Development of an Open PXIe System based on FPGA Modules

by

Andrew Ang

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Master of Engineering
in Electronic and Computer Systems Engineering.

Victoria University of Wellington
2018

Abstract

PXIe is a instrumentation platform that is used as the basis for developing test equipment, modular electronic instruments and automated test systems. A typical PXIe platform comprises of modules that are housed in a 3U Eurocard form factor chassis. The platform utilise the PCIe bus standard to enable high-speed data transfer, suitable for a range of applications. However, the platform is inherently complex, contains proprietary IP and is prohibitively expensive for most researchers and engineers who wish to utilise a modular instrumentation system. To overcome these barriers, the beginnings of an open PXIe platform has been developed. This consists of two open PXIe modules utilising modular FPGA technology. The first module is the System Controller, which introduces an embedded Linux solution and open-source PCIe driver to the platform. To simplify software development, user applications can utilise the drivers API without using kernel-level code. On the System Controller FPGA fabric is a root-port implementation that allows communication with peripheral modules. The second modules is a general purpose Peripheral Module that provides various I/O standards that users can utilise in their FPGA design, and a dedicated PCIe generation 2 x4 link with the System Controller. In the FPGA fabric of the Peripheral Module is a PCIe-DMA engine that facilitates data transfer between the two modules. The open nature and modular design will allow more economical and more flexible solutions, which will be appealing to a wide range of potential users. In addition, an example user application is developed for this system to show case the overall functionality the modules, with transfer speeds of 100 MB/s.

Acknowledgments

I am indebted to my supervisor, Dr. Robin Dykstra, for his support in this project and providing me the opportunity to undertake this project. He shared his wealth of experience and provided helpful constructive criticism to ensure that the project would be a success.

I would like to extend my thanks to the NMR group residing in Laby 308, for giving me a place to do this project and getting me involved in the area of NMR research.

Finally, I wish to thank my family and friends for their support and encouragement throughout my years of study.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Objectives	3
1.3	Thesis Overview	3
2	Background	5
2.1	Electronic Test Equipment	5
2.2	Instrumentation Platforms	6
2.2.1	VME eXtensions for Instrumentation(VXI)	6
2.2.2	PCI express eXtensions for Instrumentation (PXIe) . .	7
2.2.3	LAN eXtensions for Instrumentation (LXI)	8
2.2.4	Evaluation of Instrumentation Platforms	9
2.3	Computer Systems Overview	9
2.4	PCIe Architecture	10
2.4.1	Definitions	11
2.4.2	Architecture Layout	12
2.4.3	Configuration of PCIe Devices	21
2.5	PXIe Signals	26
2.5.1	PXI Signals	26
2.5.2	PXIe Synchronisation	26
2.6	Operating System	27
2.6.1	Windows	29
2.6.2	Linux	29

2.6.3	Selection of OS	30
2.7	Integrated Circuit Technology	31
2.8	Investigation of I/O Standards	33
2.9	Existing Solutions	35
2.10	Previous PXIe Modules for MRI Instrumentation Develop- ment	39
2.10.1	RF Transceiver Board	39
2.10.2	PXIe System Controller with Zynq 7045 SoM	40
2.10.3	PicoZed System Controller Implementation	41
3	System Controller Hardware Design Review	43
3.1	System Controller Mechanical Construction	43
3.1.1	Connectors	44
3.1.2	PCB Dimension	45
3.2	Verification of the PicoZed Interface	46
3.2.1	Verification of connector positioning	46
3.2.2	Verification of PicoZed Pinout	48
3.3	PCIe Connectivity of the System Controller	50
3.3.1	System Controller Clock System	50
3.4	Implemented Peripherals	51
3.4.1	microSD	52
3.4.2	UART	54
3.4.3	USB OTG	54
3.4.4	Ethernet	54
3.5	System Controller Power System	55
3.5.1	PicoZed Power Requirements	56
3.5.2	PicoZed Power Estimation	57
3.5.3	PicoZed Power Sequencing	60
3.6	System Controller Fabrication Process	60
3.7	Summary	62

