy to implement with minor modifications to the data-flow controlling modules.

Due to round robin manner, each and every channel gets less number of chances for data transfer to RAM. Hence, the possibility of the buffers on the write side running full increases. This is a major disadvantage to this approach and this problem keeps increasing in direct proportion to the increase in the number of channels.

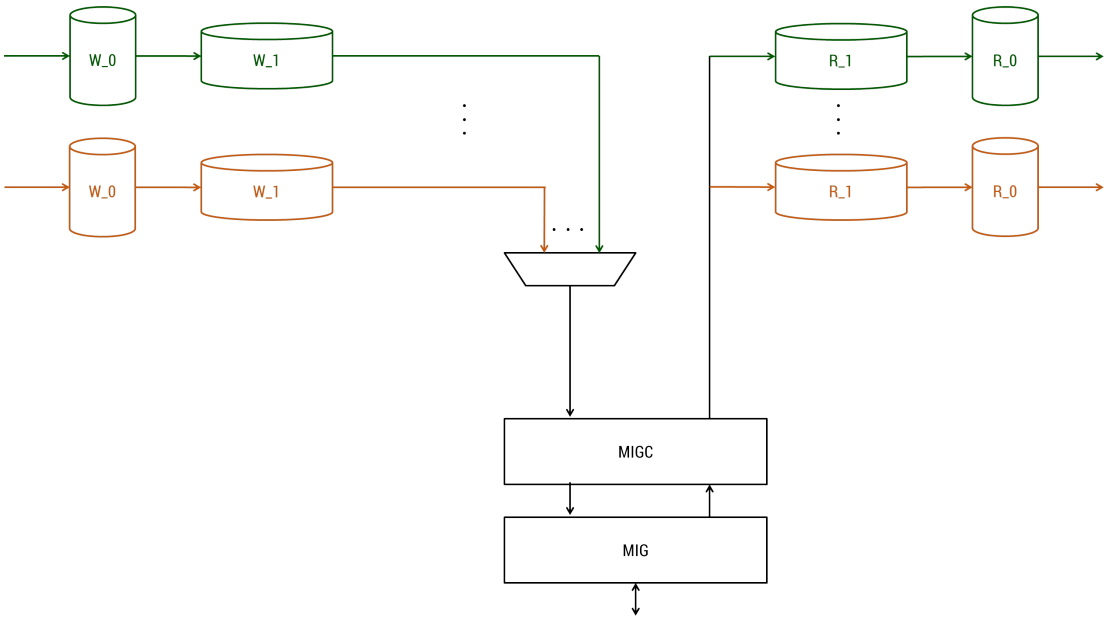

Figure 3.2.: Architecture for super-scalar multi-channel data-flow manager

3.2.2. Super-scalar approach with individual WRITE_2

Figure 3.3 represents the next approach towards the development of multi-channel DFM. To reduce the risk of write buffers running full an extra level of write buffer

has been introduced. WRITE_2 buffer has both input and output data width of 512bits.Addition of extra level of buffer increases the amount of data that can be stored on the write side thereby reducing the pressure of data transfer to the RAM.

This approach overcomes the problem mentioned in the previous approach but it gives rise to another problem. It increases the amount of on-board memory usage which is already limited.

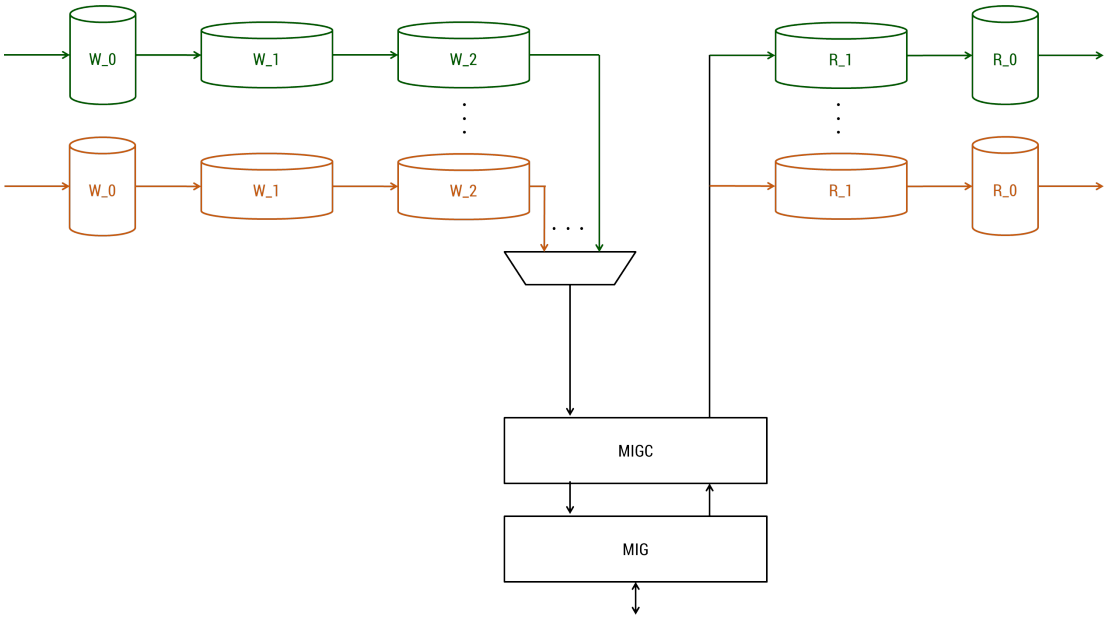

Figure 3.3.: Architecture for super-scalar multi-channel data-flow manager with individual WRITE_2

3.2.3. Super-scalar approach with common WRITE_2

Figure 3.4 describes the final approach which is a trade-off between the previous two approaches. The input data to all the channels comes from different sources and is incoming at different rate. Therefore, the channels where the input is connected to a source with low data rate might not have a risk of write buffers running full. Hence, instead of connecting WRITE_2 buffer to all the channels, there is only need to attach these level 2 buffers to only required channels. But for this to be possible there has to be prior information about the data rate of all the sources.

In absence of the data rate information a generalized WRITE_2 buffer has to be implemented. Hence, a common WRITE_2 buffer is used which is shared by all the channels and its memory locations will be in majority occupied by the channels

with a high data rate input.

This approach not only reduces the risk of buffers running full but also keeps a control over the amount of on-board resources used. Hence, this approach has been chosen to develop the multi-channel DFM. Also, the multi-channel DFM is designed in a generic way such that the user can choose the number of channels from a range of 1 to 32.

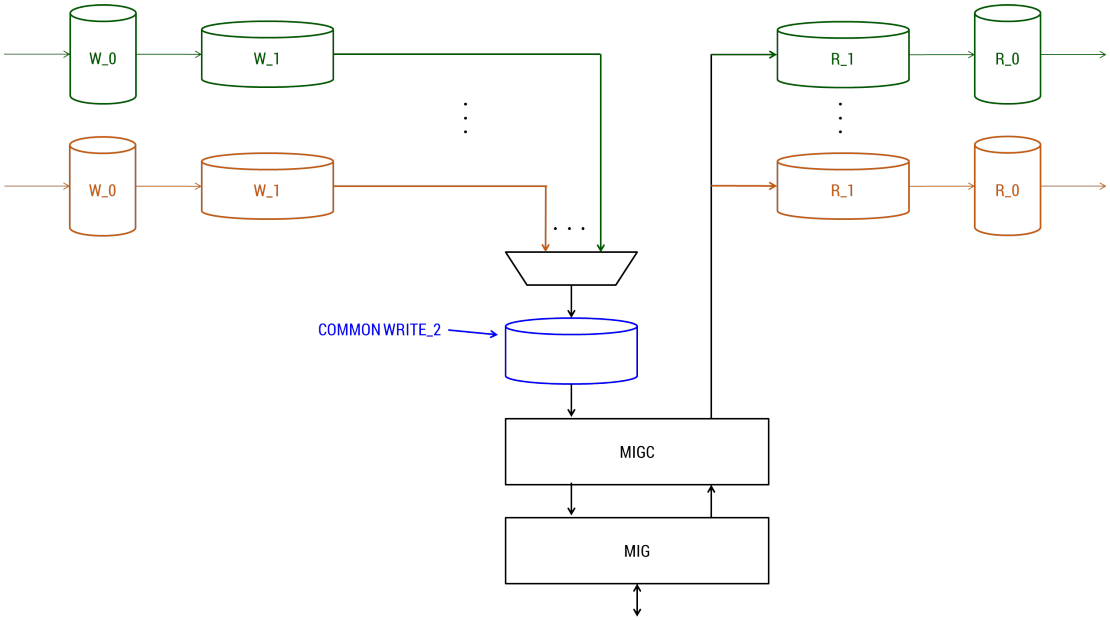

Figure 3.4.: Architecture for super-scalar multi-channel data-flow manager with common WRITE_2

3.3. Common WRITE_2 FIFO

The common write buffer has both input port and output port width of 517bits. The lower 512bits of the buffer location is filled with the data received from the WRITE_1 buffer of the channel. The higher 5bits are used to store the id of the channel from where the data is received from. The Xilinx FIFO IPCore is used to create all the buffers. The core generator does not directly provide an option to modify the data width of the ports. Hence, the number of bits for the channel id is limited to 5bits which limits the maximum number of channels possible to 32.

3.4. Data-flow controller

As discussed in the section 3.1, DFC is the main component which handles the job of decision making for the data path selection. In case of a single-channel DFM a single DFC is present which communicates with all the other components. But when the numbers of channels increase, as in case of the multi-channel DFM, there comes into existence multiple DFCs which need to communicate with a single MIGC.

To make it easier for data synchronization between multiple channels and to avoid increasing the complexity of the DFC to incorporate the status of other channels, DFC is split into two parts: DFC Master and DFC Slave.

DFC Slave now plays the same role as the DFC in the case of single-channel DFM except for handling the data transfer between WRITE_1 to WRITE_2 and between MIGC and READ_1.

DFC Master handles the round robin iteration of all the channels and data transfer from the WRITE_1 to WRITE_2 as and when required. On the other hand, MIGC iterates through the READ_1 and provides the data to the read buffer as the need arises.

3.4.1. Data-flow controller master

Data Flow Controller Master is designed as per the state machine in the figure 3.5. The main job of DFC Master is to transfer data from the WRITE_1 of each channel to common WRITE_2 in a round robin manner.

The DFC Master checks the empty signal (CHNL_W_1_EMP) of WRITE_1 buffer of the current channel, full signal (W_2_FULL) of WRITE_2 and full signal coming from MIGC (MIG_CNTLR_CHNL_RAM_FULL) for the memory slice dedicated for the current channel and decides whether to initiate the data transfer or not. Then it increments the channel id and repeats the same for the next channel. DFC Master gives the appropriate read enable signal, write enable signal and select line for the multiplexer and confirms the data transfer.

The data width of the output from WRITE_1 is 512bits and the data width of the input to WRITE_2 is 517bits. The 5bits of current channel id not only acts as the select line for the mux but also serve as the higher 5bits of the data input to the WRITE_2.The channel id stored along with the data in the WRITE_2 is used by MIGC to calculate the appropriate memory location address for the write operation.

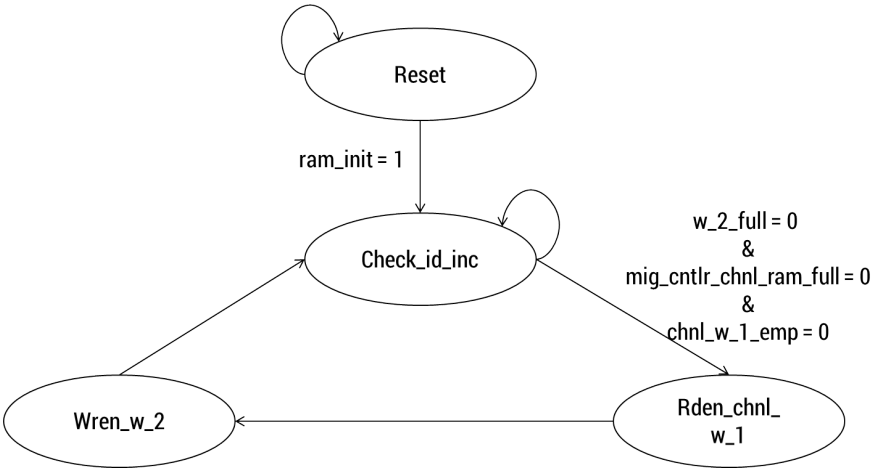

Figure 3.5.: Finite state machine for data-flow controller master

3.4.2. Data-flow controller slave

Data Flow Controller Slave is designed as per the state machine in the figure 3.6. The main job of DFC Slave is to handle the local data transfer. The local data transfer comprises of 3 data transfer operations which are:

1. **Data flow from WRITE_0 to READ_0**

   Initially, after the system reset, there is no data available in any of the buffers. In this case, there is no need for data to be stored to the RAM as there is enough space in the READ_0 buffer. Hence, direct transfer of data from WRITE_0 to READ_0 occurs.

   Since, the memory location size of both the buffers is same only one free location in READ_0 is required to initiate the data transfer. Hence, the full signal (R_0_FULL) of READ_0 and empty signal (W_0_EMP) of WRITE_0 are used to make the decision.

