Masters’s Thesis

High Speed Backbone for FPGA at ESS

Max Andersson
Gabriel Jönsson

Department of Electrical and Information Technology,
Faculty of Engineering, LTH, Lund University, 2016.

High speed backbone for FPGA at ESS

Max Andersson
`elt11man@student.lu.se`
Gabriel Jönsson
`ael10gjo@student.lu.se`

Department of Electrical and Information Technology
Lund University

Advisor: Fredrik Kristensen

January 20, 2016

Printed in Sweden
E-huset, Lund, 2016

Abstract

The purpose of this project is to reverse engineer and re-design a PCI Express communication system which is currently being used at ESS in Lund. The current model is non-modifiable and our goal is to create a system open for customization. The aspects explored are communication between hardware and software using PCI Express, data handling and arbitration, direct memory access and how these can be implemented in hardware. We have successfully re-created the original design with a fully utilized read interface and a significantly slower write interface. The write function has been studied to find possible options to improve the current design. This system will be installed in 150 different parts of the accelerator and although it is a small part, it will be vital for the overall performance.

Acknowledgments

A lot of thanks to the warmth and guidance received from the people around us. Fredrik Kristensen you have committed a lot of time and effort to assist and inspire us during the last five months. Yeimy Lopez and Jessica Ericsson you have given us hope and comfort when everything seemed hopeless. For this we thank you and all of the competent people at ESS that helped us in any way. Urša Rojec, Klemen Strnisa, Maurizio Donna and Niklas Claesson; thank you too for your contributions. We would also like to thank Fredik Rusek for taking the role as examiner.

Table of Contents

1 Introduction **1**
- 1.1 Background 1
- 1.2 Purpose of the project 2
  - 1.2.1 Practical application 2
  - 1.2.2 Goals 3
- 1.3 Thesis structure 6

2 Theory **9**
- 2.1 PCI Express 9
  - 2.1.1 Compability 9
  - 2.1.2 Root complex structure 9
  - 2.1.3 Enumeration 10
  - 2.1.4 PCIe hierarchy 11
  - 2.1.5 TLP - Transaction layer package 12
- 2.2 AXI - Advanced eXtensible Interface 15
  - 2.2.1 Master/Slave 16

3 System **17**
- 3.1 Development tools 17
  - 3.1.1 ISE project navigator 17
  - 3.1.2 iMPACT 17
- 3.2 Debugging tools 17
  - 3.2.1 ISim - ISE simulator 17
  - 3.2.2 Questasim 18
  - 3.2.3 Analyzer 18
- 3.3 Hardware equipment 18

4 Implementation **19**
- 4.1 Architecture overview 19
- 4.2 PCI Express interface 19
  - 4.2.1 Ingress 19
  - 4.2.2 Egress 22
  - 4.2.3 Register file 23

4.2.4 DMA 27
4.3 AXI interconnect - read/write arbiter 28
4.4 Memory interface 29
4.4.1 AXI controller 30
4.4.2 User interface 30
4.4.3 Memory interface controller 30

5 Verification 33
5.1 Test bench 33
5.2 Chipscope 33
5.3 Software tests 33

6 Results 35
6.1 Design resources 35
6.2 Performance 36
6.2.1 Clock frequency 36
6.2.2 Read 36
6.2.3 Write 36

7 Analysis 37
7.1 Conclusions 37
7.2 Lessons Learned 38
7.3 If we were to do it again 40
7.4 Future work 40

References 43

List of Figures

1.1 Overview of the LLRF system setup. 2
1.2 Setup for cavity testing and electric field regulation 3
1.3 First set goal of the project. 4
1.4 Second set goal of the project. 4
1.5 Third set goal of the project. 5
1.6 Fourth set goal of the project. 5
1.7 Fifth set goal of the project. 6

2.1 Example of a PCIe system with a root complex and multiple endpoints in the form of one memory, two PCIe endpoints and a PCI/PCI-X bus which can be connected with multiple peripherals which are compatible with older bus standards. 10
2.2 Simple representation of a link between the layers of two PCIe devices. 11
2.3 The contents of a TLP. The parts added by the individual layers are marked with dotted lines. The Data and ECRC data blocks are optional dependent on the purpose of the package. 12
2.4 The first header present in all TLPs [5] 13
2.5 The TLP header with the additional memory request header. 14
2.6 The TLP header with the additional completion header. 14
2.7 AXI communication between master and slave. 15

3.1 SIS8300-L MTCA.4 Digitizer Struck Board 18

4.1 Overview of the system as a whole. 20
4.2 FSM with transaction triggers in block Ingress 21
4.3 FSM with transaction triggers in block Egress 22
4.4 FSM with transaction triggers in block DMA 28
4.5 Structure of the interconnect. 29
4.6 Simple representation of the interconnect arbiter. 29
4.7 Overview of the memory interface. 30
4.8 Memory initialization process performed by the physical layer in the memory interface. 31

7.1 Illustration of correct/incorrect RTL code hierarchy. 39

List of Tables

2.1 TLP formats 13
2.2 TLP types 13

4.1 Register address mapping 24
4.2 Register 0x021 - Memory steady state signals 25
4.3 Register 0x221 - IRQ status 27
4.4 Register 0x222 - IRQ clear 27
4.5 Memory specification parameters needed for the memory interface. . 30

6.1 Device utilization extracted from place and route report. 35
6.2 Logic distribution extracted from place and route report. 35
6.3 Part of specific feature utilization from place and route report. 36

7.1 SDRAM command control signals 40

Abbreviations

ESS	European Spallation Source
FPGA	Field-Programmable Gate Array
AXI	Advanced eXtensible Interface
PCIe	Peripheral Component Interconnect Express
I/O	Input/Output
ADC	Analog to Digital Converter
DAC	Digital to Analog Converter
CPU	Central Processing Unit
TLP	Transaction Layer Package
BDF	Bus, Device, Function
BAR	Base Address Register
MIG	Memory Interface Generator
DDR3	Double Data Rate type Three
SDRAM	Synchronous Dynamic Random Access Memory
RTL	Register-Transfer Language
IP	Intellectual Property
FIFO	First In First Out
LLRF	Low Level Radio Frequency
DMA	Direct Memory Access
FSM	Finite State Machine

Chapter 1

Introduction

This chapter summarizes the report and explains both the motivation and the reason for this project being created. We introduce a background of how the ESS particle accelerator will be used to split atoms to create neutrons, and we also highlight the goals and purposes of this work to give the reader a good overview of this project.

1.1 Background

The European Spallation Source (ESS) is a multi-disciplinary research facility based on what will be the world's most powerful neutron source [1]. It is a large collaboration between 17 different countries with employees from over 40 different nations. The facility will be built in Lund and is projected to reach full capacity in 2025 with a run time of 5000 hours a year and reliability of 95% [2].

ESS will technically include four building parts:

1. The first part is the ion source responsible for delivering a proton beam. For the facility's current specifications the planned component for this part is a compact Electron Cyclotron Resonance (ECR) source. The beam is planned to fire with a frequency of 14 Hz and should be held for a period of 2.86 ms. The reliability should reach 99.9% and thereby only contribute 0.1% to the total uncertainty.

2. Within the next part, the accelerator, the protons from the ion source are pushed together and accelerated through the facility. 150 cavities are placed along the 600 meter long tunnel to accelerate and guide the beam. Each of these cavities will be connected with a Field-Programmable Gate Array (FPGA), which will be responsible for real time regulation of the electric field inside the cavity. An overview of the regulator system is shown in Figure 1.1. The idea is to regulate the field so that the proton beam hits it on it's positive flank, making the protons accelerate. This is in theory done by representing sampled data on the IQ plane and multiplying them with a rotation factor to get to the sought positions [3].

   The FPGA is also connected to a Central Processing Unit (CPU) responsible for the FPGA setup, longtime logging and multi-pulse algorithms.

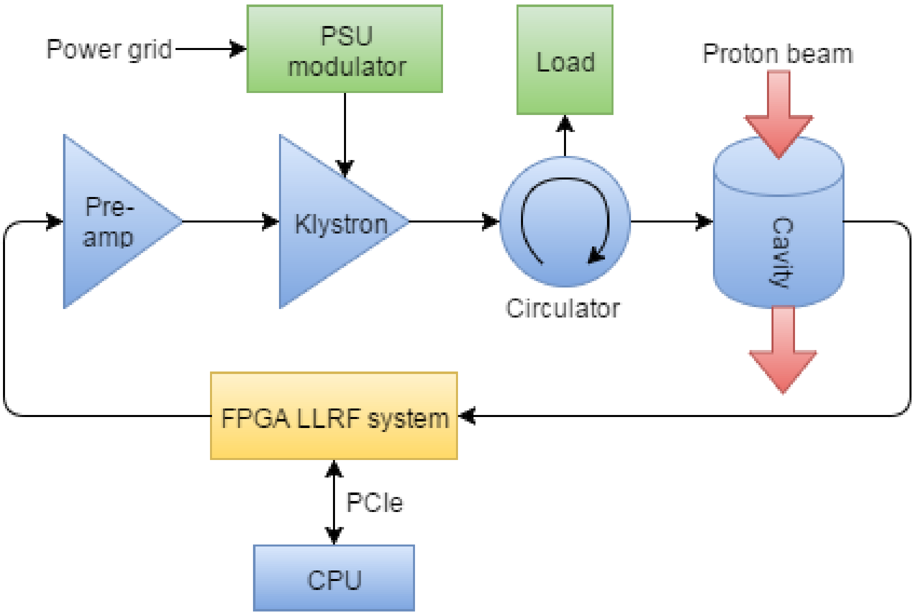


Figure 1.1: Overview of the LLRF system setup.

3. The purpose of the target is to transform high powered proton beams to low energy neutron beams. This is achieved by spallation, a process in which splitter is ejected from a body after it is hit by a large impact. In this case the impact will be produced by the accelerated proton beam. In this project the body (the target) is specified as a rotating wheel of tungsten cooled by helium gas. An important factor of this target is a radiation shielding system consisting of approximately 7000 tons of steel. This is crucial since spallation not only emits neutrons but also dangerous gamma- and fast neutron radiation.
4. Research instruments constitute the final part of ESS. The facility is planned to have 22 research instruments running in 2025. These instruments gather neutrons to support research in a multitude of fields such as medicine, biotechnology and energy.

1.2 Purpose of the project

1.2.1 Practical application

This application is meant to serve as an intermediary between software (SW) driven applications and the FPGA. Figure 1.2 shows the setup for cavity testing as it is today, where our design is placed on the digitizer board in the MTCA.4 crate to help the Low-Level Radio Frequency (LLRF) control system communicate with the CPU. As of today there is a working design for the communication part but its functionality is questionable. The original design was bought from a third party company that only provided a netlist file with constraint information (NGC) for implementation. This makes it impossible to change or adapt the functionality. To get full control of the implementation it would have to be rebuilt in-house. Since this design has to fit with the current design SW, all settings and parameters of the black-boxed design will therefore have to be reverse engineered.