4	Prototyping and Proof of Concept	63
4.1	Components	63
4.2	Test Set-up	64
4.3	PCIe Connectivity Testing	66
4.4	Summary	67
5	Peripheral Module Hardware Design	69
5.1	Overview	69
5.2	Open PXIe System Hardware Objectives	69
5.3	Peripheral Module Mechanical Implementation	72
5.3.1	Connectors	72
5.3.2	Peripheral Module Skeleton PCB	74
5.4	Interfacing with a Modular FPGA Solution	75
5.4.1	Peripheral Module FPGA Selection	75
5.4.2	Peripheral Module Board-to-Board Connectors	76
5.5	High Speed Design	77
5.5.1	PCB Layout	77
5.5.2	PCB Layer Stack	79
5.5.3	Characteristic Impedance	79
5.5.4	Routing Guidelines	81
5.6	PCIe Hardware Design	83
5.6.1	GTP Architecture	84
5.6.2	PCIe Implementation on the Peripheral Module	86
5.7	Clock System	86
5.7.1	Peripheral Module Clock System	87
5.8	Peripherals	89
5.8.1	UART	90
5.8.2	Pmod Interface	90
5.8.3	Joint Test Action Group (JTAG)	91
5.8.4	FPGA Mezzanine Carrier (FMC)	91
5.9	Power Design	92

5.9.1	Power Requirements	93
5.9.2	TE0712 Power Estimation	94
5.9.3	Power Sequencing	95
5.10	Fabrication of the Peripheral Module	96
5.11	Summary	97
6	Firmware Development	99
6.1	Open PXIe System Firmware Objectives	99
6.2	Xilinx Toolset	100
6.2.1	Vivado Design Suite	102
6.2.2	Petalinux	102
6.3	Advanced eXtensible Interface (AXI) Bus	103
6.4	System Controller FPGA Firmware Design	103
6.4.1	Zynq Processing System	104
6.4.2	PCIe Connectivity	104
6.5	Peripheral Module FPGA Firmware Design	107
6.5.1	FMC ADC Interface Example	108
6.5.2	Microblaze Firmware Development	112
6.5.3	PCIe Connectivity	113
6.6	Linux OS Development	115
6.7	Linux PCIe Driver	117
6.8	Application User Interface (API) for Users	121
6.9	Summary	123
7	Design Evaluation and Analysis	127
7.1	Mechanical and Electrical Testing and Evaluation of the De- veloped Modules	127
7.2	FPGA Design Evaluation	128
7.2.1	Static Timing and Routing Evaluation	129
7.2.2	Resource Utilisation	133
7.2.3	Functional Verification	136
7.3	Testing of an ADC Firmware using Prototype System	144

7.4	Driver and API Verification	147
7.4.1	Loading of the PCIe driver on a Linux OS	148
7.4.2	Example User Application	150
7.5	Overall Cost	153
7.6	Summary	154
8	Conclusions	157
8.1	Review	157
8.2	Future Work	160
8.3	Summary	162
	Appendices	175
A	Peripheral Module Hardware Schematics	177
B	System Controller FPGA Firmware	187
C	Peripheral Module FPGA Firmware	189
D	2017 16th International Conference on Field-Programmable Technology (ICFPT)	191
E	2017 11th International Conference on Sensing Technology (ICST)	197

List of Figures

2.1	VXI 6U form factor instrumentation platform [1]	7
2.2	PXIe 3U form factor chassis [2]	8
2.3	PXIe 3U form factor chassis [3]	11
2.4	PCIe architecture device layers	13
2.5	PCIe packet architecture	13
2.6	Data link layer architecture [4]	15
2.7	Physical layer transmit architecture [4]	17
2.8	High level overview of PCIe data scrambling	18
2.9	High level overview of 8b/10b encoding process	20
2.10	PCIe configuration address space	24
2.11	Typical PCIe topology with root complex bus 0 highlighted [4]	25
2.12	BAR structure after a base address is assigned by an oper- ating system	25
2.13	Distribution of PXIe and PXI signals in a PXIe instrumenta- tion system. Slot 1 is a system controller slot. Slots 2 and 3 are PXIe/PXI peripheral module hybrid slots. Slot 4 is a combined PXIe timing module and peripheral module slot. Slots between 5 and 8 are PXI peripheral module slots. [5] .	28
2.14	An FMC HPC male (top) and female (bottom) connector [6]	34
2.15	PXIe-7961 peripheral module from National Instrumnts [7]	36
2.16	SPEXI FMC carrier from CERN [8]	37
2.17	PXIe700 peripheral module from Sundance Technology [9] .	38