   LCL_DATA_IN_RAM is a signal driven by empty signals of the current channel WRITE_1, current channel READ_1, common WRITE_2 and data available in RAM signal (GBL_DATA_IN_RAM) of the current channel memory slice from MIGC. LCL_DATA_IN_RAM signal assures that no data is present at any other location.

2. **Data flow from WRITE_0 to WRITE_1**

   Only after READ_0 runs full, there arises a need to transfer the data to RAM. To achieve this, at first, data should be transferred from WRITE_0

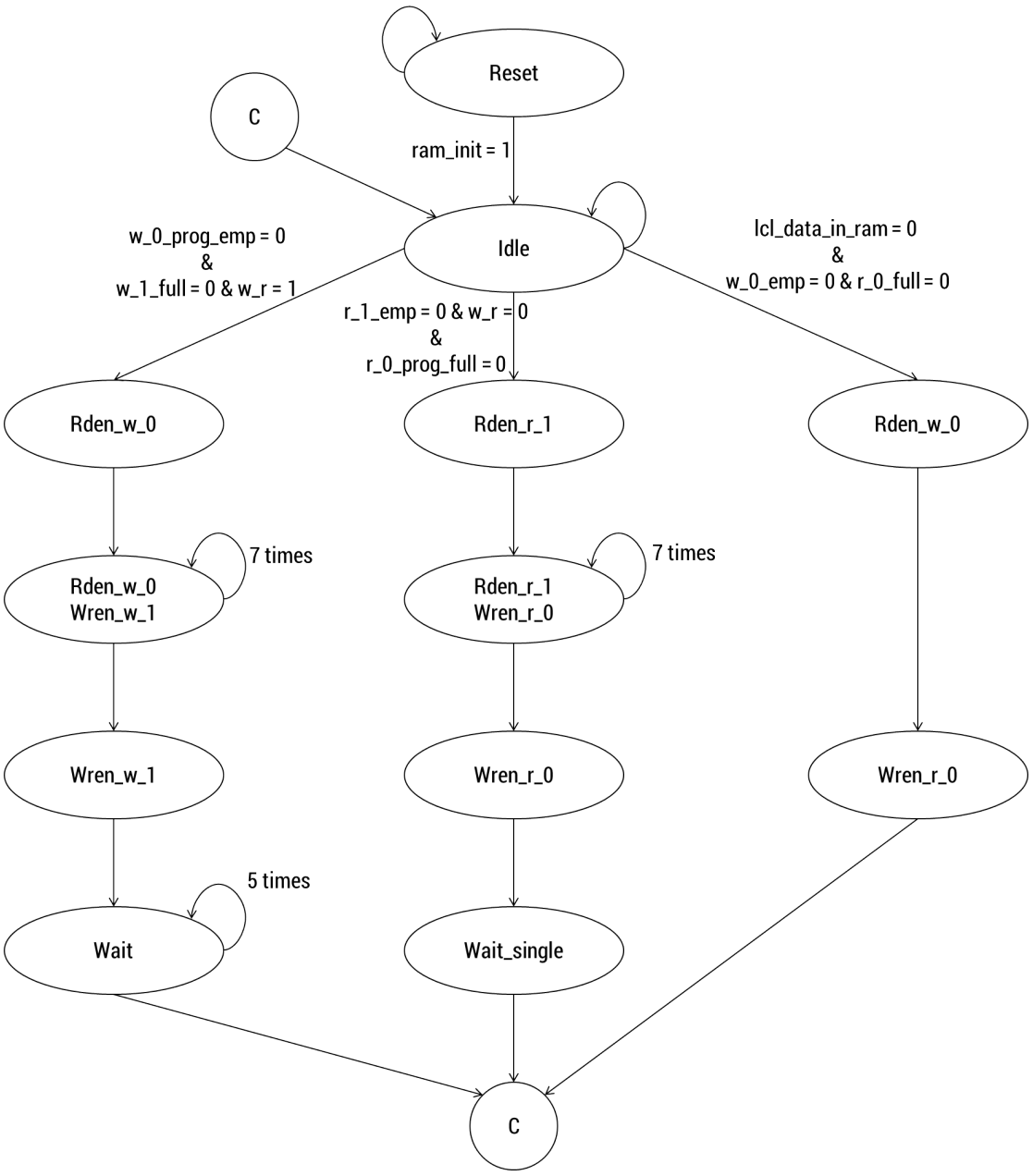

Figure 3.6.: Finite state machine for data-flow controller slave

to WRITE_1.

Memory location size of WRITE_0 is 64bits while that of WRITE_1 is 512bits. Data stored in 8 memory locations from WRITE_0 needs to be transferred to WRITE_1. Hence, instead of empty signal (W_0_EMP), programmable empty signal (W_0_PROG_EMP) is used which assures there are at least 8 memory locations filled in WRITE_0.

W_R is a signal which stores the information about the next operation to be performed and keep alternating between write and read. The operation between WRITE_0 and READ_0 has the highest priority and the other two operations are given equal priority so as to confirm that the data written to the WRITE_0 reaches RAM via WRITE_1 and data read from RAM to READ_1 is read by READ_0.

The extra wait states are required to give time to the FIFOs to update its full, programmable full, empty and programmable empty signals. Detailed information regarding the FIFOs is mentioned in chapter 4.

3. **Data flow from READ_1 to READ_0**

   In this operation, memory location size of READ_1 is 512bits and that of READ_0 is 64 bits. Hence, data once read from READ_1 has to occupy 8 memory locations in READ_0. Hence, the programmable full signal (R_0_PROG_FULL) of READ_0 is used instead of full signal (R_0_FULL).

3.5. Memory interface generator controller

Memory Interface Generator Controller (MIGC) is designed using the finite state machine mentioned in figure 3.7. The objective of MIGC is to handle the read and write requests to the Memory Interface Generator (MIG).

RAM_INIT denotes the PHY_INIT_DONE signal coming from the MIG. This is the most important signal which is being used by all the finite state machines. RAM_INIT indicated that the external RAM has been successfully calibrated. Hence, no operation should be carried out until the RAM_INIT signal is asserted.

MIG_RDY is the second most important signal after the signal RAM_INIT. The process of writing to the RAM or reading from the RAM should not move forward unless the MIG is ready to handle requests. Write and read operations are given equal priority. At the end of any one of the operation, the chance is given for the other operation to execute. The conditions for the state transitioning are defined in such a way that the above mentioned rule is followed.

Write operation starts if there is any data present in the common WRITE_2 buffer and MIG is ready to process the data and that the write bus of the MIG is also available. These conditions are verified by the empty signal (W_2_EMP) of WRITE_2, MIG_RDY and MIG_WDF_RDY. The 512bit data is read from the WRITE_2.

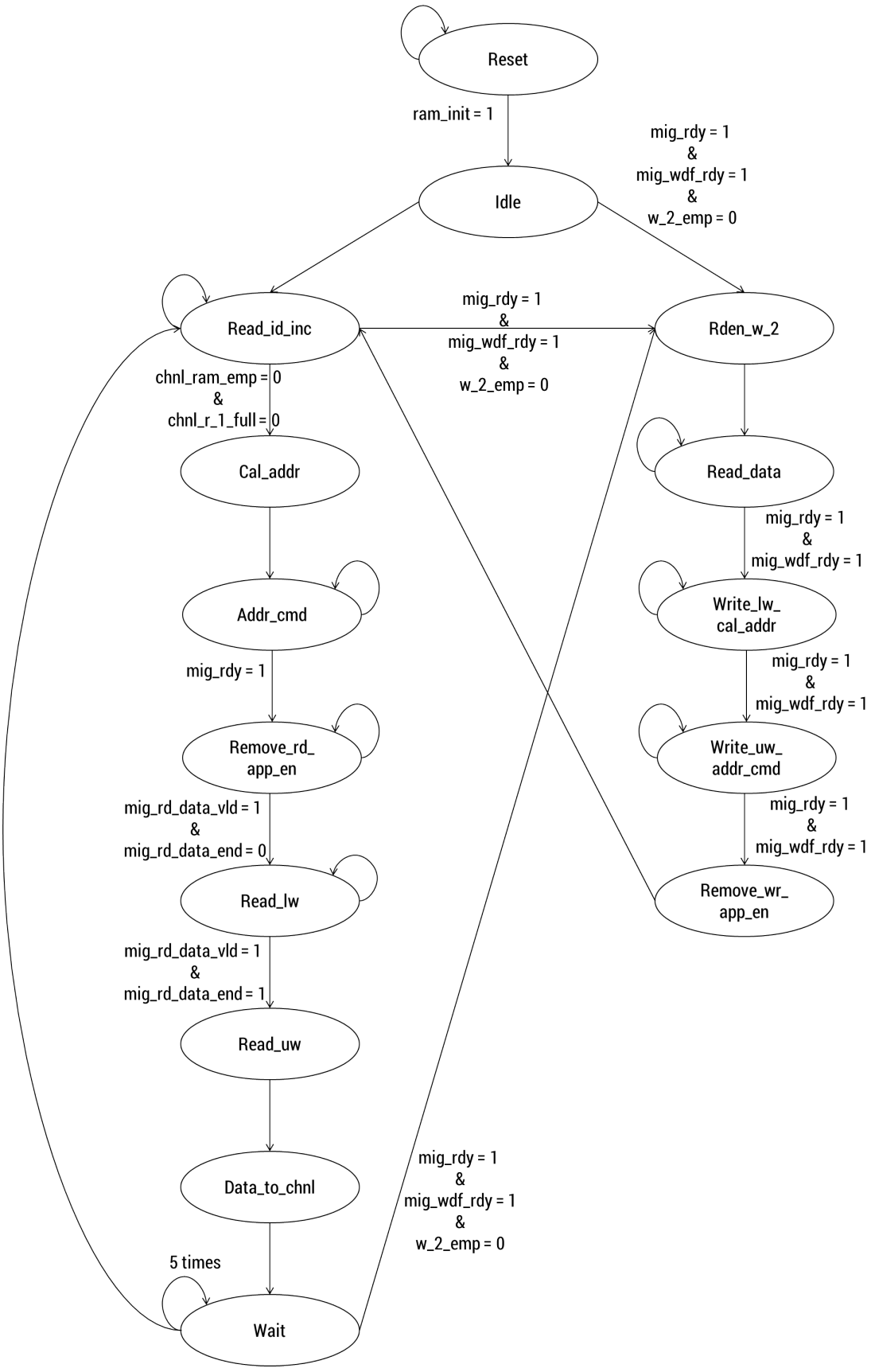

Figure 3.7.: Finite state machine for memory interface generator controller

During this time, the address for the write memory location is also calculated and given along with the write command. MIG_ADDR and MIG_CMD denote the address bus and the command bus while MIG_EN is the signal which is given to indicate the MIG that valid address, command and data is present on the input ports.

For read operation, a round robin manner is implemented. MIGC checks the full signal (CHNL_R_1_FULL) of READ_1 of each channel one after the other and decides if there is free memory location for a data to be read from the RAM and written to the READ_1 buffer of the current channel. Depending upon the channel id the read address is calculated and given along with the read command and enable signal. During write operation 512bits data is split into two words of 256bits and then written. Similar to the write operation, data read from the RAM is also available in two words of 256bits each. The data is indicated by a valid signal (MIG_RD_DATA_VALID) while the last word is indicated by the read end signal (MIG_RD_DATA_END).

The address calculation for both write and read operation is based on the write pointer, read pointer and data count. These 3 values are maintained and stored for each and every channel in an array of composite data type. Also depending on these values and operations, full signal (CHNL_RAM_FULL) and empty signal (CHNL_RAM_EMP) are available which are used by other modules as a deciding factor for their operations.

There are two generics defined for the MIGC and a constant value used for setting different vector sizes. They are as follows:

1. **NO_OF_CHANNELS:** This generic helps to indicate the entity about the total number of channels generated. The channel id starts from 0 to NO_OF_CHANNELS-1. For example, in case NO_OF_CHANNELS is set to 4, the channels have id from 0 to 3. The minimum value that NO_OF_CHANNELS can have is 1 while the maximum value is 32.

2. **DATA_DEPTH:** DATA_DEPTH indicates the number of memory locations of the external memory module reserved for a particular channel. The value set here should be a power of 2. The depth set here is used for all the channels irrespective of their need. The user should ensure that the product of DATA_DEPTH, NO_OF_CHANNELS and memory location width should be less than or equal to the maximum size of the RAM.

3. **VECTOR_WIDTH:** It is a constant that is calculated from the value of the DATA_DEPTH. VECTOR_WIDTH indicates the number of bits required to address all the memory locations for the set DATA_DEPTH. For example, if the DATA_DEPTH is set to 8 memory locations then the VECTOR_WIDTH calculated is 3bits.

The composite data type used for the storage of the address pointers and the data count is as shown below.

```
type channel_info is
record
    chnl_write_ptr  : std_logic_vector (VECTOR_WIDTH-1 downto 0);
    chnl_read_ptr   : std_logic_vector (VECTOR_WIDTH-1 downto 0);
    chnl_data_count : std_logic_vector (VECTOR_WIDTH-1 downto 0);
end record;
```

The write pointer and the read pointer work in sync with each other with the help of the data count to avoid over writing or reading of garbage data. The pointers are standard logic vectors which reset to zero when incremented at its highest values. This feature becomes an advantage in creating cyclic pointers. The whole memory slice for a particular channel can be used in a cyclic manner without any extra effort in maintaining the addresses.