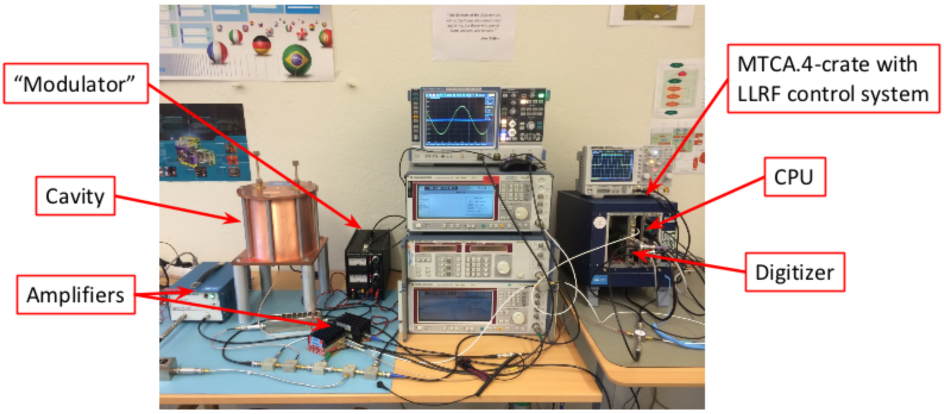


Figure 1.2: Setup for cavity testing and electric field regulation

1.2.2 Goals

At the start of the project a number of partial goals were decided on, both theoretical and practical. The theoretical parts encompass the general understanding of how a modern particle accelerator works. The practical parts will be designed in both firmware and software and will be divided into five smaller milestones. By proceeding in this manner the project will have a better structure and the accomplishments will be easier to measure.

Goal 1

The first goal will be to create the custom Peripheral Component Interconnect Express (PCIe) interface, as shown in Figure 1.3. The goal will be fulfilled when communication between software and the interface behaves as the built in version.

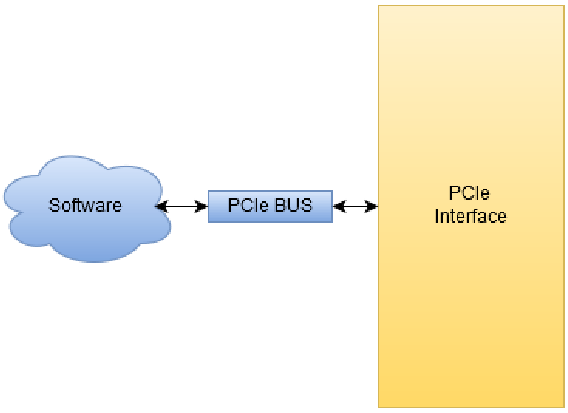


Figure 1.3: First set goal of the project.

Goal 2

The second goal will be to extend the interface with a Direct Memory Access (DMA) block, as shown in Figure 1.4. This will be tested with one of the FPGAs internal memory blocks to avoid unnecessary problems with the Double Data Rate type Three (DDR3) Synchronous Dynamic Random Access Memory (SDRAM). This goal will be fulfilled when it is possible to successfully transfer data to and from the memory through the DMA.

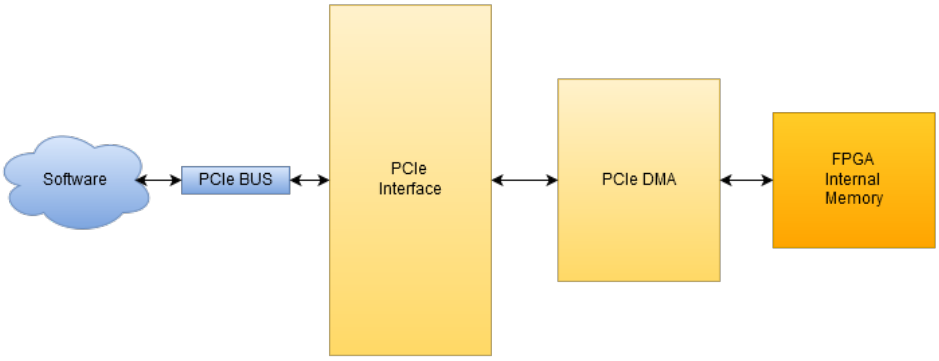


Figure 1.4: Second set goal of the project.

Goal 3

During the third stage of the project the built in version of the PCIe IF and DMA will be used again. The custom memory interface will then be added, as shown in Figure 1.5, to get more control over writing and reading from the memory. Because this block has to work for our memory block, we will also switch the FPGAs internal memory to the DDR3 memory. This goal will be completed when it is possible to read and write to the memory.

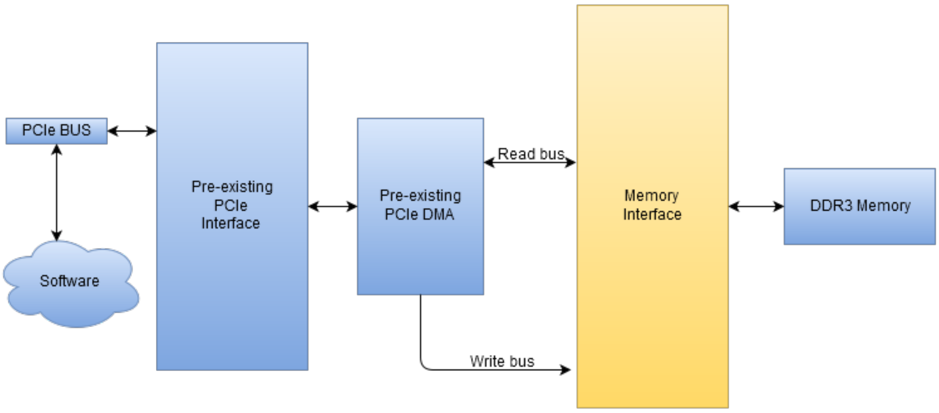

PCIe BUS
Software
Pre-existing PCIe Interface
Pre-existing PCIe DMA
Read bus
Write bus
Memory Interface
DDR3 Memory

Figure 1.5: Third set goal of the project.

Goal 4

Here the custom blocks from goal two and three are combined, as shown in Figure 1.6. No new blocks are added. This goal will be completed when goal two and three are completed again in the combined system.

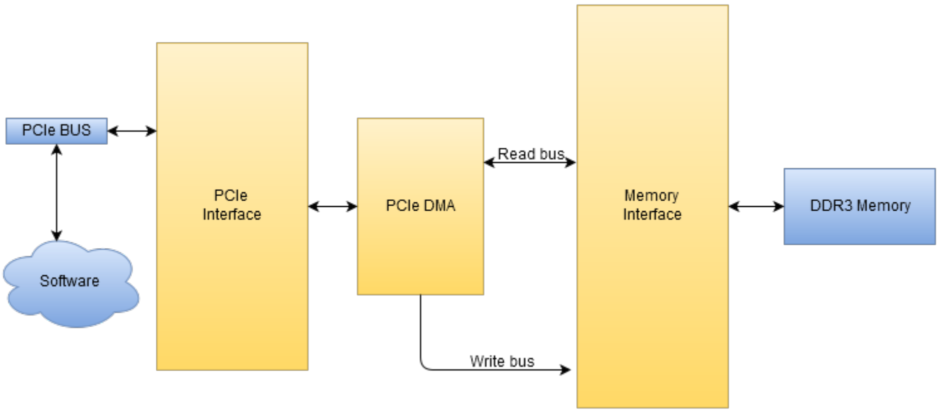

PCIe BUS
Software
PCIe Interface
PCIe DMA
Read bus
Write bus
Memory Interface
DDR3 Memory

Figure 1.6: Fourth set goal of the project.

Goal 5

By adding the analog to digital input from the cavities, as shown in Figure 2.7, a new interface is added to the system. Now there are two different inputs that can write to memory. Since the software and the data from the cavities can not write to memory at the same time, a write arbiter is needed. The purpose of this arbiter is to control which input gets access to the memory and which input gets delayed. This goal will be completed when the arbiter can give correct priority to two different inputs and write the correct input to memory.

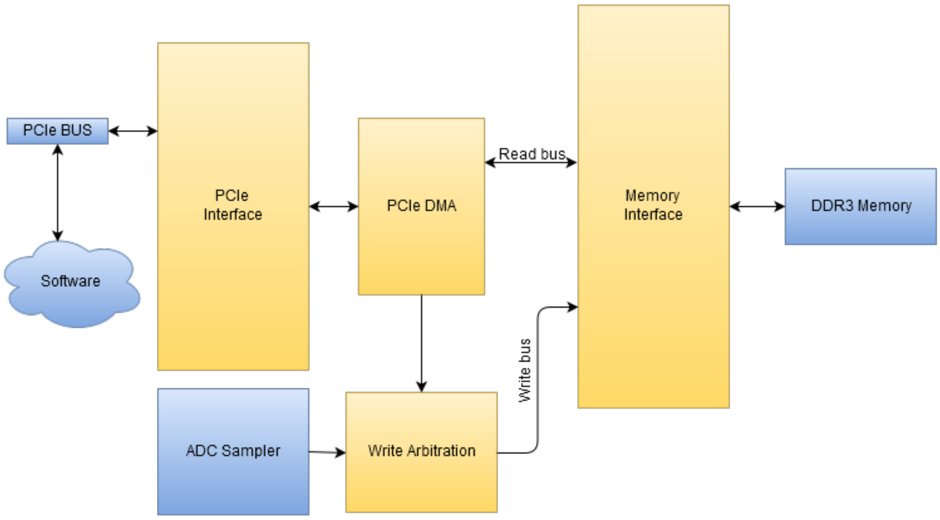

PCIe BUS
Software
PCIe Interface
PCIe DMA
Read bus
Memory Interface
DDR3 Memory
Write bus
ADC Sampler
Write Arbitration

Figure 1.7: Fifth set goal of the project.

Optional final goal

Time permitting, a final goal could be defined through the addition of a read arbiter.A read arbiter has the same purpose as the write arbiter from the fifth goal, the difference being that it manages reading from memory instead of writing.

1.3 Thesis structure

Here follows a short description of the chapters introduced in this report.

Introduction

Introduces the project and motivation to the thesis.

Theory

Explains some of the concepts used during the thesis.

System

Introduces the tools and equipment used during our work.

Implementation

Explains the functionality of all the included blocks.

Verification

Describes the tools used during verification.

Results

Presents the final results of the finished project.

Analysis

Discusses our experience of this project and possible future work which could advance the system.

References

Lists the references that have been used during the thesis.

Chapter 2

Theory

This chapter introduces multiple concepts such as the different bus systems that have been used throughout the project. Only the most general and relevant parts are included and described thoroughly.

2.1 PCI Express

2.1.1 Compability

PCIe is a third generation input/output (I/O) bus standard with second generation predecessors PCI and PCI-X. The new bus is backwards compatible with the old standards though, and can therefore be used together with older systems. This feature simplifies the plug and play functionality of the bus as long as the connected peripherals are compatible with the older bus standards.