2.18	RF Transceiver Peripheral Module [10]	40
2.19	Previously implemented PXIe system controller with Zynq 7045 [10]	41
2.20	PicoZed System Controller 3D rendering of designed board	42
3.1	PXIe System Controller backplane connectors [5]	45
3.2	System Controller backplane connectors implementation . .	46
3.3	System Controller hardware implementation with width and height measured on Altium	47
3.4	System Controller layer stack from PCB manufacturer	47
3.5	Avnet's PicoZed 7015 Module [11]	48
3.6	PicoZed 7015 footprint from Avnet [12]	49
3.7	System Controller PCIe connectivity schematic	50
3.8	Schematic of System Controller's PCIe connections	51
3.9	System Controller Clock System	52
3.10	microSD card interface	53
3.11	microSD card schematic	53
3.12	PicoZed's Ethernet implementation	55
3.13	Ethernet data connections on PicoZed	55
3.14	RJ-45 connector with integrated magnetics connections	56
3.15	System Controller power architecture	57
3.16	XPE settings for PicoZed power estimation	58
3.17	PicoZed power supply estimation	59
3.18	Affected narrow pads on System Controller hardware design	61
3.19	Built System Controller module	62
4.1	Zynq Mini-ITX development board from Avnet, emulating the PXIe system controller [13]	64
4.2	Zynq PicoZed with carrier card from Avnet, emulating the PXIe peripheral module [14]	65

4.3	Zynq Mini-ITX development board in chassis (1), PCIe extender (2) and PicoZed with Carrier Card (3) in prototype set-up.	66
4.4	PicoZed endpoint <i>lspci</i> output from Linux command line . .	67
5.1	Proposed open PXIe hardware architecture	70
5.2	Peripheral Module high-level architecture	72
5.3	PXIe Peripheral Module backplane connectors [5]	73
5.4	PXIe Peripheral Module skeleton PCB	74
5.5	Trenz Electronics' TE0712 Module [15]	76
5.6	TE0712 hermaphroditic board-to-board connectors (highlighted red)	77
5.7	Peripheral Module layout arrangement	78
5.8	Layer stackup for the System Controller and Peripheral Module	80
5.9	An example of routing over a split plane [16]	81
5.10	An example of a ground reference plane (highlighted) on layer two	82
5.11	An example of a length matched differential signal (highlighted blue) on the FMC Peripheral Module	83
5.12	An example of AC coupling capacitor placement to reduce potential reflection [16]	84
5.13	GTP TX and RX transceiver architecture [17]	85
5.14	Schematic of Peripheral Module's PCIe connections	87
5.15	Peripheral Module Clock System	88
5.16	Conversion of LVDS to single-ended signal for SMA output	89
5.17	Mechanical dimensions of a single width FMC card [18] . . .	92
5.18	Peripheral Module power architecture	94
5.19	XPE settings for TE0712 power consumption estimation . . .	95
5.20	TE0712 power supply current draw	96
5.21	Populated Peripheral Module PCB with AD9467 FMC card .	98

6.1	High-level firmware architecture	101
6.2	Peripheral configuration in Zynq PS core	105
6.3	High-level AXI-PCIe architecture [19]	106
6.4	System Controller Zynq PCIe BAR configuration	106
6.5	Overall FPGA firmware architecture for Peripheral Module including AD9467 and PCIe connectivity	108
6.6	AD9467 timing diagram [20]	109
6.7	Bit extraction RTL schematic	110
6.8	IDDR primitive timing diagram [21]	110
6.9	AD9467 interface data flow to DDR memory with DMA controller	111
6.10	PCIe-DMA implementation	114
6.11	Enabling PCIe support during kernel configuration	116
7.1	Developed System Controller (1) and Peripheral Module (2) in PXIe-1062Q chassis	129
7.2	System Controller's FPGA paths that did not meet setup time	132
7.3	Paths that did not meet setup time in AXI-PCIe RTL schematic	132
7.4	Areas of congestion with utilisation greater than 70%	133
7.5	System Controller's FPGA paths that failed timing repre- sented on Zynq 7045	134
7.6	Testbench for AXI-PCIe in root-port configuration	137
7.7	Verification of root-port sending 0xa4a4a4a4 memory write transaction from root-port to endpoint model	138
7.8	PCIe transceivers toggling during transmission	139
7.9	Verification of memory read transaction from end point model to root-port	141
7.10	System reset de-asserted at timestamp 4.995 μ s.	142
7.11	Reading Device and Vendor ID from the endpoint's BAR on sixth row of waveform diagram	143
7.12	Verification of root-port sending 0x61626364 to AXI-PCIe endpoint as part of memory read transaction	145