The calculation for the read or write addresses are done based on the corresponding pointer value and the channel value. For example, in case of number if channels is 4 and data depth is 8, the channel id is 2bits long while the channel pointers for write and read are 3bits long. The virtual addresses are channel id and the pointer value concatenated with channel id being the higher bits. Due to this the virtual addresses for the 4 channels are as shown in the table 3.2.

Channel ID	**Start Address**	**End Address**
Channel_0	00000	00111
Channel_1	01000	01111
Channel_2	10000	10111
Channel_3	11000	11111

Table 3.2.: Virtual address range for the multiple channels

These virtual addresses are then converted to physical address by multiplication with 8. This multiplication is necessary as the MIG is set to be used in burst mode BL8.

3.6. Generic multiplexer

Multiplexers are generally used to select one of the inputs and pass it forward depending on the value of the select lines. To select the data output from one of the WRITE_1 buffer of each channel and pass it on as the input to the common WRITE_2 buffer a need for a multiplexer arises.

In the case of a DFM where the number of channels can vary, a generic multiplexer needed to be created. Under normal circumstances when talking about a generic entity the freedom that a developer/user has that he can manipulate the data width of the ports.

In this scenario, the data width of the ports remain the same (i.e. 512bits) but the number of ports itself varies depending on the number of channels. To implement such a functionality, a different type of multiplexer is created where the number of input port is only one 1 quantity but its data width increases in accordance to the increase in the number of channels.

Instead of selecting a particular input from multiple inputs depending on the select lines, here a particular section out of the whole input port is selected and presented at the output port.

4. Realization

This section contains the description of the Virtex-6 FPGA and the Evaluation board utilized for the realization, evaluation and validation. The Xilinx Integrated Software Environment (ISE) was used for design and ISE Simulator (ISim) was used for validation in simulation.

4.1. Hardware

4.1.1. Virtex-6

"Virtex-6 FPGAs are re-programmable integrated circuits designed for the application specific platforms. Virtex-6 FPGAs contain hardware components that enable designers to innovate different products" [Vir15]. Virtex-6 These features assist logic designers to build the high-performance functionality into a FPGA-based system. They are an alternative to ASIC. Virtex-6 FPGAs are best utilized for *"high-performance logic designs, signal processing, and time-critical embedded systems"* [Vir15].

The Virtex-6 FPGA (XC6VLX240T) is used to implement the Data Flow Manager. It contains 241,152 Logic Cells. There are 37,680 Configurable Logic Block Slices. The total amount of on-board RAM is separated into Distributed RAM and Block RAM. The maximum amount of Distributed RAM available is 3,650Kb while the maximum amount of Block RAM is 14,976Kb. Total number of I/O pins is 720 which are distributed in 18 I/O banks. There are 24 GTX low-power transceivers which are used to create different interfaces so as to be able to communicate with the FPGA [Vir15].

4.1.2. HiTech Global development board

HiTech Global development board (HTG-V5-PCIE) contains Virtex-6 FPGA on-board. It also includes a 1GB DDR3 small outline dual in-line memory module (SODIMM), which is used an external memory module for the DFM.

Communication mechanisms available on-board are GTX ports, PCI Express, Ethernet, SFP Interface, USB, USB to UART bridge.

There are three different clock sources which are available on-board. The first clock is a 100MHz oscillator which is being used by the DFM and the external DDR3 RAM. There is a clock socket which can be used to connect external oscillators. There is a third super clock which *"provides variety of low-jitter differential clock through crystals and a frequency synthesizer* [Xilc].

Figure 4.1 contains the image of the development used. There are different input and output ports on the board that are connected to different push buttons or DIP switches and few LEDs. The push buttons are used to provide the necessary starting signals and the different output statuses are verified with the help of the LEDs.

USB to UART Bridge is used to connect the board to the computer through which the data is passed from one and received from the other end. The detailed description of the working of the UART as a means to send and receive data is mentioned in the chapter 5.

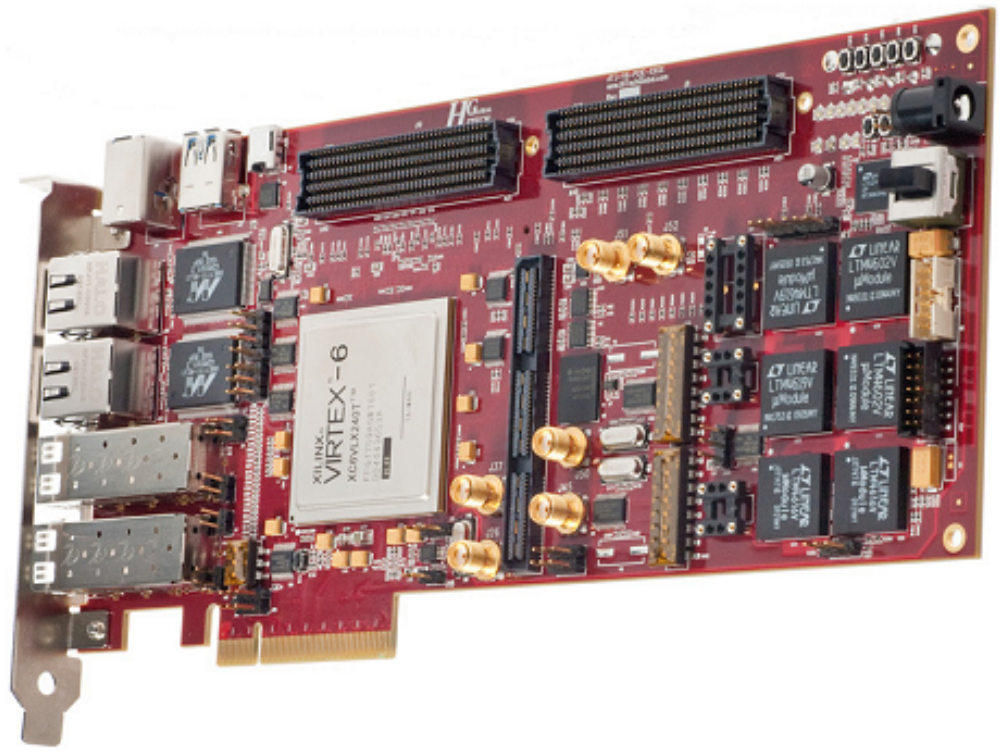

Figure 4.1.: HiTech Global HTG-V6-PCIE Development Board [Xilc]

4.1.3. DDR3 SDRAM

SRAM and DRAM are two major types of RAM that are available today. SRAMs are bigger, faster and much more expensive than DRAMs. Hence, SRAM are utilized where there is requirement of high speed memories and DRAM are used where the amount of data to be stored is higher in quantity. We not only require memory with higher capacity but also a memory which is able to provide the requested data at faster rates. Hence, the modern version of synchronous DRAM i.e. DDR3 SDRAM is used for our purpose.

DDR3 SDRAM is a most widely use DRAM which has a high bandwidth interface. Data rate of DDR3 RAM is approximately twice as compared to DDR2 RAM. It has improved access latencies. It is able to provide better performance than its predecessors. The gap between the data rates of SRAM and DRAM is reducing as the technology behind the making of the RAM is developing. The amount of energy utilized by this RAM is also much less leading to its popularity among the mobile computing platforms and low power applications like use in embedded systems along with microprocessors and FPGAs.

4.2. Toolchain

4.2.1. Xilinx integrated software environment

Xilinx integrated software environment (ISE) is software developed by Xilinx for synthesis of Hardware Description Language (HDL) designs. With the help of ISE, developers attain the capability to synthesize their designs and perform different kinds of analysis and simulation of the design.

ISE is capable of creating dense logic circuit which helps in accommodating bigger systems on a single FPGA chip thereby reducing the overall cost of the project. The designs created by ISE are also much faster and have lower latencies compared to other similar tools [Xil16].

4.2.2. Integrated software environment simulator

Integrated software environment simulator (ISim) is simulation software available along with Xilinx ISE. It helps the developer to verify the intended working of the design on a computer without the need of a hardware device. It helps in understanding the logical behaviour and timing behaviour. It heavily reduces the time required for analysis, testing and verification. It's one of the key features is that

it supports both VHDL and Verilog [Xilb].

4.2.3. Core generator

Xilinx Core Generator is a tool which helps in reducing the design time of systems by giving access to the pre-designed IPs. It contains a catalogue with IPs for different types of domains like Automotive, Communication & Networking, Digital Signal Processing, Embedded Processing, Memories & Storage Elements, Video & Image Processing, etc.

System developers can make bigger and more complex systems with the use of these readily available design blocks. This not only saves time and money but also makes the system much more efficient [Xila].

4.3. Intellectual property core

4.3.1. First in first out buffer

"The FIFO Generator core is an IPCore developed by Xilinx. It provides the complete function of a first-in first-out memory queue. It can be used in any application that require storage and retrieval of data with high-performance designs. The core provides an enhanced solution for all FIFO configurations with minimum resource usage" [Fif12].

The Xilinx FIFO Generator core has two types of FIFO interfaces: Native and AXI4. The interface used in this thesis is native interface. There are multiple customization options which help in creating a high performance FIFO.

FIFO can be configured to have data width from 1 to 1024bits and data depth up to 4,194,304 words. The input and output ports of the FIFO can be symmetric or asymmetric. The ratio between the input and output ports can be set to any value between 1:1, 1:2, 2:1, 1:4, 4:1, 1:8 and 8:1 [Fif12].

The kinds of FIFOs used in the DFM are with the ratios 1:1, 1:8 and 8:1. Write and read characteristics of asymmetric FIFOs is different from that of symmetric FIFOs. Some of these characteristics are mentioned in the following sections.

Figure 4.2 shows the Native Interface FIFOs Signal Diagram. It contains all the possible signals that are related to the FIFO.

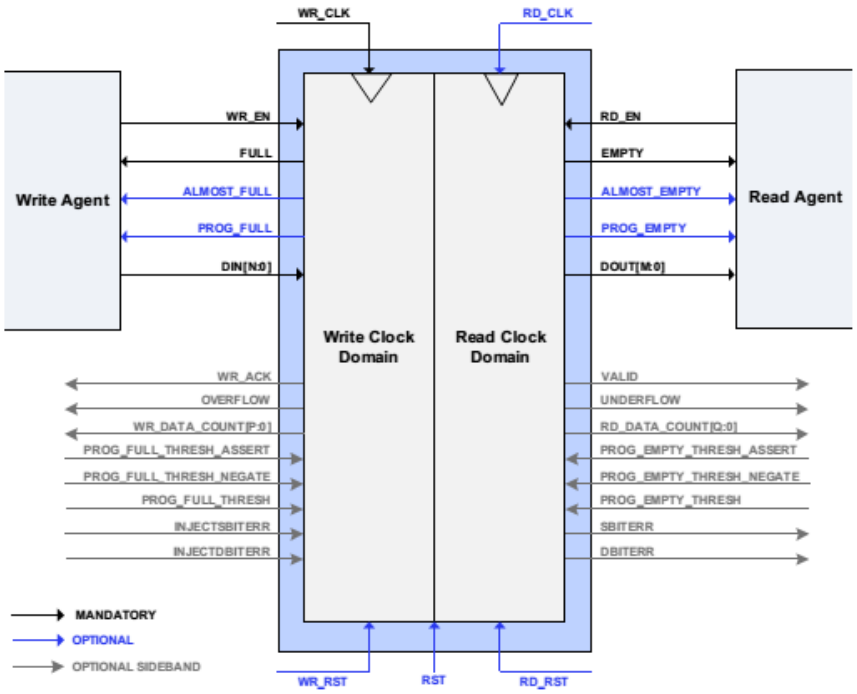

Figure 4.2.: Native interface FIFO signal diagram [Fif12]

The table 4.1 consists of the description of all the important ports related to the FIFO.

Name	**Type**	**Description**
RST	Input	Asynchronous reset
WR_CLK	Input	Synchronous clock with the read signals
RD_CLK	Input	Synchronous clock with the write signals
DIN [N:0]	Input	Data input to the FIFO
WR_EN	Input	Write enable for data on DIN to be written into the FIFO
RD_EN	Input	Read enable for data to be read from FIFO via DOUT
DOUT [N:0]	Output	Data output from the FIFO
FULL	Output	FIFO is completely filled
ALMOST_FULL	Output	Only one more data can be written to the FIFO
PROG_FULL	Output	FIFO is filled up to or more than the threshold
WR_ACK	Output	Previous write to the FIFO was successful
OVERFLOW	Output	Previous write was unsuccessful as the FIFO is full
EMPTY	Output	FIFO is completely empty
ALMOST_EMPTY	Output	Only one more data can be read from the FIFO

PROG_EMPTY	Output	FIFO is empty up to or beyond the threshold
VALID	Output	Data available at DOUT is valid
UNDERFLOW	Output	Previous read was unsuccessful as the FIFO is empty
WR_DATA_COUNT [C:0]	Output	Count of the data written to the FIFO
RD_DATA_COUNT [C:0]	Output	Count of the data that can be read from the FIFO

Table 4.1.: Interface signals of FIFO with independent clock [Fif12]

4.3.2. Asymmetric FIFO

Asymmetric FIFOs can only be implemented with independent clocks and block RAM. *"For FIFOs with asymmetric aspect ratios, the full and empty flags are inactive until a complete word has been written or read. The FIFO does not sanction access to partial words"* [Fif12].