One of the large differences of this version compared with the older standards is the difference in internal connectivity of the bus. Both PCI and PCI-X use a multidrop parallel interconnected shared bus model. This means that the system has one bus that all peripherals are connected to and through which they communicate with each other in parallel. PCIe devices do not work simultaneously in parallel but in serial point-to-point connections. Instead of communicating with a shared bus, the devices are connected with switches or direct connections to a root complex. PCI and PCI-X buses can be connected to the root complex as if they were PCIe endpoints, making the whole structure backwards compatible.

2.1.2 Root complex structure

A root complex is used to connect the CPU and memory subsystems to PCIe connections. This helps to relieve the CPU from work since the root complex generates memory and I/O packages in response to the transactions sent by the CPU. The same transactions can also be received from devices located further downstream in the hierarchy. These devices can either be switches that allow the root port to connect to more devices than it has available ports, or endpoint devices that are all requesters or completers of PCIe transactions. An example of a root complex system is shown in Figure 2.1. Memory blocks, Ethernet connectors or graphic devices can all be endpoints if they follow the PCIe standard.

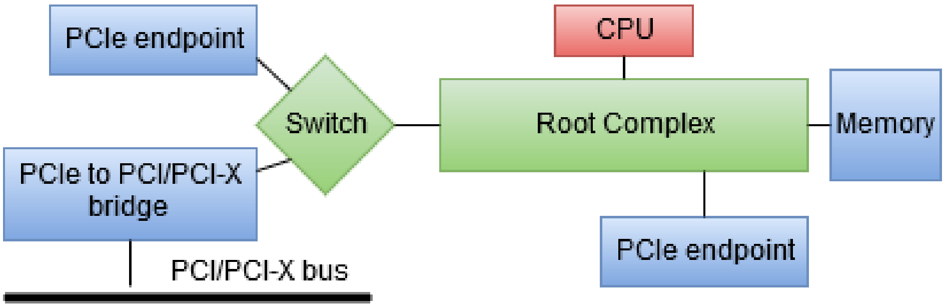


Figure 2.1: Example of a PCIe system with a root complex and multiple endpoints in the form of one memory, two PCIe endpoints and a PCI/PCI-X bus which can be connected with multiple peripherals which are compatible with older bus standards.

2.1.3 Enumeration

Since there is no available information at start up the CPU needs a method to identify the surrounding PCIe devices. Enumeration is a process that makes it possible to identify the connected devices by finding the corresponding Vendor- and Device ID numbers. These numbers are found at address zero of the PCIe compatibility registers of each PCIe device. During enumeration the CPU addresses all possible devices by going through all Bus, Device, Function (BDF) numbers. If a device exists on the specified port the device will answer with its ID numbers whereas if no device is present on the specified slot a value of all ones is returned. When all devices that incorporate a Base Address Register (BAR) and have valid device- and vendor IDs have been enumerated, the CPU can start sending transactions. The addresses stay valid until the system is powered down and the settings are lost. Once the system turns on again, the enumeration process restarts.

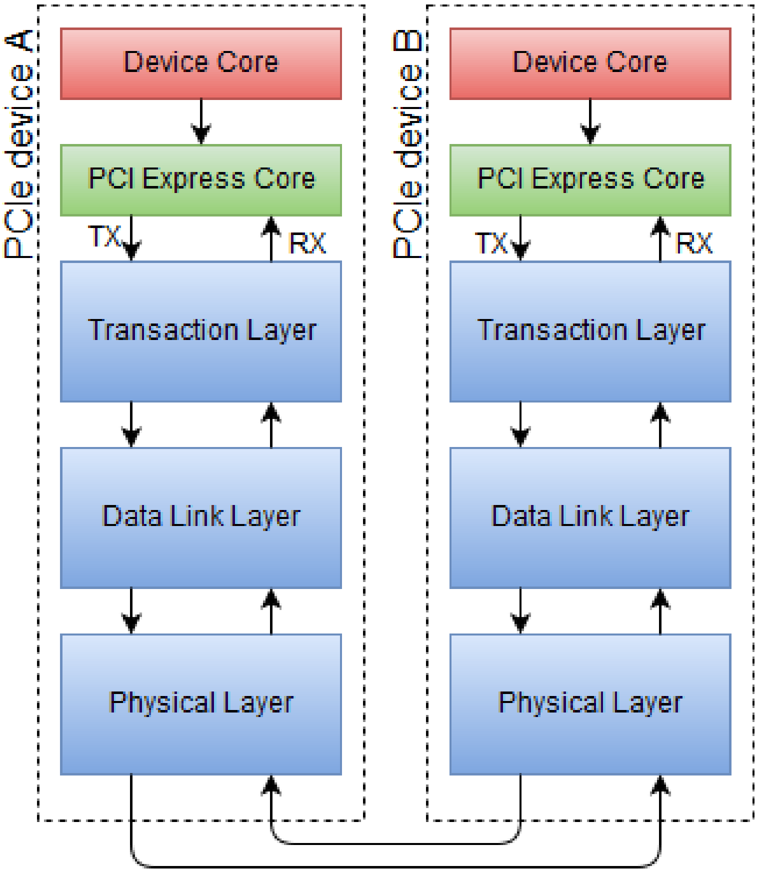

PCIe device A
PCIe device B
Device Core
Device Core
PCI Express Core
PCI Express Core
TX
RX
TX
RX
Transaction Layer
Transaction Layer
Data Link Layer
Data Link Layer
Physical Layer
Physical Layer

Figure 2.2: Simple representation of a link between the layers of two PCIe devices.

2.1.4 PCIe hierarchy

The communication between PCIe devices is managed by transmission and reception of packages. These packages are called Transaction Layer packages (TLP) since they originate from the Transaction Layer of the PCIe hierarchy. This layer is located at the top of the hierarchy, and is followed by the Data Link Layer and the Physical Layer.A simplified version of how these layers communicate is shown in Figure 2.2 [4].

Transaction Layer

The Transaction Layer contains buffers that are used to store inbound and outgoing TLPs. The packages sent to the receiver are assembled with the data header, data payload, and depending on whether it is included in the specification, an end to end Cyclic Redundancy Check (CRC) field. On the receiving side this process is reversed by extracting the information from the package before sending it to the device core. If the receiver buffer is full at the time a new transmission is started then the package is stalled on the transmission side. This is a feature called flow control and is implemented on the hardware level of the PCIe protocol.

Data Link Layer

When receiving a TLP from the Transaction Layer, the Data Link Layer adds a sequence number and a Link CRC field to the header. Before sending the package on to the Physical Layer, a copy of the TLP is saved in a replay buffer. When the package is received by the Data Link Layer of the other device the CRC field is checked for errors. If there is no error an Acknowledge (ACK) package is sent back. When the acknowledgement is received the specified TLP is cleared from the replay buffer. On the other hand, if there is an error, a NACK is sent back to the transmitter. The transmitter then retransmits the TLP from the replay buffer. In this way the transmission is self correcting on a transmission level without the involvement of SW.

Physical Layer

The physical layer only has a minor role in affecting the outgoing packages in the form of adding a start/end bit at the respective sides of the package. These bits are called framing bits, as shown in Figure 2.3. The receiving side of the connection uses these bits to determine where the packages start and end. In addition to this task the physical layer is responsible for link training and initialization. These processes configure both sides of a link to be synced with each other. An example of one such configuration is where a device with two data lanes is connected to a device with four data lanes. In such case the initialization process decides which two of the four lanes on the second device shall be connected to the two of the first device.

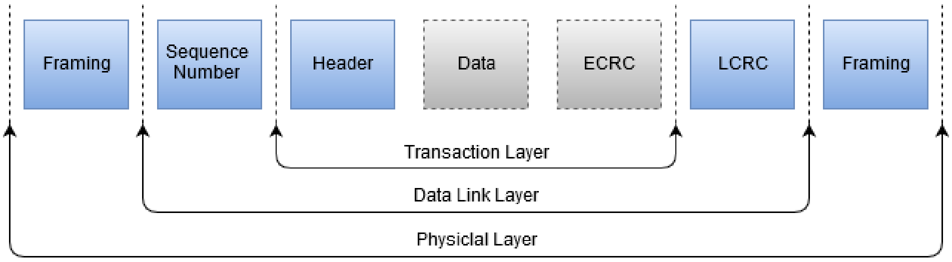


Figure 2.3: The contents of a TLP. The parts added by the individual layers are marked with dotted lines. The Data and ECRC data blocks are optional dependent on the purpose of the package.

2.1.5 TLP - Transaction layer package

Packages leaving the transaction layer can have multiple purposes e.g. memory- or I/O requests. Independent of what kind of package it is, the first four byte header of the TLP always has the same structure as shown in Figure 2.4. The Reserved

(R) fields in the header must be left as zeros and are unmodified on their way to the receiver. The other fields are important for knowing what the purpose of the package is and what kind of attributes it has.

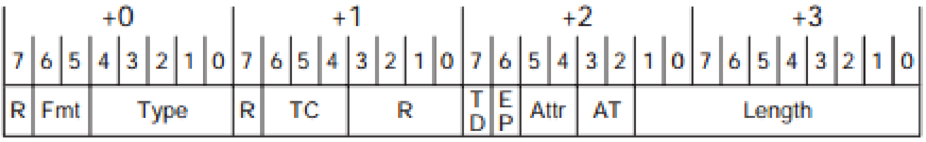


Figure 2.4: The first header present in all TLPs [5]

.

The Format (FMT) field indicates how big the header is in Double Words (DW - 32 bits) and if the package contains data or not as shown in Table 2.1. In the case of this system, only header sizes of three DWs are used since only 32-bit addresses are supported.

Table 2.1: TLP formats

FMT	TLP format
00	3 DW header, no data
01	4 DW header, no data
10	3 DW header, with data
11	4 DW header, with data

Type decides on the purpose of the package. In this system the formats needed are shown in Table 2.2, although in the general case up to 17 different format and type combinations can be supported.

Table 2.2: TLP types

Type	FMT	TLP type
00000	00	MRd - Memory Read Request
00000	10	MWr - Memory Write Request
01010	00	Cpl - Compeltion without data
01010	10	CplD - Completion with data

Traffic Class (TC) is used internally in the PCIe fabric and is responsible for servicing the packages. TLP Digest (TD) indicates the presence of a digest field at the end of the package, including an ECRC. The EP field shows if the TLP is poisoned, which is an error detection concept not covered in this project. Attribute (Attr) gives the possibility to modify the handling of TLPs. The first bit in the field controls how the TLP is ordered, default or relaxed. The second bit is responsible for hardware coherency management, which similar to TLP poisoning is an attribute not supported in this project. The length field contains the size of the data payload in DWs.