7.13	Verification of root-port receiving 0x61626364 from AXI-PCIe endpoint memory write transaction	146
7.14	100 KHz sine wave capture with an AD9467 ADC consisting of 2048 data points	147
7.15	Result of loading PCIe driver on Linux OS	148
7.16	Peripheral Module endpoint <i>lspci</i> output from Linux command line	149
7.17	PXle-6535 <i>lspci</i> output	149
7.18	Peripheral Module polling routine	152
7.19	System Controller capture-polling routine	153
8.1	open PXle platform with developed System Controller, Peripheral Module and PXle-1062Q chassis from National Instruments [3]	163

List of Tables

2.1	Overview of bus commands	10
2.2	PCI Configuration Space Header for a PCI Type 0 (Non-Bridge) Device	22
2.3	PCI Configuration Space Header for a PCI Type 1 Device . .	23
2.4	PCI Express Capability Structure of Endpoint Devices	23
5.1	Impedance parameters for the Peripheral Module	80
5.2	Backplane current capacity for each slot in PXIe-1062Q chassis [2]	93
6.1	Peripheral Module AXI address space	114
6.2	Overview of the open PXIe platform API	123
7.1	System Controller top 3 FPGA firmware resource utilisation on the Xilinx Zynq 7z015clg485-1	135
7.2	Peripheral Module top 3 FPGA firmware resource utilisation on the Xilinx Artix-7 7a100tfgg484-2	135
7.3	Component cost breakdown for developed System Controller and Peripheral Module	154

List of Acronyms

ADF	Advanced Differential Fabric
ADC	Analogue-to-Digital Converter
ANSI	American National Standards Institute
API	Application Programming Interface
ATE	Automated Test Equipment
AXI	Advanced Extensible Interface
BAR	Base Address Register
CDO	Clock Domain Crossing
CPU	Central Processing Unit
DCO	Data Clock Output
DDR	Dual Data Rate
DLL	Data-Link-Layer
DLLP	Data-Link-Layer Packet
DMA	Direct Memory Access
EHM	Enriched Hard-Metric
ELF	Executable and Linkable Format
FMC	FPGA Mezzanine Card
FPGA	Field Programmable Gate Array
HCSL	High Speed Current Steering Logic
HPC	High Pin Count
IC	Integrated Circuit
I/O	Input/Output
IDDR	Input DDR Resources

IP	Intellectual Property
JTAG	Joint Test Action Group
LPC	Low Pin Count
LUT	Look Up Table
LVDS	Low Voltage Differential Signal
LVPECL	Low Voltage Positive Emitter Coupled Logic
MIO	Multiple Input/Output
NMR	Nuclear Magnetic Resonance
OS	Operating System
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
PCIe	PCI Express
PCI-SIG	Peripheral Component Interconnect Special Interest Group
PL	Programmable Logic
PLL	Phase Lock Loop
PS	Processor System
PXIe	PCI eXtensions for Instrumentation
PXISA	PXI Systems Alliance
RTL	Register Transfer Level
SoC	System-on-Chip
SoM	System-on-Module
SPI	Serial Peripheral Interface
SRAM	Static Random-Access Memory
TL	Transaction Layer
TLP	Transaction Layer Packet
UPM	Universal Power Modules
USB	Universal Serial Bus
VUW	Victoria University of Wellington
XPE	Xilinx Power Estimator