Figure 4.3 shows the operation of a FIFO with aspect ratio of 1:4 (write width=2, read width=8). For a 1:4 FIFO, 4 write operations need to be performed before a single read operation can be performed. Here, 4 2bit data is written one after the another in order 01, 00, 11 and 10. The memory location gets filled up from most significant bit(MSB) to least significant bit(LSB). Now, when a read operation is performed, the data read is 01_00_11_10 or 4E.

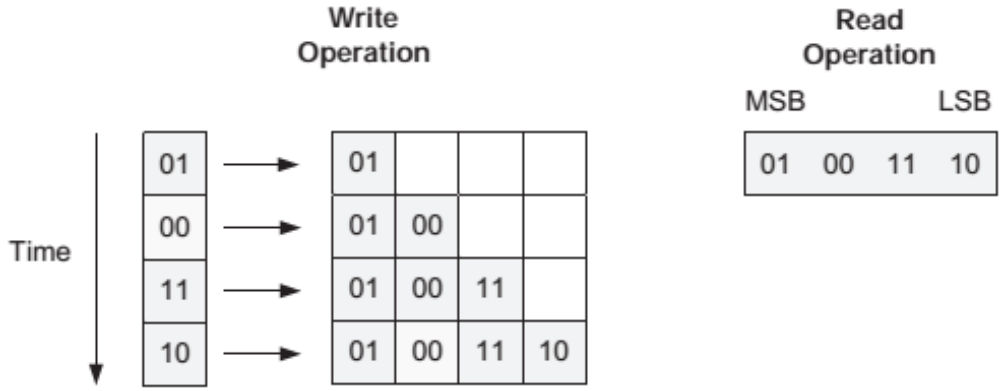

Figure 4.3.: 1:4 Aspect ratio FIFO data ordering [Fif12]

Figure 4.4 shows the transitions of different signals related to the write and read operations explaining the working of the 1:4 FIFO.

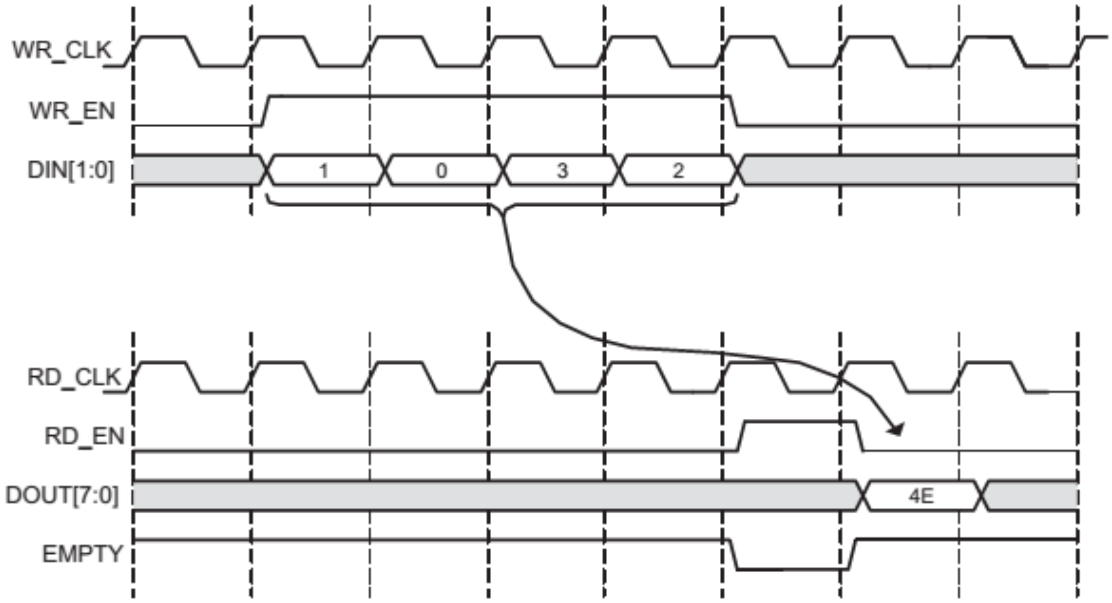

Figure 4.4.: 1:4 Aspect ratio FIFO status flag behaviour [Fif12]

Figure 4.5 shows the example of a FIFO with aspect ratio of 4:1(write width=8, read width=2). Here a single write operation is performed with the data 11_00_01_11 or C7. When the first read operation is executed, the data from the MSB is received first i.e. 11. This is followed by 00, 01 and 11 for the consecutive read operations.

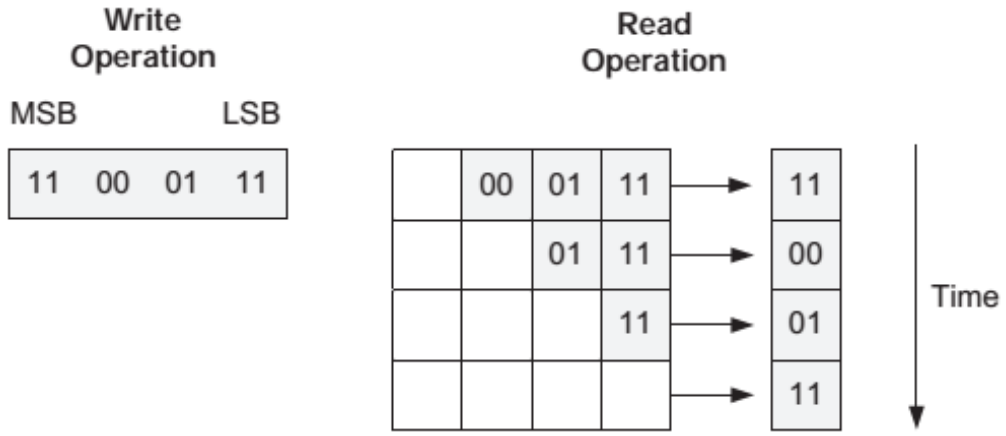

Figure 4.5.: 4:1 Aspect ratio FIFO data ordering [Fif12]

Figure 4.6 shows the transitions of different signals related to the write and read operations explaining the working of the 4:1 FIFO.

The latencies of the different status flags vary as compared to its counterpart in

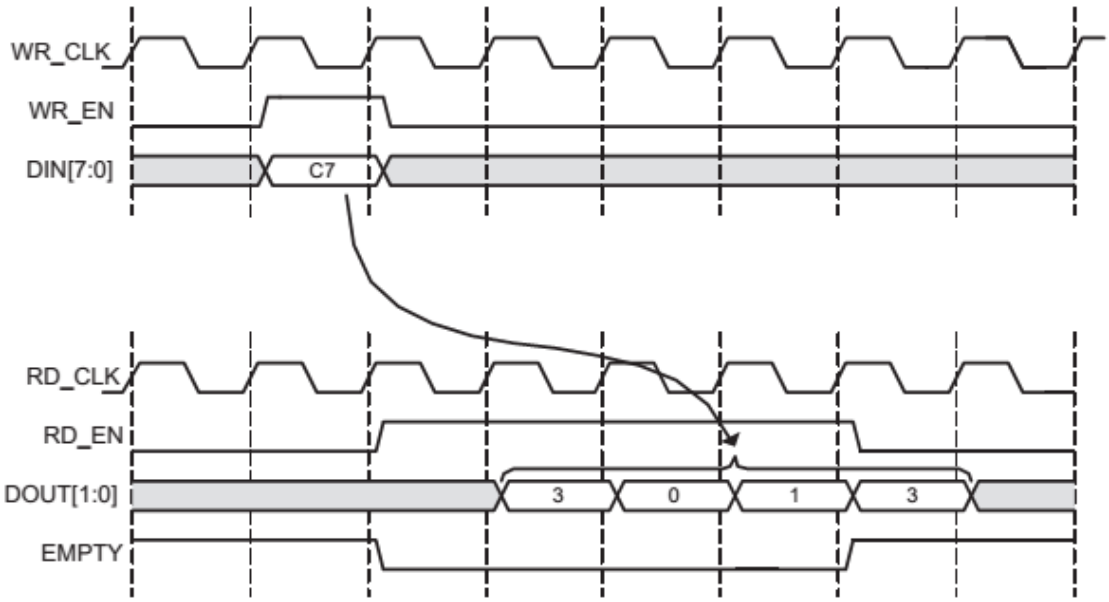

Figure 4.6.: 4:1 Aspect ratio FIFO status flag behaviour [Fif12]

case of symmetric aspect ratios. The tables below contain the exact latencies for different scenarios with respect to the read clock (RD_CLK) and write clock (WR_CLK).

Signal	**Latency**
FULL	0
ALMOST_FULL	0
PROG_FULL	1 WR_CLK
WR_ACK	0
OVERFLOW	0
WR_DATA_COUNT	1 WR_CLK

Table 4.2.: Write flags update latency due to a write operation [Fif12]

Signal	**Latency**
EMPTY	0
ALMOST_EMPTY	0
PROG_EMPTY	1 RD_CLK
VALID	0
UNDERFLOW	0
RD_DATA_COUNT	1 RD_CLK

Table 4.3.: Read flags update latency due to a read operation [Fif12]

Signal	Latency
FULL	1 RD_CLK + 4 WR_CLK (+ 1 WR_CLK)
ALMOST_FULL	1 RD_CLK + 4 WR_CLK (+ 1 WR_CLK)
PROG_FULL	1 RD_CLK + 5 WR_CLK (+ 1 WR_CLK)
WR_ACK	N/A
OVERFLOW	N/A
WR_DATA_COUNT	1 RD_CLK + 4 WR_CLK (+ 1 WR_CLK)

Table 4.4.: Write flags update latency due to a read operation [Fif12]

- The crossing clock domain logic in independent clock FIFOs introduces a WR_CLK uncertainty to the latency calculation.
- Write handshaking signals are not affected by a read operation.

Signal	Latency
EMPTY	1 WR_CLK + 4 RD_CLK (+ 1 RD_CLK)
ALMOST_EMPTY	1 WR_CLK + 4 RD_CLK (+ 1 RD_CLK)
PROG_EMPTY	1 WR_CLK + 5 RD_CLK (+ 1 RD_CLK)
VALID	N/A
UNDERFLOW	N/A
RD_DATA_COUNT	1 WR_CLK + 4 RD_CLK (+ 1 RD_CLK)

Table 4.5.: Read flags update latency due to a write operation [Fif12]

- The crossing clock domain logic in independent clock FIFOs introduces a RD_CLK uncertainty to the latency calculation.
- Read handshaking signals are not affected write operation.

The actual FIFO depth is also one of the characteristics of the FIFO which is different in case of asymmetric FIFOs as compared to a symmetric FIFO. The actual depth of the FIFO depends on the 3 factors [Fif12]:

1. Common Clock or Independent Clock.
2. Standard or First Word Fall Through (FWFT)
3. Symmetric or Asymmetric Aspect Ratio

In case of the asymmetric FIFOs used in the DFM the actual depths of the FIFO is calculated as [Fif12]:

1. actual_write_depth = gui_write_depth – 1

2. actual_read_depth = gui_read_depth – 1

Due to this change in the characteristic, the possible total data written to a 1:8 FIFO is different than the possible total data read. Hence, to keep a balance instead of full signal (W_FULL), programmable full signal (W_PROG_FULL) is used in the finite state machines for the decision making.

4.3.3. Memory interface generator

Memory interface generator (MIG) is a pre-built logic block, available as an IP-Core for use by the system designers. The version of the generated MIG is 3.92. The main task of the MIG is to be a mediator for the user design and the RAM. It translates and routes the appropriate commands and data between the user design and the RAM. MIG contains three different types of interfaces namely AXI4 Slave Interface, Native Interface and User Interface.

AXI4 Slave Interface: Advanced extensible interface (AXI) is a protocol introduced by ARM Advanced Microcontroller Bus Architecture(AMBA), now adopted by Xilinx as a protocol for its IP cores from Spartan-6 and Virtex-6 series onwards [Mig13].

Native Interface: Native interface gives a much higher degree of control over the communication between the user design and the RAM. It is an interface used by advanced users to achieve the most efficiency out of the system. Many communications are carried out in parallel and data may be returned out of order. It's the designer's job to reorder the data [Mig13].

User Interface: User interface is the simplest interface which can be used for data transfer between the user design and the RAM. The table 4.6 consists of all the ports related to the user interface of MIG.

Name	Type	Description
rst	Input	Active high reset
clk	Input	Clock
phy_init_done	Output	Signal indicating the RAM is calibrated
app_addr	Input	Address for the current request
app_cmd	Input	Command for the request (Read: 001, Write: 000)
app_en	Input	Enable for the beginning of the request
app_rdy	Output	Signal indicating MIG is ready for further requests
app_rd_data	Output	Data to the user design
app_rd_data_end	Output	Indicates the last read cycle
app_rd_data_valid	Output	Read data available is valid

app_wdf_data	Input	Data from the user design
app_wdf_end	Input	Indicates the last write cycle
app_wdf_wren	Input	Enable signal for Write data
app_wdf_rdy	Output	Signal indicating MIG is ready for write request

Table 4.6.: Memory interface generator user interface signals [Mig13]

The figure 4.7 represents the MIG in User Interface mode [Mig13].