In the case of the TLP being a memory request additional headers are added as shown in Figure 2.6. The additional fields guarantee that the package is associated with the right sources and destinations. Requester ID corresponds to the BDF number of the system that requested the read/write. The Tag field helps with pairing one request with its corresponding completion. Byte Enable (BE) tells the completer which bytes of the package payload should be written/read. The address field shows to- or from where data should be written/read.

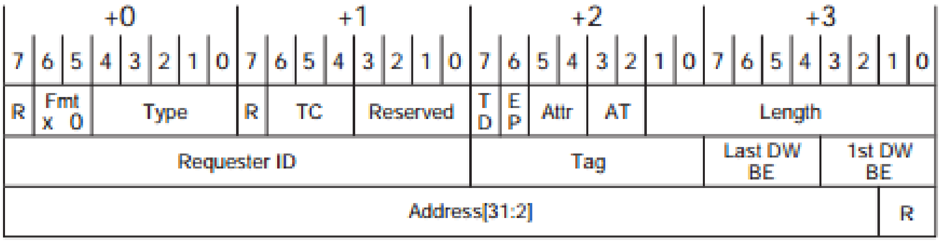

+0
+1
+2
+3
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
R
Fmt x 0
Type
R
TC
Reserved
TD
EP
Attr
AT
Length
Requester ID
Tag
Last DW BE
1st DW BE
Address[31:2]
R

Figure 2.5: The TLP header with the additional memory request header.

For completions the TLP header is extended as shown in Figure 2.6. Completer ID tells the requester which system sent the completion. This is defined as the systems BDF number. The Completion Status field contains knowledge about the packages valid status. Byte Count determines how much data remains to be received for this tag. The Lower Address field is only set for memory read completions. The value of the field is the byte address for the first enabled byte of data returned with the completion [5].

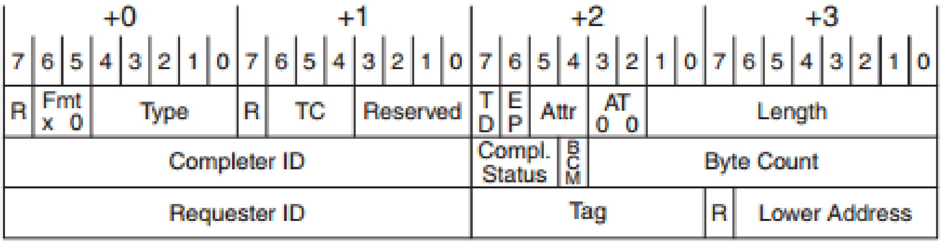

+0
+1
+2
+3
7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
R
Fmt x 0
Type
R
TC
Reserved
TD
EP
Attr
AT 0 0
Length
Completer ID
Compl. Status
BCM
Byte Count
Requester ID
Tag
R
Lower Address

Figure 2.6: The TLP header with the additional completion header.

2.2 AXI - Advanced eXtensible Interface

AXI is a bus protocol first was introduced in 1996 by the company ARM. In 2010 the second version of the bus, AXI4, was introduced. Three versions of the protocol, AXI4, AXI4-Lite and AXI4-Stream were also brought out each with it's own use. AXI4 and AXI4-Lite are memory mapped interfaces implemented in the form of two channel pairs, one data -and address channel in both directions, and a response channel. This way it is possible for read and write transactions between a master-slave pair to be performed simultaneously. The difference between these two standards is that AXI4-Lite does not implement any burst functionality and therefore can only send one package at a time. On the other hand AXI4-stream is a high speed interface that only sends data and control signals, meaning that the interface does not implement an addressing functionality. Many of the Intellectual Properties (IP) that Xilinx provides supports one or multiple of these AXI interfaces, which therefore makes it a lot easier to design systems since all the blocks share a common bus type.

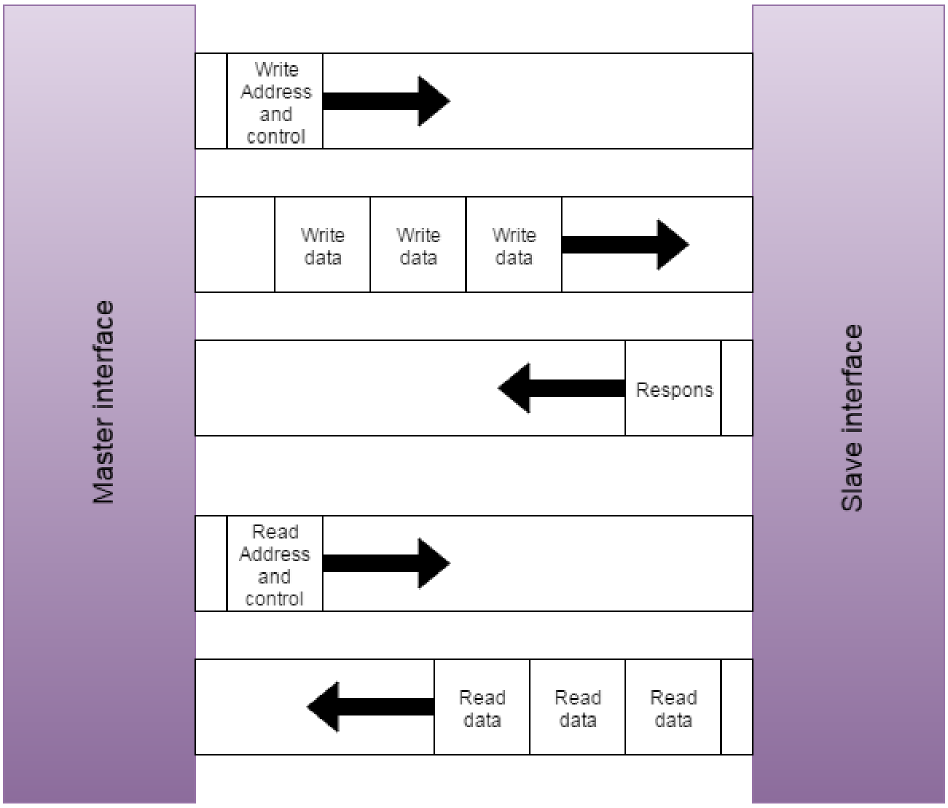

Master interface
Write Address and control
Write data
Write data
Write data
Respons
Read Address and control
Read data
Read data
Read data
Slave interface

Figure 2.7: AXI communication between master and slave.

2.2.1 Master/Slave

The bus structure is built up by having a master and a slave. The master sends write- and read- requests to the slave while the slave sends responses to the master. Instead of having a header for different packages the bus has different channels dedicated to different purposes. There are five channels: write- (W), read- (R), address write- (AW), address read- (AR) and response- (R) channel. W and R are dedicated for data transactions while AW and AR are dedicated to addressing memory. The R channel indicates whether or not a write was completed properly.

Write channel

The write channel has five signals: data, valid, last, keep and ready. The master offers one beat of data by asserting the valid signal. The data is accepted if the ready signal is asserted one clock cycle when valid is high. By using last, the master can tell the slave that the present data is the last data of the package. Keep is a byte enable signal that indicates which bytes in data are valid.

Read channel

The read channel works in the same way as the write channel except data travels from slave to master. The master accepts data by asserting the ready signal, whilst valid, last and keep tells the master which data to accept.

Address write channel and response

There are seven signals in the AW channel: addr, valid, ready, burst, cache, len, size, lock, qos and prot. These signals describe what will be presented on the W channel as well as where data is stored. The master offers an address together with a description of the package by asserting the valid signal. The slave accepts the address by asserting ready for one clock cycle. In AXI4 the burst signal indicates which mode the addressing should be in, e.g. incremental as we use in this design, whilst the rest of the signals describe how to handle data. Len defines how many beats the package carries, size is how many bytes each beat carries and cache defines the possibility of modifying the data. To read more information on signal declarations please turn to Xilinx AXI reference pages [6].

Address read channel

The AR channel works similarly to the AW channel but instead of to where and how the master tells the slave from where and how. This way we know what to suspect from the read channel.

Chapter 3

System

3.1 Development tools

3.1.1 ISE project navigator

Xilinx project navigator is a complete environment for the design and development of Xilinx products [7]. It provides project and design source management and allows the user in easy steps to transverse through the FPGA design flow such as synthesis and place and route. ISE also provides a tool called chipscope which allows the user to trigger and store signal values during run time for better debug possibilities. After place and route has been performed a .bit file is created containing all needed data to route the design on the FPGA. The program can be run by using the command prompt in windows and can also be scripted using Tool Command Language (TCL) scripts to automate the work flow. Even though this is preferred we used the graphical interface when designing.

3.1.2 iMPACT

This program is used to configure the FPGA device. The FPGA connects its routing net by reading the information on a Programmable Read Only Memory (PROM). This PROM is configured via the JTAG connection on the board and can be set up by iMPACT. iMPACT uses the .bit file created in ISE to generate a .mcs file that is used to configure the PROM.

3.2 Debugging tools

3.2.1 ISim - ISE simulator

ISim has been the main debugging tool used to run pre-synthesis simulations of the design. The tool is built into ISE and makes it possible to observe the design-generated signals in response to predefined stimuli.

3.2.2 Questasim

In extension to Isim, Questasim has been used. As a third party simulator it provides some extra features and behaves a little differently from the Isim simulator. The reason for using different simulators will not be covered in this report to a great extent than explaining the lessons learned whilst using the different tools.

3.2.3 Analyzer

This program is necessary when using Chipscope. In this program the user can connect to the FPGAs JTAG connection, set up trigger conditions and create signal waveforms of the sampled data. The connections on the FPGA are known by configuring the program using an ISE generated .bit file. Waveforms can then be exported as value change dump (VCD) files and analyzed in e.g. Questasim. In Questasim VCD files needs to be converted to WLF files.

3.3 Hardware equipment

The given hardware is a SIS8300-L MTCA.4 Digitizer Struck Board shown in Figure 3.1. The board contains multiple connectivity possibilities, such as analog to digital- and digital to analog converter (ADC/DAC) channels and a JTAG connector. The four lane PCIe bus will be the primary focus of this project. PCIe is a point to point protocol that will handle the communication between the FPGA located on the board, which is a Virtex6 model, and the software. Also accessible on the board is a 4x4Gbit DDR3 memory that can be connected to the FPGA. This is the memory which is referenced to in Chapter 1.2.2 when describing the goals.