Chapter 1

Introduction

Electronic test equipment is used to create signals and capture responses from electronic systems under test. Through test equipment, proper operation can be determined or faults within the system can be traced. Examples of electronic test equipment range from a simple debugging LED, to a full 2 m high 19 inch rack, Automated Test Equipment (ATE). As the area of electronics becomes more wide-range, it is common to see multiple test equipment occupying a work bench. An emerging new trend is replacing separate test equipment with a single rack system. These rack systems can house multiple modules that carry the same functionality, while extending the range of testing applications due to its modular nature and having the modern convenience of ATE systems through on-board computer modules. A electronic test equipment rack can facilitate development and debugging, while having the benefits of modularity and ATE features.

1.1 Motivation

A conventional test station comprises of multiple, standalone instruments that can measure different parameters in a circuit, provide stimulus and analyse the response. Example of core electronic instruments are multi-meters, power supplies, signal generators and oscilloscopes. These instru-

ments can take up a large portion of bench space, reducing the size of a working area. An alternative solution is to utilise a rack system with a smaller physical footprint. Test equipment and a desktop computer can be replaced with a rack system module counterpart. Not only is a rack system useful for ATE, but for some sophisticated systems such as Magnetic Resonance Imaging (MRI) systems, the modular nature is an essential requirement. The only external component required to complete a rack instrumentation set up is a computer monitor. In turn, the small area space associated with test rack systems provides an appealing alternative.

Although a rack system has a smaller area footprint on a workbench, the cost of a rack platform is prohibitively expensive for most engineers and researchers. In addition, modules for rack systems require computer software and intellectual property (IP) that are sold separately. This further increases the price of a rack instrumentation system. A modularised system does provide many benefits in testing applications. However the high cost of purchasing such a system is not justified for most users.

Coupled with the high price tag, instrumentation platforms are inherently complex. The complexity of such platforms results in a high entry barrier for development and further customisation. This does not promote users to develop custom solutions. Instead, they are forced to purchase expensive hardware, along with proprietary software required for their application.

The School of Engineering at Victoria University of Wellington has utilised an electronic instrumentation platform for Magnetic Resonance Imaging (MRI) applications. The modularity of the platform allows construction of a sophisticated system for different experiments. A previous attempt has been made to develop two modules [22] for an standardised instrumentation system. However, development of custom instrumentation hardware is complex and utilised expensive components. A new simplified, in-house system is desired while being economical at the same time.

1.2 Objectives

The outcome of this thesis is to develop two core modules for a generic instrumentation platform. These modules will be a lower-cost and more flexible alternative to commercial instrumentation solutions. Their respective hardware design will be simplistic, allowing users to easily develop their own module solutions. Each module will be coupled with their respective firmware to implement their core functionality. This low-level firmware can be abstracted through an Application Programming Interface (API) to simplify software development on the platform. A versatile solution is also developed to allow the platform to be utilised in different applications. With commercial instrumentation systems, it is common to purchase different modules for different applications. By opting for a versatile solution, it minimises the cost of purchasing different modules by providing an all-in-one solution.

Together, the two modules form an open and versatile platform to facilitate hardware and software development for a range of applications.

1.3 Thesis Overview

This chapter introduced the motivation for an open instrumentation system with high-level objectives introduced in Section 1.2. Chapter 2 explores and discusses the area of instrumentation systems and existing solutions which are comparable to this project. These solutions are analysed for their merits that may be utilised in this project, and disadvantages that should be either avoided or improved upon in the design of the core modules. Chapter 3 reviews and verifies a previously designed module which is used later to develop an operating system and custom drivers. Chapter 4 discusses the development of a prototype system to emulate an instrumentation system. Chapter 5 and 6 detail the hardware design of the Peripheral Module and the firmware design of both modules respectively.

Both chapters introduce the specific objectives required for each aspect of the design, followed by a detailed approach to meet these objectives. Chapter 7 the hardware and firmware aspects of the modules against their respective objectives introduced in previous chapters. Finally, Chapter 8 concludes the successful thesis followed by a discussion of future work that can be done towards a complete open instrumentation system.