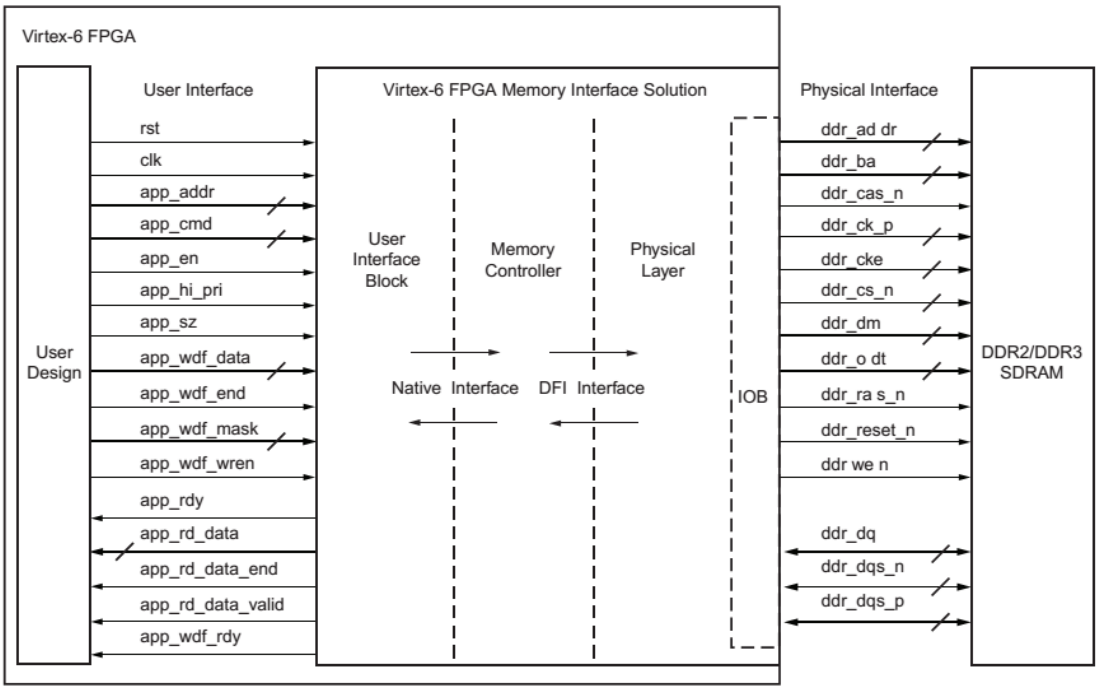

Figure 4.7.: User interface Virtex-6 FPGA memory interface solution [Mig13]

4.3.3.1. Command Path

Figure 4.8 shows the timing diagram for the signal transitions for sending a command to the MIG. The app_en signal is asserted to indicate a valid command. The appropriate command and the address are written to the corresponding buses along with the app_en signal. But the command is only registered after the app_rdy signal gets high. The developer needs to hold the values to the app_cmd, app_addr and app_en until the app_rdy signal is asserted.

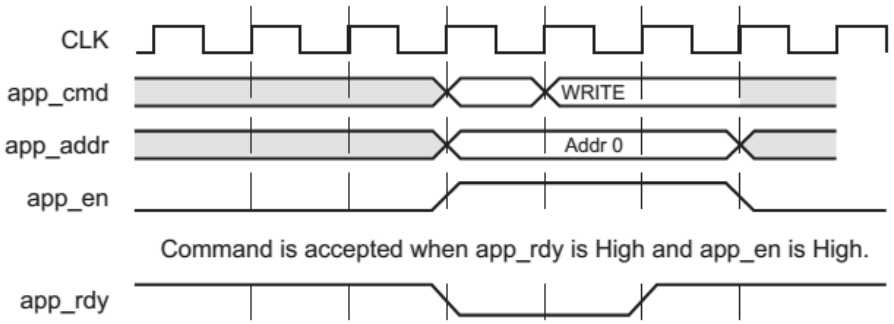

Figure 4.8.: Memory interface generator command timing diagram [Mig13]

4.3.3.2. Write Operation

Similar to Command Path, the write data also depends on enable signal (app_wdf_wren) and the ready signal (app_wdf_rdy). The data for the MIG in burst mode BL8 is written in two words. The app_wdf_rdy should be held high along with the app_wdf_wren signal for both the words. The second data word is indicated by app_wdf_end signal.

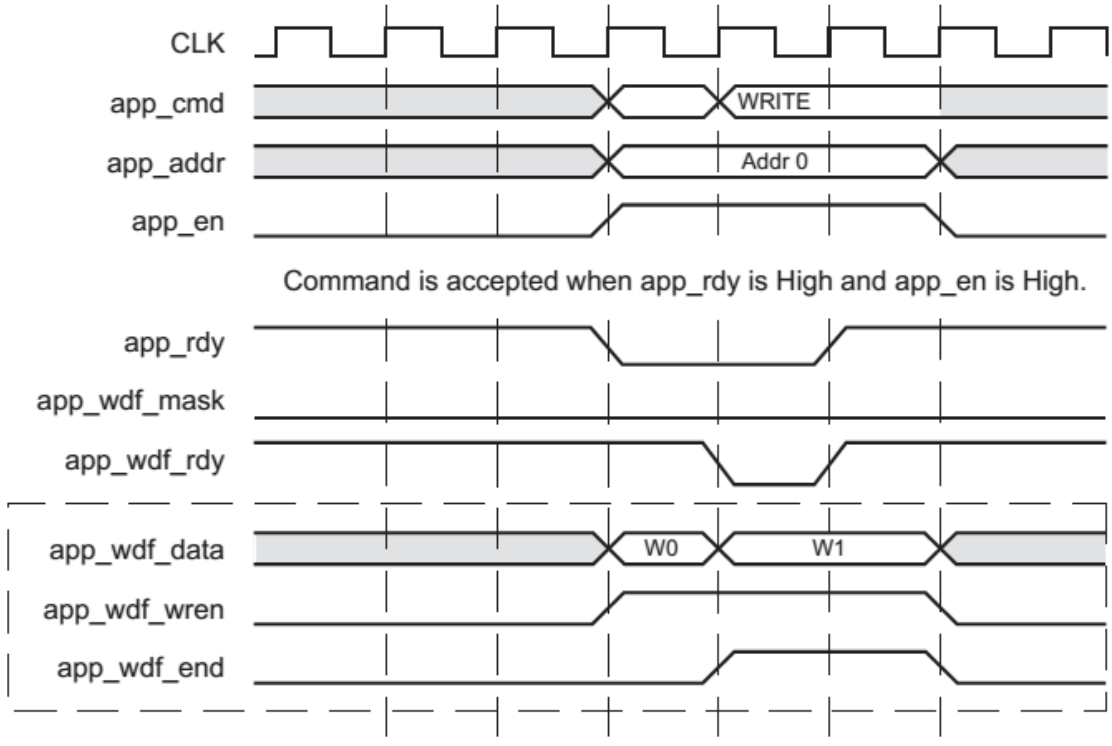

Figure 4.9.: Memory interface generator write timing diagram [Mig13]

The writing of data to the write bus can be in 3 time events as shown in figure

4.10. The write data can be given along with the command as shown in event 1. It can also be given before the command as shown in event 2. The data can be given after the command but in this case the difference between the command and first data word should not be more than 2 clock cycles. This is depicted in event 3 of the figure.

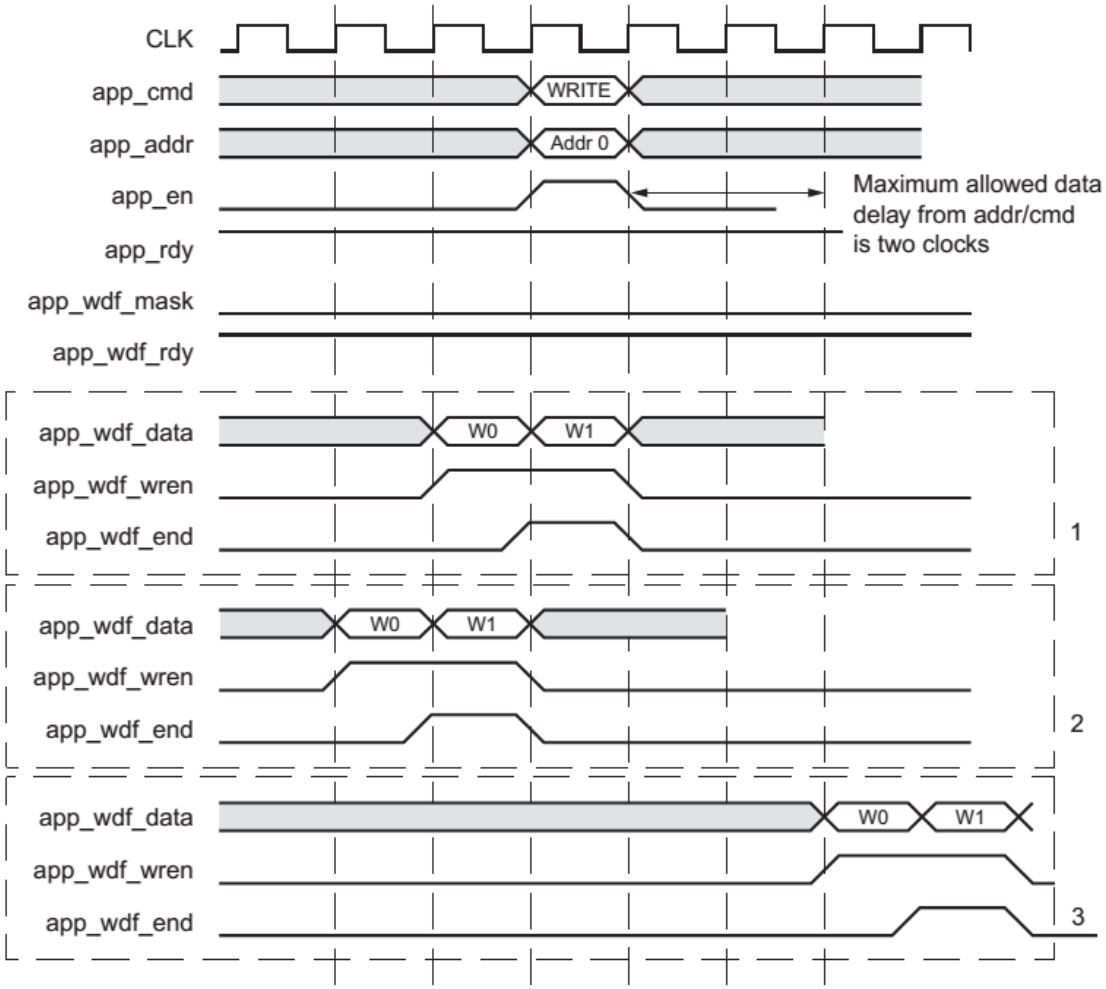

Figure 4.10.: Memory interface generator write data with respect to command time events [Mig13]

4.3.3.3. Write Burst Mode

The MIG has the feature of writing data in the burst mode. For BL8, 8 continuous write commands can be given to the MIG one after the other consecutively. The data on the write bus should be in the same order as the commands and the addresses. Here, the difference between the write data and the command can be more than 2 clock cycles. Figure 4.11 portrays the timing diagram for the burst mode.

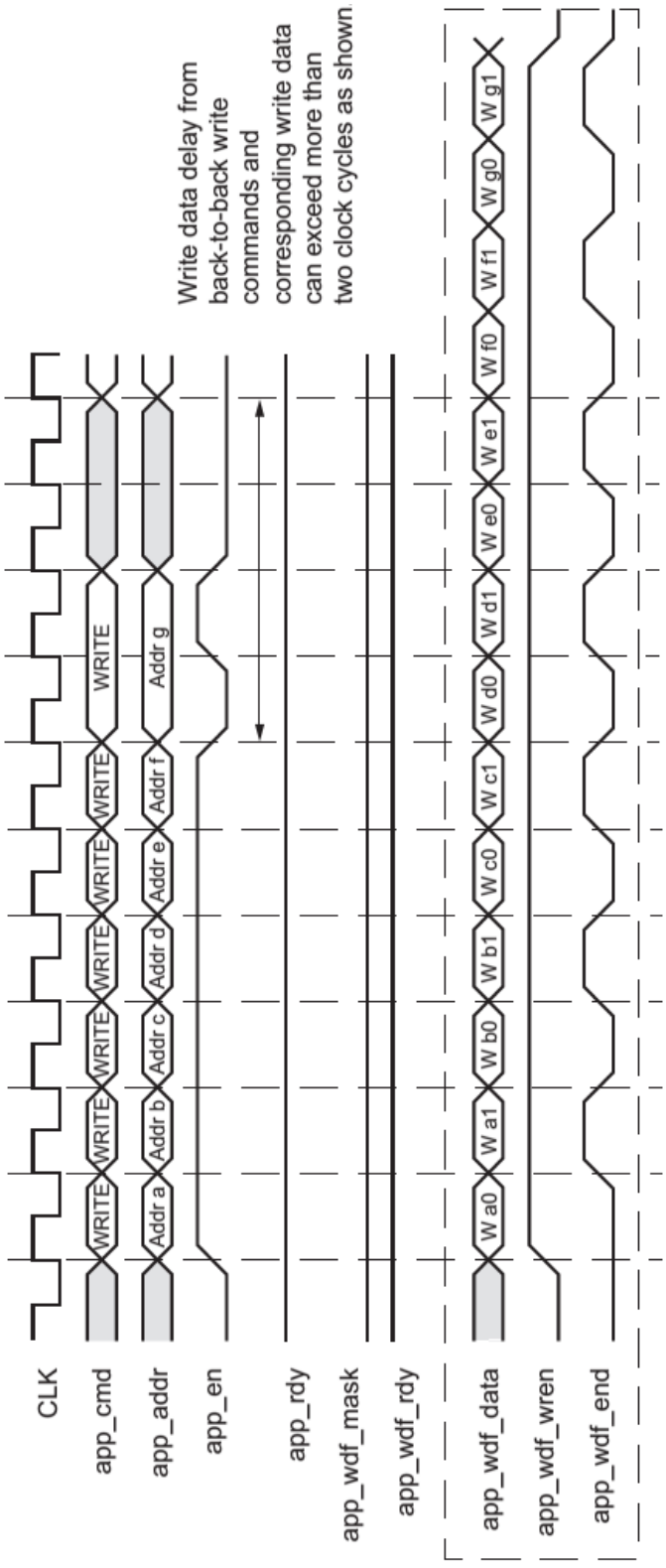

Figure 4.11.: Memory interface generator write data in burst mode BL8 [Mig13]

4.3.3.4. Read Operation

After a read command is issued, the MIG indicate the read data with the assertion of the read data valid signal (app_rd_data_valid). The data received from the MIG is also in two data words. The second valid word is indicated by the app_rd_data_end signal. Figure 4.12 shows the timing diagram for the read operation.