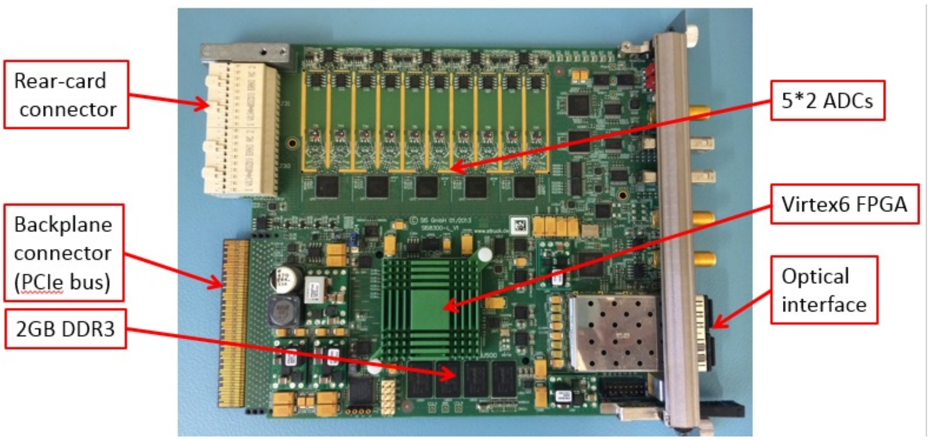


Figure 3.1: SIS8300-L MTCA.4 Digitizer Struck Board

Chapter 4

Implementation

4.1 Architecture overview

In Figure 4.1, an overview of the system is shown. All the system parts described in this chapter are included along with an explanation of how the parts are connected.

4.2 PCI Express interface

The endpoint PCIe block of this project is the Virtex-6 Integrated Block for PCI Express v2.5 [8] implemented using the IP CORE generator in ISE. This IP includes an AXI4-stream interface, which comes in handy when implementing the rest of the design. The IP consists of modifiable vhdl code and a black box (standard ISE component PCIE_2_0 [8]) containing the PCIe data- and physical layers. The modifiable files consists of the AXI4-stream interface and the receive- and transmit buffers. Thus the transaction layer that handles the incoming TLPs needs to be implemented separately. For this task two blocks are implemented; Egress is responsible for encoding outgoing TLPs and Ingress for decoding the incoming TLPs. With these two block a user is able to write and read 32 bit words to a register file using write- and read requests. This register file contains control signals for the circuit.To access the DDR3 memory a DMA block is implemented to relieve the CPU of large data transfers. To do this the user sets up DMA registers seen in table 4.1, which the DMA block uses to create either write- or read requests to send upstream.

4.2.1 Ingress

The Ingress block is the first step on the receiving side of the interface. In this block the headers of the incoming packages are evaluated and reactions depending on the format and type fields of the package are generated. A TLP sent downstream is considered to be valid if it has the format and type of a write- or read request of 32 bits or a completion containing data to be written to DDR3 memory.

When the logic detects a start of a TLP the Finit State Machine (FSM), shown in Figure 4.2, determines the format and type of the package and triggers transactions to corresponding states. In the write state, data and address are

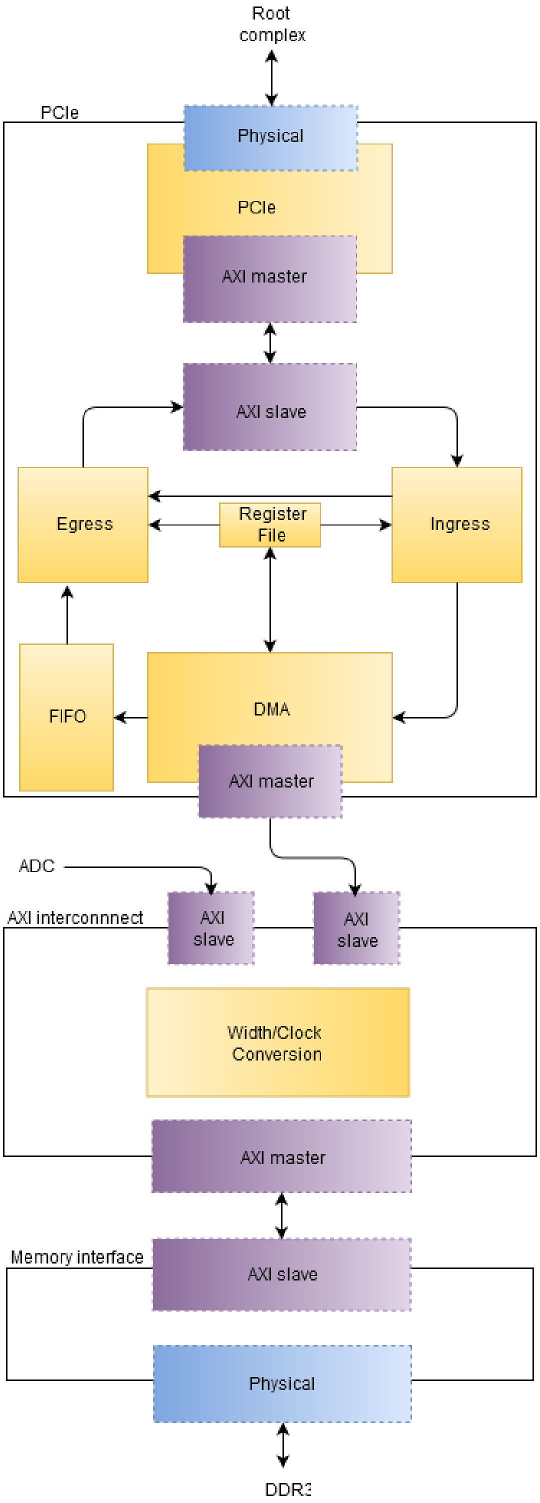

Root
complex
PCIe
Physical
PCIe
AXI master
AXI slave
Egress
Register
File
Ingress
FIFO
DMA
AXI master
ADC
AXI interconnnect
AXI
slave
AXI
slave
Width/Clock
Conversion
AXI master
Memory interface
AXI slave
Physical
DDR3

Figure 4.1: Overview of the system as a whole.

decoded and sent together with a write enable signal to the internal register block. At the same time the FSM transverse into the W8_STATE. This invokes a control signal (write_busy) to go high, telling the FSM to wait in the W8_STATE until the FSM is ready to accept a new TLP. If the TLP is a read request the address is decoded and the data is discarded. The address is sent to the register block and FSM waits in the W8_STATE until the control signal (comp_done) indicates that a completion containing the right data was sent upstream before returning to RST_STATE.

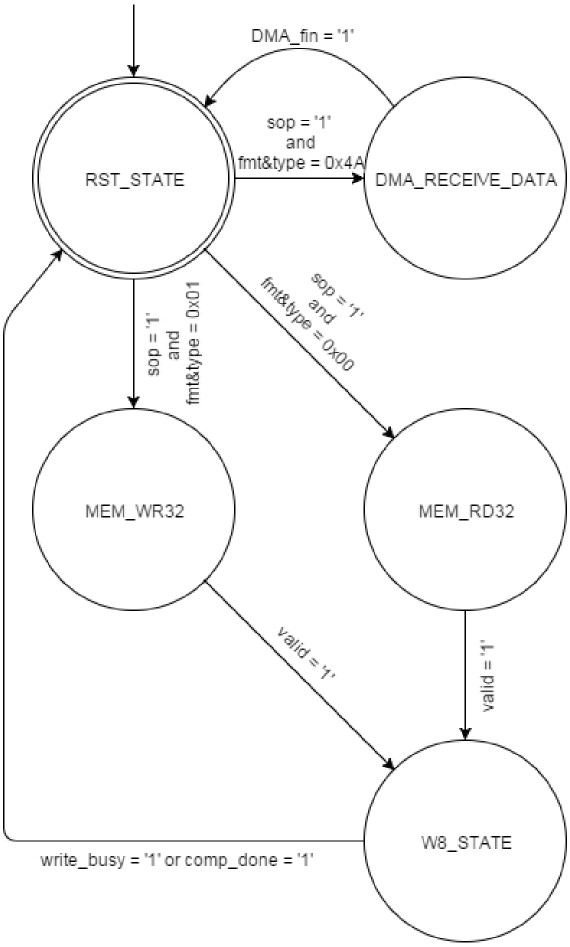

DMA_fin = '1'
sop = '1' and fmt&type = 0x4A
RST_STATE
DMA_RECEIVE_DATA
sop = '1' and fmt&type = 0x01
sop = '1' and fmt&type = 0x00
MEM_WR32
MEM_RD32
valid = '1'
valid = '1'
W8_STATE
write_busy = '1' or comp_done = '1'

Figure 4.2: FSM with transaction triggers in block Ingress

The only way that a completion can be sent downstream is if a DMA transaction has been initiated by the CPU, in which case a read request has been sent upstream. The completion will contain the requested data and should be forwarded to the DMA unit, which will write it into memory. During transfer the FSM will lock in RECEIVE_DMA_DATA state, leaving the DMA to handle all data until the last data has been received.

4.2.2 Egress

This block is responsible for generating TLPs that will be sent upstream in response to requests from other blocks. The main objective is to create TLP headers of the outgoing data. There are three possible scenarios that can occur namely: DMA write transaction is initialized, DMA read transaction is initialized or a completion needs to be sent as a reaction to a received read request.

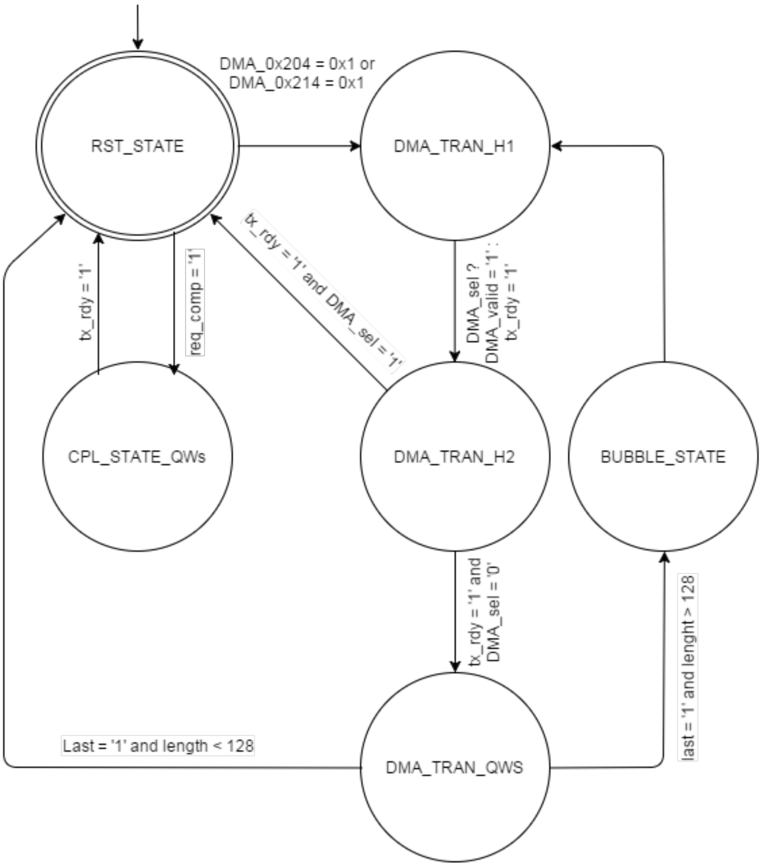

DMA_0x204 = 0x1 or
DMA_0x214 = 0x1
RST_STATE
DMA_TRAN_H1
tx_rdy = '1'
req_comp = '1'
tx_rdy = '1' and DMA_sel = '1'
DMA_sel ?
DMA_valid = '1' :
tx_rdy = '1'
CPL_STATE_QWs
DMA_TRAN_H2
BUBBLE_STATE
tx_rdy = '1' and
DMA_sel = '0'
last = '1' and lenght > 128
Last = '1' and length < 128
DMA_TRAN_QWS

Figure 4.3: FSM with transaction triggers in block Egress

DMA read request

In RST_STATE a DMA read request will be initiated by the computer writing the DMA WRITE CTRL register, explained in Section 4.2.3. As this happens a register called DMA_SEL is assigned to low to clarify that a read transaction is active, and FSM shown in Figure 4.3 switches to DMA_TRAN_H1 state. Here it waits until valid data from memory is present in the DMA_READ_FIFO shown in

Figure 4.1. Because of packet configurations in the DMA_READ_FIFO a header can now be created knowing that data will follow continuously. A write request TLP is created and sent upstream together with requested data. This is written to a base address specified by the DMA READ DST registers. If the read requests length is larger then 128 bytes, which is the maximum payload size configured in the PCIe block, the FSM cycles back to DMA_TRAN_H1 state after sending first TLP to repeat the procedure - each time adding an offset of 128 bytes to the base address until all requested data has been written to computer memory. As the last package is sent an interrupt is sent to the computer to request a polling of the IRQ Status register shown in Table 4.3.