Chapter 2

Background

Introduced in Chapter 1, the motivation of this project is to develop two economical core modules for a generic instrumentation platform which can be utilised by engineers and researchers for different applications. The two modules need to conform to an existing instrumentation standard. This ensures compatibility with other solutions to keep the platform versatile if the user chooses to leverage other commercial products. Additionally, the functionality of the two modules will need to be decided. By exploring existing modules, it presents an opportunity to explore what engineers and researchers are expecting from an instrumentation system. This chapter presents the background and existing solutions in these areas.

2.1 Electronic Test Equipment

Modern test equipment usually consists of oscilloscopes, function generators, voltmeters and many others. These allow an engineer to create signals and capture responses from equipment under test. In this way, the equipment can be tested for faults, which can be traced back to its source. Around the 1990s, there was a demand for 'rack-and-stack' or chassis/mainframe-based test systems. The idea was to reduce the space associated with separate test equipment by compartmentalising different

equipment into a single chassis. This provides a dynamic test bench that can be customised for different testing applications. As electronic development became more widespread, electronic manufacturers required faster turn-around time. This led to automating electronic tests through specialised equipment called ATE (automatic test equipment). Measurements and evaluation of an electronic system can be made quickly, and results can be stored on a local server for future reference. The rise of ATE spawned multiple standards such as LXI, VXI and PXIe.

2.2 Instrumentation Platforms

It was realised that computers can be used for automated electronic test and measurement systems by combining with traditional, standalone electronic instrumentation. This led to ATE being widely employed in areas of inspection, quality assurance and production testing of electronic devices and sub-assemblies [23].

Industry-standard ATE equipment comprised of chassis-mainframe-based systems with signal sources linked to measurement instruments. These are often under the control of custom software applications running on an operating system or an external computer.

Introduced in this section are leading electronic instrumentation platforms that are considered for the proposed open platform.

2.2.1 VME eXtensions for Instrumentation(VXI)

VXI [24] was established in 1987 by a consortium of test and measurement companies. The goal was to develop a standard that meet the need for portable applications and to provide an open, modular architecture to integrate with the already established GPIB test system. The 6U form factor architecture (illustrated in Figure 2.1) is based upon an open standard platform, VMEbus, which defines bus lines for timing and trigger-

ing, mechanical requirements and standard protocols for configuration, communication and multi-chassis extension. The architecture allowed 40 MB/s data rates making it ideal for instrument systems that require high throughput. The standard has been very successful in military-aerospace test applications and high-channel-count applications in manufacturing test. However, VXI did not achieve mainstream usage because of its high cost and complicated integration. In addition, the underlying bus architecture was not a commonly found standard used outside of measurement and automation, and thus, there are only a handful of modules that the standard offers.

Figure 2.1: VXI 6U form factor instrumentation platform [1]

2.2.2 PCI express eXtensions for Instrumentation (PXIe)

PXIe is the successor of PXI, a modular 3U form factor instrumentation platform introduced in 1997 by National Instruments [5]. Designed for measurement and automation applications, there are over 1150 PXIe modules that meet those applications. Taking advantage of the increasing performance of computers, a typical PXIe setup contains a system controller, a complete computer with an operating system on a module, simplifying the hardware access and control to modules within the system. This is

contained within a PXIe chassis as illustrated in Figure 2.2. The system controller can access and control peripheral modules through PCIe. PXIe replaces the older PCI bus with the PCI express (PCIe), which increases bandwidth from 132 MB/s to 1 GB/s [5]. The standard allows for backward compatibility with older PXI equipment which might prove useful for users who have older PXI modules on hand. The open standard is supported by the PXI Systems Alliance (PXISA) with 66 major sponsor members [25].

Figure 2.2: PXIe 3U form factor chassis [2]

2.2.3 LAN eXtensions for Instrumentation (LXI)

In 2005, Keysight technologies and VXI technology, Inc. introduced LXI, an instrumentation and data acquisition standard using the Ethernet protocol to send data. It combines GPIB instruments, VXI and PXIe modules to provide the high throughput of LAN as well as test related features found in VXI and PXIe [26]. LXI is used more for distributed systems where different platforms can communicate and send data through the Ethernet protocol.