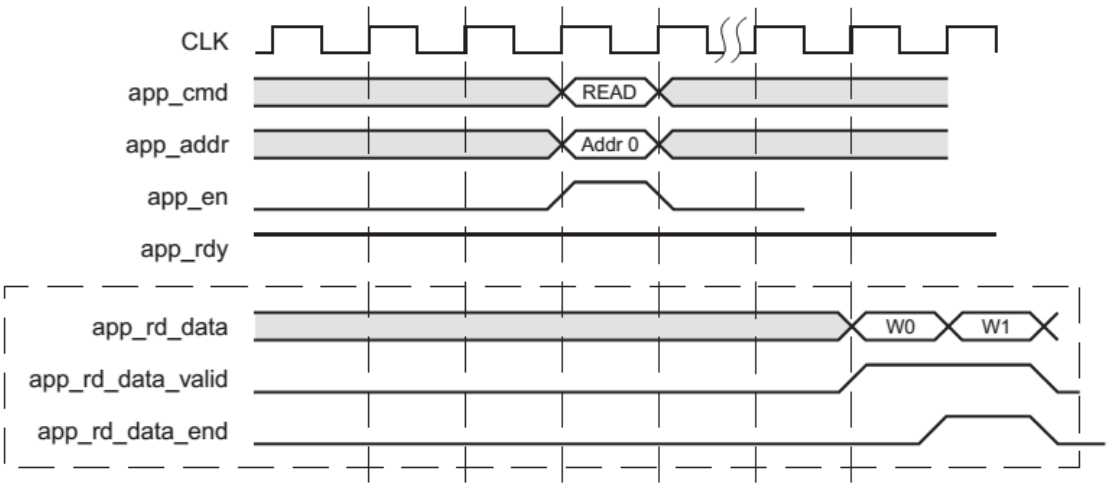

Figure 4.12.: Memory interface generator read timing diagram [Mig13]

5. Evaluation and Validation

5.1. Test scenarios

To evaluate and validate the function of the DFM different test are supposed to be conducted. The objective of these tests is to verify that the data fed to the input of the DFM is received successfully at the output. The data received is verified to be in the proper order. The following test scenarios help in verifying the different work conditions of the DFM.

1. Direct data path from WRITE_0 to READ_0 for one channel.

   For this test, the amount of data stream given at the input is just long enough such that most of the data directly flows from the WRITE_0 to READ_0. In no way should the READ_0 buffer run full to avoid the data going from WRITE_0 to WRITE_1. This process is repeated multiple times with time gaps in between. Repeated data stream helps in verifying that in normal working case if the data stream at the input is available in bursts then the DFM is able to route the data properly through the direct path always if no data is present in any of the other buffers or the external memory module.

2. Data path from WRITE_0 to READ_0 via the external RAM for one channel.

   The data stream given as the input to test this scenario is very long. The data flows through all the buffers and the external RAM. The objective of this test scenario is to verify that the data path has no leaks and all the data that is written to the external RAM is also read from the RAM and received at the output. The same process is repeated multiple times with variable data stream length and with time gaps in between. The long data stream length verifies the data path via the RAM while the short data streams in between helps to regression test the working of the DFM already tested in the first test scenario.

3. Working of simultaneous multiple channels.

   For this test, the number of channels is set to 2. The data stream is fed to the input ports of both the channels and the output received is verified. The main objective of this test is to verify the working of multiple channels

in parallel with respect to one another. This also verifies the round robin working of the DFC Master and the MIGC. The output verifies that all the channels get the chance to write its data to the external RAM and read the data from it. The data is then sent in variable stream lengths and in variable channel order and the data received is verified.

To test the above mentioned scenarios a test environment is created. This test environment helps in sending variable data stream via the USB to UART Bridge on the development board and receiving the output data which can be verified on the computer.

5.2. Data-flow manager tester

The figure 5.1 shows the basic architecture of test environment used to evaluate and validate the functioning of the DFM. Since the DFM has a feature to have variable channels from a range of 1 to 32, a similar architecture for the test environment is necessary so as to incorporate the working of the entire system.

Data flow manager tester (DFMT) is an entity which has the same feature to increase its number of data feeding and receiving channels in the range of 1 to 32. The test environment is designed in such a way that it receives its input data from the computer through a UART interface and the processed data is sent back through the same interface.

RS232 debugger is a module which receives the serial data from the computer and buffers it. This data is read and depending on the channel id this data is passed on to the input buffer (I/P FIFO) of the fast processing module (FPM) of that particular channel with the help of the UART controller. This process is repeatedly done with different channel ids and all the input buffers are filled. All the FPMs start processing when the user gives a signal with a push button on the development board.

The empty signals of all the buffers from the first channel are connected to the LEDs. The statuses of the LEDs indicate whether data has been written to the buffer or not. When the complete processing of the data is complete, the output is stored in the output buffer (O/P FIFO) of the corresponding channel.

Now the read command along with the channel id is given to the DFMT via the computer. Depending on the channel id, UART controller reads the data from the corresponding O/P FIFO and passes it onto the RS232 debugger which then transmits it to the computer.

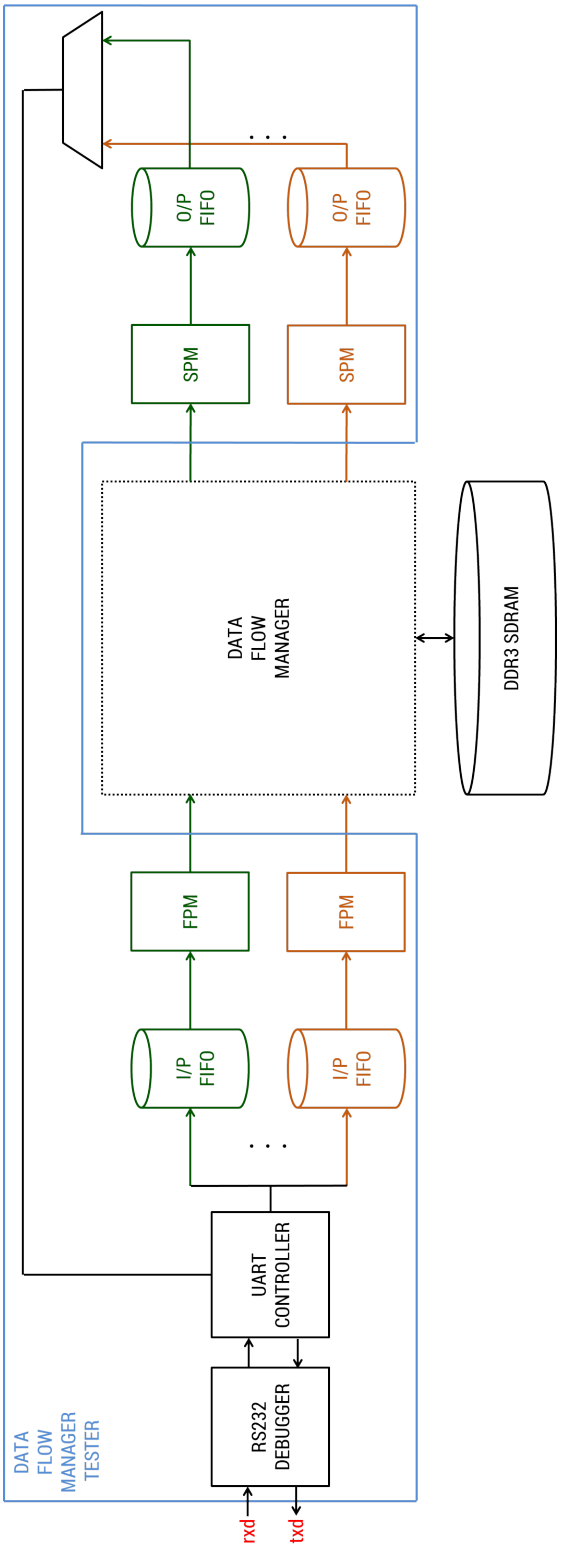

Figure 5.1.: Architecture of the data-flow manager tester

The data transmission and reception on the computer is handled with the use of any software which can interact with the RS232 port of the computer. The sample syntaxes of the commands used to send the data and receive are as follows:

- WRITE Command

  `{write_cmd}␣{channel_id}␣{data_length}␣{data_byte_stream}`
  `0x57␣0x01␣0x05␣0xaa␣0xbb␣0xcc␣0xdd␣0xee`

- READ Command

  `{read_cmd}␣{channel_id}`
  `0x52␣0x01`

5.3. RS232 debugger

The RS232 debugger is an entity which helps in converting the serial data received on the RXD port to the data format that is understood by the different entities of the test environment. RS232 debugger received the data sent to it from the serial port of the computer and then stores it in a reception buffer. This data can then be read from the buffer and used for further testing. Similarly, there is a buffer for storing the transmission data. The data to be sent out is stored in the transmission buffer. This data is then read by the RS232 debugger and then converted to the appropriate form that is recognizable by the computer and transmitted via the TXD port.

5.4. Fast processing module

Figure 5.2 portrays the finite state machine used to develop the entity of fast processing module (FPM). This state machine is simple and just does the job of reading the data from the I/P FIFO, buffering it for one clock cycle and then passes it out to the DFM. At the start point of DFM this data is written to the WRITE_0 buffer and then further processed.

The state machine starts its processing only at the push of a button by the user. It also keeps a check on the empty signal (IP_FIFO_EMP) of the I/P FIFO and full signal (DFM_CHNL_FULL) of the WRITE_0 buffer of the particular channel available as a port of the DFM itself to assure proper data transfer and avoid any data overflow.

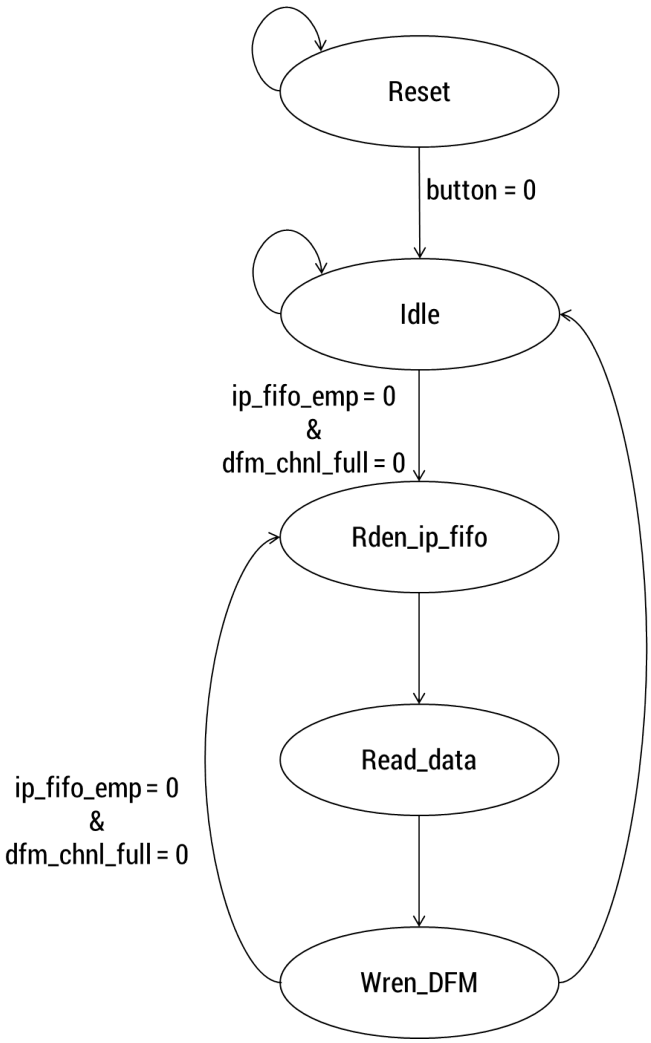

Figure 5.2.: Fast processing module

5.5. Slow processing module

Figure 5.3 describes the finite state machine used to develop slow processing module (SPM). This state machine is also very similar to the state machine for FPM. The job of SPM is to read the data from the READ_0 of the corresponding channel and buffer it for 2 clock cycles before passing it on to the O/P FIFO. It uses empty signal (DFM_CHNL_EMP) of READ_0 of DFM and full signal (OP_FIFO_FULL) of O/P FIFO to initiate the data transfer.