DMA write request

Write request is mostly handled by the ingress block but needs to be initiated by sending read requests to the computer. This way the CPU does not need to initiate all transactions. In RST_STATE the FSM is waiting for DMA WRITE CTRL to be written. When this happens DMA_SEL is asserted high and FSM switches to DMA_TRAN_H1 state where the first request header is created. FSM then switches to DMA_TRAN_H2 where addressing specified in DMA WRITE SRC is made before returning to RST_STATE.

Completion

If a completion is requested the block builds the first header of the completion header and then jumps to CPL_STATE_QW1, which will combine the second header together with the requested data. The addressing of the register bank is done in the RST_STATE so there is no latency loss as long as the transaction buffer is ready.

4.2.3 Register file

This block contains registers that are available for read- and write operations from CPU. The registers acts as control- and status registers so that the computer can keep track of what's happening on the hardware side of the system. The address mapping can be read from Table 4.1 followed by an explanation of specific registers.

Table 4.1: Register address mapping

Offset	Access	Function
0x000	R	Module Identifier/Firmware Version register
0x010	R/W	ADC Acquisition Control/Status register
0x011	R/W	ADC Sample Control/Status register
0x020	R/W	
0x021	R/W	
0x0FF	R/W	Master Reset register
0x120	R/W	ADC ch 1 Memory Sample Start Block Address
0x121	R/W	ADC ch 2 Memory Sample Start Block Address
...	R/W	
0x129	R/W	ADC ch 10 Memory Sample Start Block Address
0x12A	R/W	ADC chx Sample Block Length register
0x200	R/W	DMA READ DST ADR LO32
0x201	R/W	DMA READ DST ADR HI32
0x202	R/W	DMA READ SRC ADR LO32
0x203	R/W	DMA READ LEN
0x204	R/W	DMA READ CTRL
0x205	R/W	DMA readout Sample byte swap control
0x210	R/W	DMA WRITE DST ADR LO32
0x211	R/W	DMA WRITE DST ADR HI32
0x212	R/W	DMA WRITE SRC ADR LO32
0x213	R/W	DMA WRITE LEN
0x214	R/W	DMA WRITE CTRL
0x220	R/W	IRQ Enable
0x221	R/W	IRQ Status
0x222	R/W	IRQ Clear
0x400	R/W	User defined register implementation
...		...
0x4FF	R/W	

0x000 - Module Identifier/Firmware Version register

This register is used by CPU to identify an open connection between the computer and the FPGA. The content is read every time CPU communicates with the FPGA.
Default value: 0x83012808

0x010 - ADC Acquisition Control/Status register

Writing a value of 0x1 to this register will perform an immediate sampling of the ADC channel.
Default value: 0x0

0x020 - Memory manual reset

This register resets the memory interface and the interconnect. This is because of a reset problem that occurs when programming the FPGA via the PROM. Reset by writing value 0x1. Reset needs to be de-asserted by writing 0x0.
Default value: 0x0

0x021 - Steady state signals

This register contains the steady state signals that validates a working design. The bit description can be seen in Table 4.2
Default values: 0x0

Table 4.2: Register 0x021 - Memory steady state signals

Bit nbr	Write function	Read function
31	-	-
...	...	...
2	-	physical memory initialization done
1	-	pcie link up
0	-	bus master enable

0x0FF - Master register reset

Writing 0x1 to this register resets all register values to default values.
Default value: 0x0

0x120 - 0x129 ADC Memory Sample Start Block Address

This register contains the start addresses of the memory where ADC channels will write their sample values. The user is responsible to check that the addresses do not overlap.
Default values : 0x0

0x12A - ADC Sample Block Length register

This register defines the number of 256 bit blocks that will be written by each active ADC channel per trigger.
Default value : 0x0

0x200 - DMA READ DST ADR LO32

This register contains lower address of the computers DMA buffer. User writes this before every memory read request.
Default value: 0x0

0x202 - DMA READ SRC ADR

This register contains the start address from which DMA will read the DDR3 memory.
Default value: 0x0

0x203 - DMA READ LEN

This register defines the number of bytes that is requested when performing a read request. The user is responsible that the length and start address combined do not overlap the last address of the DDR3 memory.
Default value: 0x0

0x204 - DMA READ CTRL

Writing the value 0x1 to this register will initialize a read request. Be sure to write register 0x200 - 0x203 before initializing the transfer to get satisfying results.

0x210 - DMA WRITE SRC ADR LO32

This register contains lower address of the computers DMA buffer where data that is suppose to be written are located.
Default value: 0x0

0x212 - DMA WRITE DST ADR

This register contains the base address of DDR3 memory to which the data will be written.
Default value: 0x0

0x213 - DMA WRITE LEN

This register defines the number of bytes that is written when performing a write request. The user is responsible that the length and start address combined do not overlap the last address of the DDR3.
Default value: 0x0

0x214 - DMA WRITE CTRL

Writing the value 0x1 to this register will initialize a write request. The user needs to be sure to write register 0x210 - 0x213 before initializing the transfer to get satisfying results.

0x221 - IRQ status

This register indicates when an interrupt has occurred. Table 4.3 shows the bit representation of the register.

Table 4.3: Register 0x221 - IRQ status

Bit nbr	Write function	Read function
31	-	-
...	...	...
15	-	User IRQ happend
14	-	DAQ Done IRQ happened
14	-	-
...	...	...
1	-	Write DMA Done IRQ happened
0	-	Read DMA Done IRQ happened

0x222 - IRQ clear

By writing this register the user can reset and clear interrupts. Write functionality are shown in Table 4.4

Table 4.4: Register 0x222 - IRQ clear

Bit nbr	Write function	Read function
31	-	-
...	...	...
15	User IRQ clear	-
14	DAQ Done IRQ clear	-
13	-	-
...	...	...
1	Write DMA Done IRQ clear	-
0	Read DMA Done IRQ clear	-

4.2.4 DMA

In this design the DMA is implemented as a state machine that regulates the flow of data in and out of memory. The AXI interface on the PCIe block is an AXI4-stream interface while the interconnect and memory interface has an AXI4 interface that is memory mapped. One challenge for the DMA block is to translate between the two formats. The FSM is implemented with four states as shown in Figure 4.4. In RST_STATE the length registers are sampled at the start of each data transfer. Data transfers are initiated by control signals from either ingress- or egress block. If control signals initialize a read transfer, the FSM switch to DMA_READ_TO_ROOT state in which logic constructs and sends read requests to the memory. In the meantime it also receives data from the interconnects read channel. When all read requests have been sent the state machine transverses to W8_STATE. Here it holds its state until all data has been received. This is managed by a counter that keeps track of the numbers of un-replied packages. When a write transaction is initialized the FSM switches to DMA_AW_STATE in which the block generates an address based on DMA WRITE DST, shown in Table 4.1. Because of the limitation on the maximum payload size configured in the PCIe block this design does not need to send more than one address per write transaction. After sending the address, FSM switches to DMA_WR_FROM_ROOT state and data is sent to the AXI interconnect. When the transfer has been made an interrupt can be generated and sent to CPU.

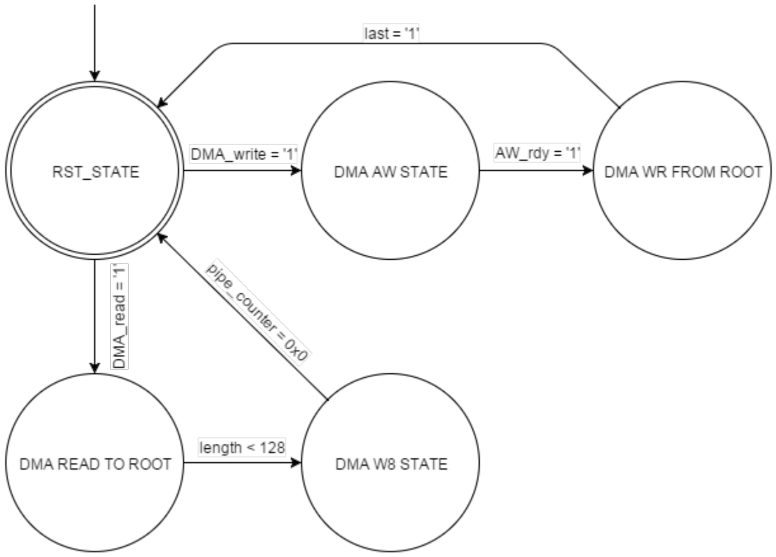


Figure 4.4: FSM with transaction triggers in block DMA

4.3 AXI interconnect - read/write arbiter

The AXI interconnect supports multiple functions relevant for this project's system, such as clock rate conversion, data width conversion and arbitration. The DDR3 memory will run at double the rate of the PCIe system and therefore the clock rate conversion is practical for this application. The interconnect supports both clock rate multiplication and division, in this case a 1:2 conversion is used. Since the AXI-converted data from the PCIe block have a data width of 64 bits while memory receives data in chunks of 256 bits, data width conversion is needed. By buffering the 64-bit data from the DMA in First In First Out (FIFO) buffers on the ports of the interconnect it is possible to up-size the data to 256 bits which then can be sent to the memory interface. This goes in both directions since the data has to be downsized in the other direction when data is received from the memory, as shown in Figure 4.5 [9].