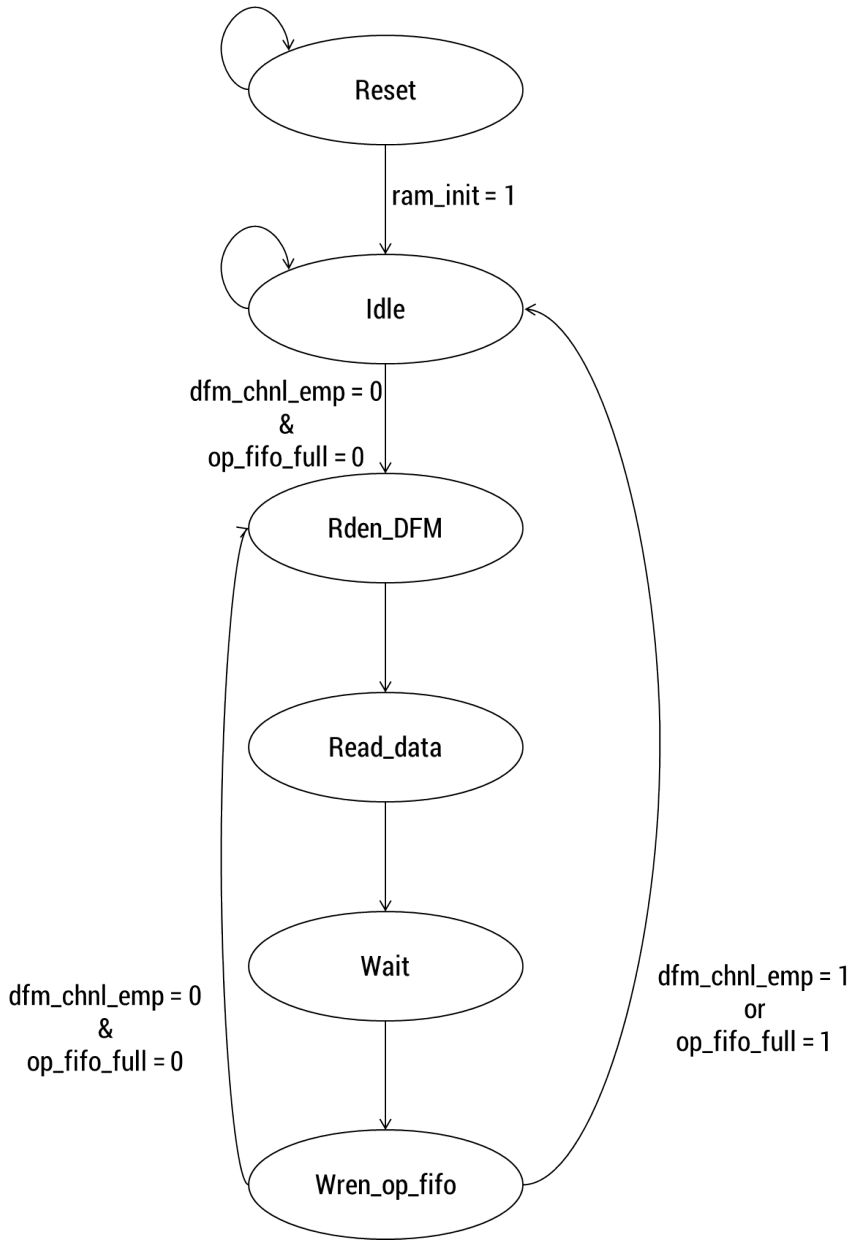

Figure 5.3.: Slow processing module

5.6. UART controller

The UART controller is developed based on the finite state machine in the figure 5.4. The UART controller is responsible for reading the data from the receiving buffer of the RS232 debugger. The data bytes read are processed according to the write and read commands mentioned earlier.

UART controller passes on the data to the appropriate channel depending on the

channel id present in the subsequent byte of the write command. UART controller also controls the multiplexer which reads the data from O/P FIFO of all the channels and writes it to the transmitting buffer of the RS232 debugger.

The receiving buffer of the RS232 debugger is in a first word fall through mode. The main signal available from the RS232 apart from the data is AVAIL signal which helps in determining if valid data is available at the data out port of the RS232. AVAIL signal is the most important signal. Major state transitions occur only on the presence of the AVAIL signal or the system waits in the corresponding wait state.

The data bytes received one by one are matched to the commands at first. If the data byte received is 0x57 (WRITE_CMD) or 0x52 (READ_CMD) then only we move ahead with the next byte. In case of mismatch the new data byte is again compared to the commands. This process goes on until a valid command is received. This command is buffered. The next data byte after the command is always supposed to be the channel id. The information regarding the total number of channels (NO_OF_CHANNELS) is provided as a generic to the state machine. For NO_OF_CHANNELS value equal to Ń; the channel id ranges from 0 to N-1. This data is used to verify if the channel id received is a valid channel id or not.

If a valid read command is received along with a valid channel id, then the UART controller provides the read enable signal to the O/P FIFO of the corresponding channel and reads all the data and writes it to the transmitting buffer of the RS232. This operation keeps repeating till either O/P FIFO is empty or the transmitting buffer is full.

If a valid write command is received along with a valid channel id, then the next byte received is used as the DATA_LENGTH for the amount of data present in the subsequent bytes. This is used in a counter so as to read the amount of data mentioned in the DATA_LENGTH. Depending on the limitation of 8bits, the maximum amount of data possible in a single command is limited to 255. The appropriate empty and full signals were used to assure that there are no chances of data underflow or overflow.

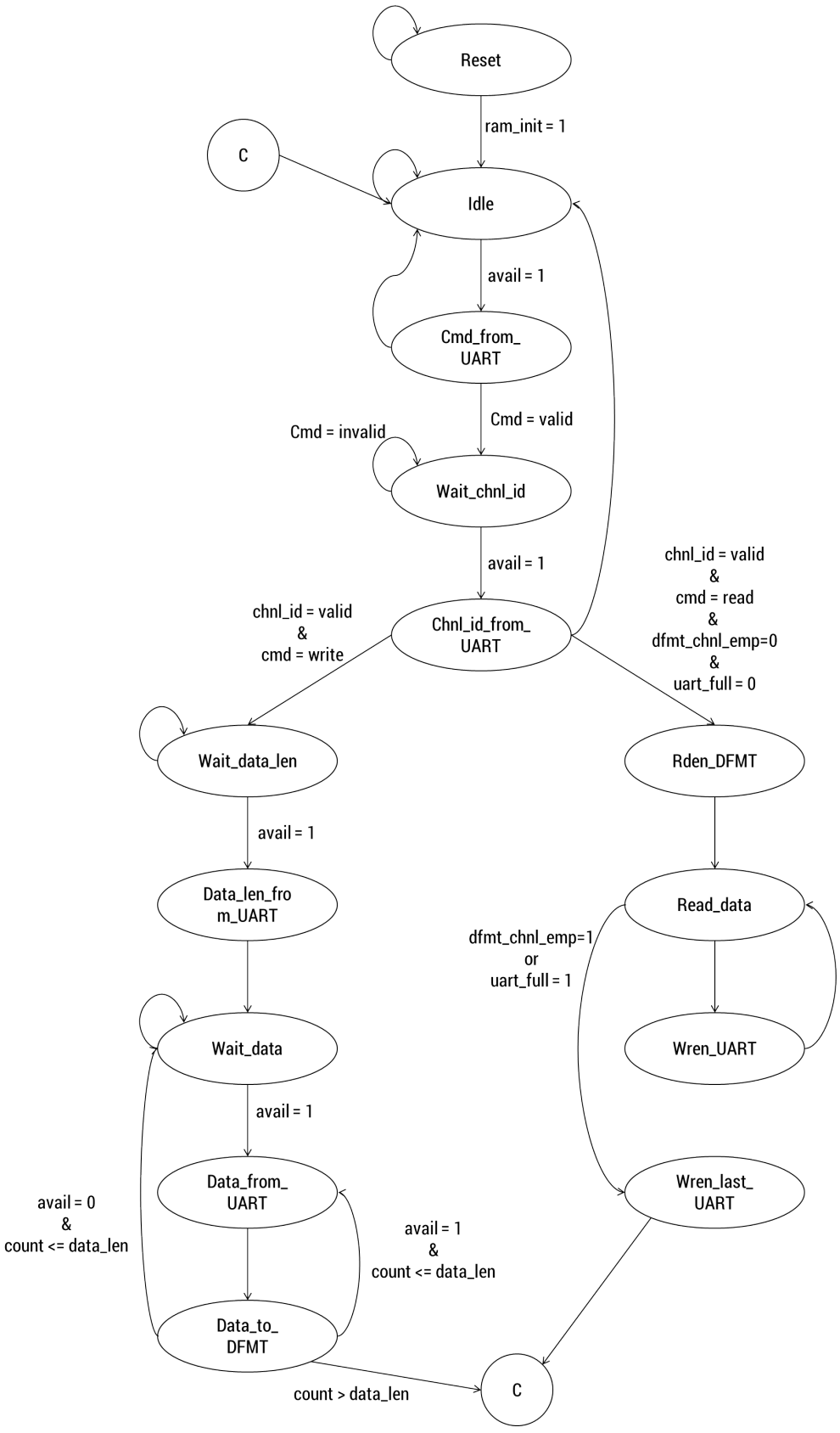

Figure 5.4.: UART controller

5.7. Resource utilization

Since the number of channels of the DFM can be configured, the resource utilization of the DFM will vary with respect to the change in the number of channels. The amount of resources used to implement the DFM increase with the increase in number of channels. With respect to the data in the table 5.1, the utilization of the Slice Registers and LUTs increases gradually with the increase in the number of channels. On the other hand, the usage of Bonded Input Output Blocks (IOBs) and RAM blocks increases at a higher rate with respect to the number of channels.

Due to the high time consumption in the synthesis of the DFM, it is not feasible to synthesize the DFM for higher number of channels. For 5 channels the amount of block RAM used is approximately 50% of the available on the FPGA. The actual data processing entities also require the on-board memory in low quantities. To allow the DFM in blocking the 50% of the resource for itself also does not seem a feasible. Hence, it can be estimated that the maximum number of channels practically should be around 5. To increase the number of channels, complete information regarding the other entities is required and only then can a decision be made to increase the number of channels. The resource utilization of the FPGA with respect to the number of channels is as shown in the table 5.1.

	Number of Channels				
Resources	**1**	**2**	**3**	**4**	**5**
Slice Registers	12327 (4%)	12684 (4%)	13028 (4%)	13371 (4%)	13717 (4%)
Lookup Tables	8057 (5%)	8455 (5%)	8352 (5%)	8683 (5%)	10195 (6%)
Bonded IOB	143 (19%)	163 (22%)	183 (25%)	203 (28%)	223 (30%)
36Kb RAM	54 (12%)	90 (21%)	126 (30%)	162 (38%)	198 (47%)
18Kb RAM	1 (1%)	1(1%)	1(1%)	1 (1%)	1 (1%)

Table 5.1.: Resource utilization of data-flow manager on Virtex-6 XC6VLX240T

5.8. Timing characteristics

The timing characteristics of the FPGA-based system are also one of the major information which helps in deciding the efficiency of the system. The table 5.2 contains the information regarding the timing characteristics of the system with respect to the number of channels.

	Number of Channels				
Parameters	**1**	**2**	**3**	**4**	**5**
Minimum Period (ns)	9.015	8.183	9.586	10.16	8.382
Maximum Frequency (MHz)	110.926	122.205	104.319	98.425	119.303

Table 5.2.: Timing characteristics of data-flow manager on Virtex-6 XC6VLX240T

6. Conclusion

FPGAs have a very unique property when compared to other data processing units like Microprocessors or ASICs. FPGA has the data processing efficiency equivalent to that of ASIC as the development on a FPGA is also application specific. FPGAs also have the ability to be reprogrammed similar to a microprocessor. Hence, FPGAs are utilized in various fields of application.

FPGAs are used in the field of high volume data processing like image or video data exploitation. The amount of memory required to store the data is high. The limited on-board memory of an FPGA is unable to provide the required space. There have been researches in the field of FPGA with an objective to increase the amount of memory with the help of external memory modules.

This master thesis proposes an idea to create a memory controller which can handle the data flow between multiple modules and a single external DDR3 SDRAM module. The Data Flow Manager (DFM) is introduced which helps in buffering the 8bit image data received from one image processing entity and then passing it on to the another entity whenever required.

The DFM is designed in a way that it has the capability to handle multiple pairs (data sender, data receiver) of data processing modules which can be configured before the synthesis. A test environment is developed known as the Data Flow Manager Tester (DFMT) helps in the functional testing of the DFM. The functions of the DFM is tested for 2 channels.

From the section 5.7, it is clear that the amount of on-board resources consumed by the DFM increases to approximately 50% when the number of channels is equal to 5. Implementation of DFM for more channels has to be done only under the condition where the resource utilizations of all the other modules are known. The configuration for the internal blocks can be tweaked to reduce the resource consumption and increase the usable number of channels without affecting the working of other data processing entities.

6.1. Problem in the current work

The data-flow manager in its current configuration and state contains shortcoming and problems. These problems need to be addressed and solved in order to increase the reliability and the efficiency in terms of resource consumption. The current problems in the data-flow manager are:

1. The amount of resources consumed is more due to limit in the data depth selection of the FIFO generator.
2. The hard coded 5bits of channel id leads to unwanted blank space when the number of channels is less.
3. DFM is designed with a particular 8bit data input and output. In this case, the DFM cannot be said to be generic.
4. Equal priorities have been given to the read and write operations in the finite state machine for memory interface generator controller. The priority should be handled depending on the data rate of the channel.
5. The buffer sizes of the RS232 debugger limits the complete data read from the DFM in one read command. This leads to a single byte data loss when the next read command is issued to the same channel.

6.2. Future work

The data-flow manager developed satisfies the requirements mentioned in the section 2.5. But there are certain shortcomings and problems which lead to future room for improvements. Suggestions to solve the problems and to improvise the data-flow manager are mentioned below:

1. Optimize the code further to reduce the resource consumption and resolve any bugs which lead to data loss.
2. Create a better FIFO component (instead of using the FIFO IPCore) with asymmetric feature and no limitations to the data depth selection.
3. Add a data width generic in the FIFO which can be programmatically set by the higher module in the hierarchy to avoid the problem of fixed vector size for channel id.
4. Create generic data width for the input and output ports of the DFM to incorporate greater variety of data processing modules.

5. Create a priority generic in the memory interface generator controller which can be configured before synthesis to help balance the data flow load on both the read and write operations.

6. Optimize the design further by changing the FIFO IPCore with intelligent other buffer modules.

Bibliography

[AHIG09] Kurt Franz Ackermann, Burghard Hoffmann, Leandro Soares Indrusiak, and Manfred Glesner. Providing memory management abstraction for self-reconfigurable video processing platforms. *International Journal of Reconfigurable Computing*, 2009:1–15, 2009.

[Ark] ARK | Your Source for Intel® Product Specifications.

[CHM] Eric S. Chung, James C. Hoe, and Ken Mai. Coram: An in-fabric memory architecture for fpga-based computing.

[Clo] Clock Domain Crossing.

[CXZC14] Ying Chen, Wanpeng Xu, Rongsheng Zhao, and Xiangning Chen. Design of a hardware/software FPGA-based driver system for a large area high resolution CCD image sensor. *Photonic Sens*, 4(3):274–280, jul 2014.

[Dan04a] Klaus Danne. Memory Management to Support Multitasking on FPGA Based Systems. In *In Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReCon*, page 21, 2004.

[Dan04b] Klaus Danne. Operating systems for fpga based computers and their memory management. In *In ARCS 2004 Organic and Pervasive Computing, Workshop Proceedings, volume P-41 of GI-Edition Lecture*. Köllen Verlag, 2004.

[Fif12] LogiCORE IP FIFO Generator v9.3 Product Guide, December 2012.

[For] Peter Forret. 12 Megapixel camera | toolstud.io.

[FP12] Laura Fischer and Yura Pyatnychko. *FPGA Design for DDR3 Memory*. Bachelor Thesis, Worcester Polytechnic Institute, Worcester, Massachusetts, USA, March 2012.

[Fpg] fpga4fun.com - Crossing clock domains.

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

[HE05] S. Heithecker and R. Ernst. Traffic shaping for an fpga based sdram controller with complex qos requirements. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 575–578, June 2005.

[Hei15] Gernot Heiser. How to steal encryption keys: Your cloud is not as secure as you may think!, April 2015.

[HP11] John C. Hoffman and Marios S. Pattichis. A high-speed dynamic partial reconfiguration controller using direct memory access through a multiport memory controller and overclocking with active feedback. *International Journal of Reconfigurable Computing*, 2011:1–10, 2011.

[JJDD14] Jain, Amit Jain, Divyanshu, and Tejas Dave. Synchronizer techniques for multi-clock domain SoCs & FPGAs, September 2014.

[Joh09] Jeff Johnson. Generating Clock Domain Crossing FIFOs | FPGA Developer, September 2009.

[Mig13] Virtex-6 FPGA Memory Interface Solutions User Guide, March 2013.

[OB14] Zvi Or-Bach. FPGAs as ASIC Alternatives: Past & Future | EE Times, April 2014.

[SB12] Wolfram Hardt Stephan Blokzyl, Matthias Vodel. A hardware accelerated real-time image processing concept for high-resolution eo sensors. In *Proceedings of the 61. Deutscher Luft- und Raumfahrtkongress*, Berlin, Germany, September 2012. Deutsche Gesellschaft für Luft- und Raumfahrt.

[TAJZ15] Kevin R. Townsend, Osama G. Attia, Phillip H. Jones, and Joseph Zambreno. A scalable unsegmented multiport memory for FPGA-based systems. *International Journal of Reconfigurable Computing*, 2015:1–12, 2015.

[Tal14] Deepak Kumar Tala. Interfacing Two Clock Domains, February 2014.

[VB13] Wim Vanderbauwhede and Khaled Benkrid, editors. *High-Performance Computing Using FPGAs*. Springer Science Business Media, 2013.

[Vir15] Virtex-6 Family Overview Product Specification, August 2015.

[WK01] G. Wigley and D. Kearney. The first real operating system for reconfigurable computers. In *Computer Systems Architecture Conference, 2001. ACSAC 2001. Proceedings. 6th Australasian*, pages 130–137, 2001.

[Xila] Xilinx CORE Generator System.

[Xilb] Xilinx ISE Simulator (ISim).

[Xilc] Xilinx Virtex-6 PCI Express Gen 2, USB 3.0, SFP+ board.

[Xil16] Xilinx ISE, September 2016. Page Version ID: 739203946.

[ZCL06] Zude Zhou, Songlin Cheng, and Quan Liu. Application of ddr controller for high-speed data acquisition board. In *First International Conference on Innovative Computing, Information and Control - Volume I (ICICIC'06)*, volume 2, pages 611–614, Aug 2006.

Appendices

A. Configuration parameters

A.1. Common configuration parameters

- FPGA family: Virtex-6
- Device: XC6VLX240T
- Package: FF1175
- Speed Grade: -2

A.2. Configuration parameters for MIG generation

- Name: mig_3_92
- Version: 3.92
- Frequency: 400MHz (Max possible 533 MHz)
- Memory Type: SODIMM
- Memory Part: MT4JSF12864HZ-1G4
- Data Width: 64
- Burst Length: 8-Fixed
- Burst Type: Sequential
- Memory Address Mapping: Row+Bank+Column
- System Clock: Single-Ended

A.3. Configuration parameters for WRITE_0 generation

- Name: fifo_w_lvl_0_8_64
- Version: 9.3

- Interface Type: Native
- Read/Write Clock Domains: Independent Clock
- Memory Type: Block RAM
- Read Mode: Standard FIFO
- Write Width: 8
- Read Width: 64
- Write Depth: 128
- Read Depth: 16
- Optional Flags: ALMOST_FULL, ALMOST_EMPTY (Unused)
- Handshaking: WR_ACK, OVERFLOW, VALID, UNDERFLOW (Unused)
- Reset Type: Asynchronous
- PROG_FULL Threshold: 120
- PROG_EMPTY Threshold: 8

A.4. Configuration parameters for WRITE_1 generation

- Name: fifo_w_lvl_1_64_512
- Version: 9.3
- Interface Type: Native
- Read/Write Clock Domains: Independent Clock
- Memory Type: Block RAM
- Read Mode: Standard FIFO
- Write Width: 64
- Read Width: 512
- Write Depth: 128
- Read Depth: 16

- Optional Flags: ALMOST_FULL, ALMOST_EMPTY (Unused)
- Handshaking: WR_ACK, OVERFLOW, VALID, UNDERFLOW (Unused)
- Reset Type: Asynchronous
- PROG_FULL Threshold: 120

A.5. Configuration parameters for READ_1 generation

- Name: fifo_r_lvl_1_512_64
- Version: 9.3
- Interface Type: Native
- Read/Write Clock Domains: Independent Clock
- Memory Type: Block RAM
- Read Mode: Standard FIFO
- Write Width: 512
- Read Width: 64
- Write Depth: 16
- Read Depth: 128
- Optional Flags: ALMOST_FULL, ALMOST_EMPTY (Unused)
- Handshaking: WR_ACK, OVERFLOW, VALID, UNDERFLOW (Unused)
- Reset Type: Asynchronous

A.6. Configuration parameters for READ_0 generation

- Name: fifo_r_lvl_0_64_8
- Version: 9.3
- Interface Type: Native
- Read/Write Clock Domains: Independent Clock

- Memory Type: Block RAM
- Read Mode: Standard FIFO
- Write Width: 64
- Read Width: 8
- Write Depth: 16
- Read Depth: 128
- Optional Flags: ALMOST_FULL, ALMOST_EMPTY (Unused)
- Handshaking: WR_ACK, OVERFLOW, VALID, UNDERFLOW (Unused)
- Reset Type: Asynchronous
- PROG_FULL Threshold: 8

A.7. Configuration parameters for common WRITE_2 generation

- Name: fifo_write_main
- Version: 9.3
- Interface Type: Native
- Read/Write Clock Domains: Common Clock
- Memory Type: Block RAM
- Read Mode: Standard FIFO
- Write/Read Width: 517
- Write/Read Depth: 256
- Optional Flags: ALMOST_FULL, ALMOST_EMPTY (Unused)
- Handshaking: WR_ACK, OVERFLOW, VALID, UNDERFLOW (Unused)
- Reset Type: Asynchronous

A.8. Configuration parameters for I/P & O/P buffer generation

- Name: fifo_test
- Version: 9.3
- Interface Type: Native
- Read/Write Clock Domains: Common Clock
- Memory Type: Block RAM
- Read Mode: Standard FIFO
- Write/Read Width: 8
- Write/Read Depth: 512
- Optional Flags: ALMOST_FULL, ALMOST_EMPTY (Unused)
- Handshaking: WR_ACK, OVERFLOW, VALID, UNDERFLOW (Used and active high)
- Reset Type: Asynchronous

A.9. Steps to configure the Data-flow manager and make it compatible to HiTech Global development board

1. Generate all the IP Cores according to the configuration parameters mentioned above.
2. Changes to "src\data_flow_manager.vhd"
   - Remove "clk_ref" from the "mig_3_92" component declaration and instantiation.
   - Assign "mig_3_92" generic "CLKFBOUT_MULT_F" a value of 16.
   - Assign "mig_3_92" generic "CLKOUT_DIVIDE" a value of 2.
3. Changes to "ip\mig_3_92\mig_3_92\user_design\rtl\ip_top\mig_3_92.vhd"
   - Remove "clk_ref" port.
   - Change constant "SYSCLK_PERIOD" value to 2*tCK*nCK_PER_CLK.

- Change the "clk_ref" mapping in "iodelay_ctrl" instantiation from "clk_ref" to "clk".
- Add input port "pll_lock" to component "iodelay_ctrl".
- Add output port "lock" to component "infrastructure".
- Add signal "mmcm_lock" of type "std_logic" and connect to "pll_lock" and "lock" mentioned above.

4. Changes to "ip\mig_3_92\mig_3_92\user_design\rtl\ip_top\iodelay_ctrl.vhd"
   - Add input port "pll_lock"
   - Assign "clk_ref" to "clk_ref_bufg".
   - Assign "sys_rst_act_hi or (not pll_lock)" to "rst_tmp_idelay" instead of "sys_rst_act_hi".
   - Remove the following code

```
diff_clk_ref: if (INPUT_CLK_TYPE = "DIFFERENTIAL") generate
    u_ibufg_clk_ref : IBUFGDS
        generic map (
            DIFF_TERM     => TRUE,
            IBUF_LOW_PWR  => FALSE
        )
        port map (
            I     => clk_ref_p,
            IB    => clk_ref_n,
            O     => clk_ref_ibufg
        );
end generate  diff_clk_ref;

se_clk_ref: if (INPUT_CLK_TYPE = "SINGLE_ENDED") generate
    u_ibufg_clk_ref : IBUFG
        generic map (
            IBUF_LOW_PWR => FALSE
        )
        port map (
            I   => clk_ref,
            O   => clk_ref_ibufg
        );
end generate se_clk_ref;

u_bufg_clk_ref : BUFG
    port map (
        O  => clk_ref_bufg,
        I  => clk_ref_ibufg
```

```
);
```

5. Changes to "ip\mig_3_92\mig_3_92\user_design\rtl\ip_top\infrastructure.vhd"
   - Add output port "lock".
   - Assignment "pll_lock" to "lock".
   - Divide current value of constant "CLKIN1_PERIOD" by 2.

```
constant CLKIN1_PERIOD  : real
        := real((CLKFBOUT_MULT_F * CLK_PERIOD))/
        real(DIVCLK_DIVIDE * CLKOUT_DIVIDE * nCK_PER_CLK * 1000);


constant CLKIN1_PERIOD  : real
        := real((CLKFBOUT_MULT_F * CLK_PERIOD))/
        real(2* DIVCLK_DIVIDE * CLKOUT_DIVIDE * nCK_PER_CLK * 1000);
```

TECHNISCHE UNIVERSITÄT CHEMNITZ

Studentenservice – Zentrales Prüfungsamt

Selbstständigkeitserklärung

Name: Vorname: geb. am: Matr.-Nr.:	Bitte beachten: 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch bei keinem anderen Prüfer als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: ……. Unterschrift: ……

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.