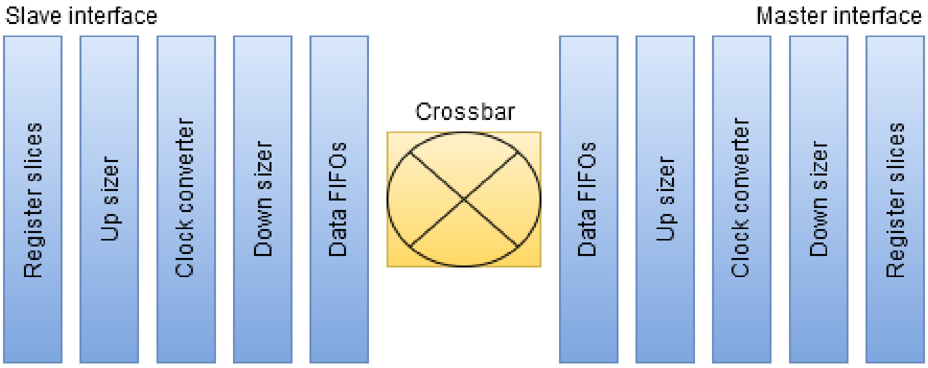

Slave interface
Master interface
Register slices
Up sizer
Clock converter
Down sizer
Data FIFOs
Crossbar
Data FIFOs
Up sizer
Clock converter
Down sizer
Register slices

Figure 4.5: Structure of the interconnect.

As described in Chapter 1.2.2, the introduction of the ADC input creates a need for read and write arbitration. The crossbar shown in Figure 4.5 includes an read -and write arbiter which can be implemented when generating the interconnect core, see a simplified model of our system in 4.6. The priority of the master devices can be set to a normal round-robin style, which means that all of them have the same priority, or are hard coded to a set priority.

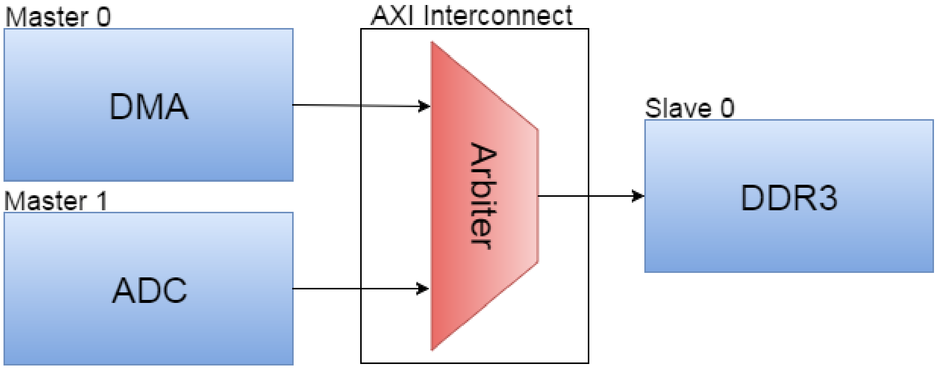

Master 0
DMA
Master 1
ADC
AXI Interconnect
Arbiter
Slave 0
DDR3

Figure 4.6: Simple representation of the interconnect arbiter.

4.4 Memory interface

The memory interface that connects the on-chip design with the external DDR3 memory was generated with Xilinxs MIG for Virtex6 and Spartan6. The block was generated in Verilog since the optional embedded AXI interface is unavailable for VHDL implementation. Parameters needed for the interface, shown in Table 4.5, were acquired from the predefined memory specification. Memory type refers to the DDR3 memory model located outside of the FPGA on the board. The data width is the width of the bus connected from memory interface to memory. The frequency specifies at which rate the DDR3 memory should be clocked. The burst length corresponds to the size of data bursts written to the memory. Other parameters required to generate the memory interface, such as AXI data- and address width, were customized to fit with the rest of the system [10].

Table 4.5: Memory specification parameters needed for the memory interface.

Memory type	Data width	Frequency	Burst length
MT41J256M16XX-15E	64-bit	400 MHz	8

The interface block consists of three parts: an AXI controller, an user interface and a memory interface controller containing a physical layer to the memory, as shown in Figure 4.7.

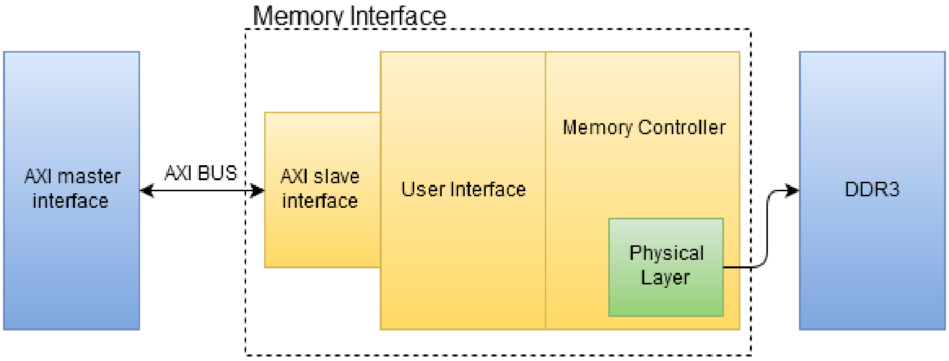


Figure 4.7: Overview of the memory interface.

4.4.1 AXI controller

The purpose of the AXI controller is to act as a slave to the master port of the AXI interconnect. It also handles the data conversion between the AXI- and user interface ports. Since the memory cannot handle too big bursts of data, the controller is also responsible for splitting the bursts into smaller bursts of four or eight packages depending on the specification in Table 4.5.

4.4.2 User interface

The user interface contains two FIFO buffers in which read and write data is buffered, making it possible to control the order in which data from memory is sent back. Since read data can be obtained in a different order than in which it was requested, this reordering is necessary.

4.4.3 Memory interface controller

The memory interface consists of bank-, rank- and column controllers and an arbiter that is connected to the physical layer. The controllers generate commands and timing parameters to send to the memory based on the translated data from the user interface. Since there are multiple controllers, more than one command can be active at the same time but only one at a time can be handled by the memory. The arbiter solves this problem by giving the controllers priority in round robin order. The active command is then forwarded to the physical layer which handles the communication with the memory. In addition to handling the low level communication the physical layer also handles memory initialization on start up. The initialization process, shown in Figure 4.8, is performed internally in the physical block right after resetting the system. During RAM initialization, memory interface writes to mode registers located in the DDR3 memory. These register determine the behavior of the memory, e.g burst length. The following three steps are needed to guarantee correct timing between memory interface and memory. Write leveling compensates for skew between the data strobe and clock. Write calibration and read leveling takes care of skew between data and strobe. The last calibration step, read phase detector calibration, synchronizes the clock which samples data from memory with its corresponding data strobe.

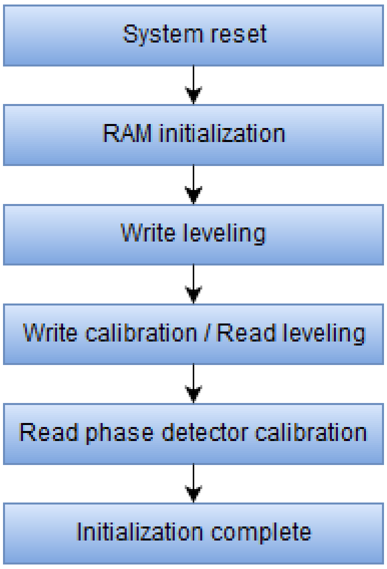


Figure 4.8: Memory initialization process performed by the physical layer in the memory interface.

Chapter 5

Verification

Three layers of tests were performed to confirm the expected behaviour of the system. By following this structure, bugs were easier to find and the pace of our work flow increased.

5.1 Test bench

Before moving on to the actual hardware, the easiest way to debug the Register Transfer Language (RTL) code was to simulate with a test bench. The stimuli to the code that was to be tested could be customized freely with a specified timing. This way corner cases could be tested without the need to navigate to the specific state in which they occurred. To make this process as effective as possible, only small blocks were tested one at a time. Otherwise, it would have been hard to find the origins of the problems.

There were also some clear disadvantages in using test benches while debugging designs, such as long simulation times and test bench errors. These errors were created by uncertainties of how the data from black boxes would look like. Examples of such boxes are the lower levels of the PCIe layers and the physical data from the software.

5.2 Chipscope

By adding a Chipscope core to the ISE project it was possible to see wave forms in real-time on the FPGA. The desired trigger and data signals that were to be observed have to be specified in the core after synthesis in ISE. The signals could then be examined in Analyzer after the whole design programmed onto the Virtex6.

5.3 Software tests

Software scripts were provided at the start of the project to be able to test the system from a user perspective. The scripts made it possible to write and read to/from both the register file and DDR3 memory. Also, a large final test of writing and reading the whole memory from top to bottom was used to find bugs that only appeared during large transactions. All software tests were written in C and worked as procedures to the software kernel.

Chapter 6

Results

Here the results are presented in two parts. One part describes how big the design got in terms of resources and the other part describes the system performance in terms of speed at the end of the project.

6.1 Design resources

The results from the place and route report is divided into two parts. First, shown in Table 6.1, is the device utilization. Utilization describes the amount of resources calculated to be used in the design, such as memories and registers. The reported distribution, shown in Table 6.2, describes how the resources are spread across the Virtex6. The FPGA is divided into so called slices, each containing four LUTs and eight flip-flops [11]. The number of used block RAMs are shown in Table 6.3

Table 6.1: Device utilization extracted from place and route report.

Type			Amount
Slice LUT	Logic		13531 out of 80000
	Route-thrus		1248
	Memory	Dual Port RAM	3310
		Shift Register	985
Slice Register			24702 out of 160000

Table 6.2: Logic distribution extracted from place and route report.

Type		Amount
Occupied Slices		9014 out of 20000
LUT Flip Flop pairs	Unused flip flop	7477 out of 29043
	Unused LUT	9,969 out of 29043
	Used LUT-FF pairs	11597 out of 29043

Table 6.3: Part of specific feature utilization from place and route report.

Type	Amount
RAMB36E1	21 out of 264
RAMB18E1	2 out of 528

6.2 Performance

6.2.1 Clock frequency

In the design there are 7 different clock sources. The ADCs are paired into five clock pairs driven on different speeds. All inside the span of 100 - 130 MHz. Our system is driven by two clock sources. One PCIe reference clock and a system clock. The PCIe reference clock runs at 100 MHz while the system clock runs at 125 MHz. Except from these the system clock is used to create the memory clock which runs at 400 MHz.

6.2.2 Read

The read functionality was calculated since the software tests were not precise enough. The calculations were based on the fact that every package is sent continuously. One package contains three headers of 32 bits and 32 blocks of data of 32 bits. These packages can be sent once every 18 clock cycles, with the PCIe bus clocked at 125 MHz. This gives a data transmission rate of 847 MB/s.

6.2.3 Write

The write functionality was tested by performing a large write of the whole 2 GB memory. The average time for one write of 2 MB was determined to 0.130 seconds. This leads to an average transmission rate of 15 MB/s.

Chapter 7

Analysis

This chapter goes through the thoughts and ambitions drawn from this thesis. Conclusions summarizes our thoughts, whilst future work describes what could be accomplished with more time. Lessons learned describes problems encountered and how they were solved, providing the reader with the knowledge to not duplicate the same mistakes.

7.1 Conclusions

Looking back at the goals, described in Section 1.2.2, all of the parts planned for build were completed. In addition, read/write arbitration is working and therefore completes the optional, final goal. That said, although the results are satisfying, the working process did not work out as planned in the goals. The main reason can be attributed to our decision of using the AXI interface, which meant that we could not test our components with the pre-existing blocks since they were incompatible. On the other hand, this also accelerated the development of the arbiter goals since the AXI interconnect has an easy to implement arbiter.

When observing the results, shown in Chapter 6, the read functions are about 56 times faster than the writes. This is to be expected for reasons further described in Section 7.4. The reads are more relevant since the ADCs are the main parts writing to memory, which also makes the overall results satisfying. When comparing the results to the original version we have found that we have the same performance.

A PCIe interface with our specifications, a 4x lane width and 2.5 Giga Transfers per second, has a maximum transfer speed of 1GB/s. Because of protocol overhead and delays a part of the transfer speed is lost as shown in the result of 847 MB/s of the read calculations shown in Section 6.2. Generally the theoretical throughput of the link can be calculated with equation 7.1, where PL is the payload size, OV the protocol overhead, N the number of lanes and G is a parameter dependent on what generation/version of the protocol is used [13].

$$Throughput_{THEO} = \frac{PL}{PL + OV} \cdot N \cdot G \cdot (250MB/s) \tag{7.1}$$

7.2 Lessons Learned

Chipscope

Chipscope proved to be a powerful tool during the debugging process, although it sometimes was hard to handle. To find all relevant signals was not always possible since, as mentioned in Chapter 5.2, Chipscope is inserted into the work flow after synthesis. The chosen signals could be gone since the design has been optimized during synthesis. To solve this problem a keep attribute had to be introduced and applied to the desired signals in the RTL code.

Isim

After using a lot of test benches, both self written and predefined by Xilinx, we have had a lot of problems with Isim. Most of the problems have been worked around by changing computer or by specifying the wave file which is going to be run in ISE before even launching Isim. These problems were hard to predict and find workarounds for. Later on in the project we switched simulation program to Questasim which was a lot easier to handle. Questasims .do file is also a lot easier to handle and more versatile then Isims .wfcg which only configures waveforms.

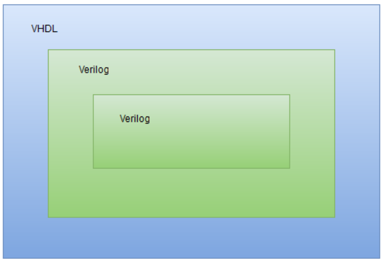
IP cores

The IPs that Xilinx offers, such as the AXI interconnect or PCIe interface, have shown to possess a lot of features of which not all are supported for all scenarios. One of these features is the PCIe interfaces option to obtain bus master status through accessing the configuration ports. This bus mastering process was part of the old version of PCI and are nowadays resolved internally in the PCIe block without the need for user interference. A lot of these details were resolved simply by putting more time into reading the documentation.

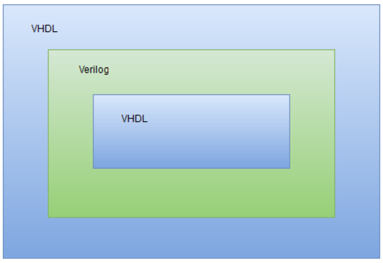
Because of the high simulation time for DDR3 memory and the lack of software side on the PCIe interface we built our own simulation modules for these blocks. By doing this we missed some key features that did not work in practice e.g. memory not sending continuous packages and the PCIe block not being able to receive discontinuous packages.

VHDL/Verilog

Our preferred writing language when the project was started was VHDL. When we started to import some of Xilinx's IPs we realised that not all of the implementation options that existed for Verilog, were present for the same block in VHDL. An example is the AXI connectivity option for the memory interface, which only could be chosen in Verilog. This resulted in us learning a lot more about Verilog but also about how a multiple language hierarchy is allowed to be constructed. If the top layers of the design are made in VHDL and a sub block is implemented in Verilog, no sub block of the Verilog code is allowed to be written in VHDL, as shown in Figure 7.1a and 7.1b. This created a problem when the memory interface needed to be implemented in Verilog to get access to the AXI interface. ISE needs to be set to have preference language Verilog for this to work, which made the rest of the source code to violate the hierarchy structure. This was fixed by generating the Verilog code in a new project, enabling us to extract the source code and then add it to the current working project.



(a) Correct



(b) Incorrect

Figure 7.1: Illustration of correct/incorrect RTL code hierarchy.

AXI interconnect

The AXI interconnect core was one of the main blocks in our design. It provided many of the features we needed such as clock- and width conversion and arbitration. One disadvantage was the fact that it did all these things. It is harder to debug a design when it is not your own and you need to rely on documentation to get a working design. We came to the conclusion that a feature called packaging, which was supposed to wait to release data until whole packages had been received, was needed. This was only working properly on the write channel. The problem lead to timing problems on the read channel, when memory did not send continuous data. This was fixed by inferring an extra FIFO between the DMA and Egress, which stacked packages into continuous chunks before transforming into TLPs.

Modifying memory command sequence

Nothing works as intended. That is one of the main lessons we have learned. In the best of worlds you need to generate a IP core and then its plug and play. In this case the memory interface used a chip select signal to disconnect the DDR3 memory. This signal was grounded on the Printed Circuit Board and therefore the inhabit command could not be used as in Table 7.1[12]. This would have made the memory interface unusable but all IPs from Xilinx are provided with the actual source code. By changing the state machine of the memory interface to send no operation instead of inhabit commands this problem was resolved.

Table 7.1: SDRAM command control signals

CS	RAS	CAS	WE	Command
1	-	-	-	Command inhabit (No operation)
0	1	1	1	No operation
0	1	1	0	Burst Terminate
0	1	0	1	Read burst
0	1	0	1	Read with auto precharge
0	1	0	0	Write burst
0	1	0	0	Write with auto precharge
0	0	1	1	Activate
0	0	1	0	Precharge
0	0	1	0	Precharge all
0	0	0	1	Auto refresh
0	0	0	0	Load mode register

7.3 If we were to do it again

The first thing that was noticeable was the documentation part. We lost a lot of time to things that could have been predicted by reading documentation more closely. We read relevant pages for the specific part we were working on, when we should have taken the time to get a better overview.

We should have created better simulation modules for the black boxes in the design. Chipscope is a powerful tool but the time to synthesize, place and route and program the FPGA is a lot of overhead to get the conclusion that it is not working.

Questasim was a great improvement from Isim. Isim had some nice features but Questasim was more reliable and user friendly. Therefore to save both time and frustration we would have switched to Questasim.

A more defined time schedule would have been helpful. We made one in the beginning and stayed with it for 5 weeks. After that we fell behind and should have created a revised version.

7.4 Future work

Reset

One problem that we have not resolved yet is an enumeration error that occurs when we start the system for the first time. The problem is quickly resolved by restarting the enumeration process one time. A similar problem occurs in the memory interface once the system starts, where the DDR3 memory initialization process does not finish. Here also, the problem is resolved by an extra reset. Combining these two problems concludes that there probably is an error somewhere in the reset logic.

ADC

The extra ADC inputs to the interconnect have been simulated and tested. The concept has been proven to work, although with some errors and not with the full amount of ADCs. To complete this part of the project not only in simulation or in an incomplete test environment would clearly be the next step forward.

DMA write

The solution when writing data to memory can be updated into a faster version. As it works today the computer is only able to write 64 bytes to memory at a time. After that one must update the DMA registers and start again. What should be possible is to write the registers once with a large length. This would have made the design faster and is definitely worth implementing.

The problem with the fast implementation is that the hardware has no control over how completions is sent back after read requests has been sent upstream. Completions can be received with different lengths and can be reordered. The reordering would need to be handled by applying unique tags to the read requests. This means that the completions would also have the unique tags and could have been ordered in that manner. Combining the completers byte count- and length field, refereed to in section 2.1.5, can be used to determine how many byte is left of a completion and how large the current package is. This would have been helpful when stacking data before sending it to the DDR3 memory.

Maximum payload size

Looking at the configurations of the PCIe block, a possibility of increasing the maximum payload size is possible. As of today it is set to be 128 bytes while the limit is at 1024 bytes. The burst length of the AXI interface has a maximum of 256 QW which is at most is a payload size of 2048 bytes. This means that if the PCIe maximum payload size would increase the AXI interfaces in the design would still be able to cope with the data sizes. This would have increased the speed of the system as well as minimized the number of addresses sent to the memory.

References

[1] ESS homepage,
`https://web.archive.org/web/20140517050510/http://europeanspallationsource.se/`,
August 2014

[2] ESS Technical Design Report 2013,
`http://docdb01.esss.lu.se/DocDB/0002/000274/015/TDR_online_ver_all.pdf`,
January 2016

[3] F. Qiua, J. Gaoa, H. Lina, R. Liua, X. Maa, P. Shaa, Y. Suna, G. Wanga, Q. Wanga, B. Xua, R. Zengc,"A new IQ detection method for LLRF", *ScienceDirect*, vol 675, pp. 139-143, 2012

[4] Ravi Budruk, Don Anderson and Tom Shanley,
PCIe Express System Architecture,
MindShare INC, First Edition, 2003

[5] PCI Express 2.0 base specification,
`http://read.pudn.com/downloads95/ebook/383403/PCI_Express_Base_Specification_v20.pdf`,
January 2016

[6] AXI reference guide,
`http://www.xilinx.com/support/documentation/ip_documentation/ug761_axi_reference_guide.pdf`,
November 2015

[7] ISE project nevigator product description,
`http://www.xilinx.com/tools/projnav.htm`,
January 2016

[8] User guide for Xilinx PCIe core,
`http://www.xilinx.com/support/documentation/ip_documentation/v6_pcie/v2_5/ug671_V6_IntBlock_PCIe.pdf`,
January 2016

[9] AXI interconnect product description,
`http://www.xilinx.com/products/intellectual-property/axi_interconnect.html`,
November 2015

[10] Xilinx Memory Interface Generator product description,
`http://www.xilinx.com/products/intellectual-property/mig.html`,
October 2015

[11] Virtex-6 Family Overview,
`http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`,
January 2016

[12] Command table for DDR3 SDRAM,
`http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/nanya-nt5ds-datasheet.pdf#page=13`,
January 2016

[13] L. Rota, M. Caselle, S. Chilingaryan, A. Kopmann, M. Weber, "A new DMA PCIe Architecture for Gigabyte Data Transmission", *Real Time Conference (RT)*, pp. 1-2, 2014

LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2016-482

http://www.eit.lth.se