2.2.4 Evaluation of Instrumentation Platforms

The common theme between the instrumentation platforms is that they all allow add-in cards that can be used for different applications. However, the largest pitfall associated with these standards are high-cost and complexity. It is decided that the PXIe platform presents the best combination of transmission speed, availability and features. By utilising PCIe, the standard provides significant bandwidth required for potential MRI applications. With the PXIe 3U form factor, it is more convenient compared to the VXI 6U form factor. Additionally, two PXIe modules were previously developed by the School of Engineering Group at Victoria University of Wellington (VUW) so their designs can be leveraged for this thesis. Utilising the PXIe standard, the proposed core modules are a new System Controller and a Peripheral Module. The combination of the two modules provide a foundation for an open PXIe platform.

2.3 Computer Systems Overview

Computer systems generally consist of three main components: the central processing unit (CPU) that carries out instructions of a computer program, memory that stores data and instructions to be processed by the CPU, Input/Output (I/O) peripherals that moves data between the computer and its external environment, and a mechanism that allows communication between the components called the system interconnect.

A bus provides interconnection pathways between two or more components within a computer system. This allows for any components on a bus to transmit signals in the form of binary 1 and binary 0 to any device attached to the bus. However, only one device can transmit at any given time. If two devices send data at the same time, it will result in both signals overlapping and the data becomes corrupt. Multiple communication pathways, or lines, allow parallel signal transmission between two or

more devices.

Computer systems contain several different buses. A bus which connects major computer components (CPU, memory, I/O) is called the system bus. It consists, typically, of fifty to hundreds of separate lines [27] which serve different functions. Data lines are collectively called a data bus which provide a path for data to be transmitted among the system components. Common data bus width consists of 32, 64 or 128 lines wide. Communication through the data bus will require information for where to send data to. This is carried out by the address bus.

The control bus is used to control the access to the data and address bus. Signals on the control signals are either a command or timing information between the components. Timing signals provides the validity of data being sent on the data and system bus. While command signals specify operations that needs to be performed. Provided in 2.1 are typical bus commands.

Table 2.1: Overview of bus commands

Function	Description
Memory write	Writes data on the bus to be written into address location
Memory read	Reads data on the bus to be read at address location
I/O write	Writes data on the bus to be output to the addressed I/O location
I/O read	Reads data at I/O addressed location to be placed on the bus
Transfer ACK	Indicates that transfer has been accepted from or placed on the bus
Bus request	A component on the bus needs to gain control of the bus
Bus grant	A component is granted with control of the bus
Interrupt request	An interrupt is pending
Interrupt ACK	Acknowledges pending interrupt has been recognised
Clock	Used to synchronise operations
Reset	Initialises all components on the bus

2.4 PCIe Architecture

As the chosen PXIe standard implements a PCIe serial bus, the two module solutions will require a PCIe engine to handle the packet transactions.

The PCIe architecture is comprised of four hierarchical layers. The advantage of having a layered topology provides an easier migration to future PCIe versions by changing only one layer of architecture while leaving the others unaffected.

This section discusses the PCIe architecture and configuration of PCIe devices.

2.4.1 Definitions

Before explaining the PCIe layers, there are definitions that will aid in the understanding of the overall architecture. For reference, Figure 2.3 illustrates an example of a PCIe topology.

Figure 2.3: PXIe 3U form factor chassis [3]

Root Complex

The Root Complex (or root port) connects the processor and memory interfaces to the PCIe switch fabric. It may be implemented in a standalone

device or be part of the processor itself. A root complex may contain more than one interface that can be connected to a PCIe switch, a PCIe endpoint or to a PCIe bridge.

Switches and Bridges

Switches allow more than one device to be connected to a PCIe interface on the root complex. The role of a switch is to route packets to the correct destination based on the packet address.

PCIe bridges can be described as either forward or reverse bridges. In the case of forward bridges, they allow an older PCI endpoint to connect to a newer PCIe bus. Reverse bridges are the opposite. They provide a connection for older PCIe endpoints to a newer version of the PCIe bus. An example of a forward PCIe usage case is connecting a PCIe generation 2 endpoint to a PCIe generation 3 bus.

PCIe Endpoints

PCIe endpoints are devices on the PCIe bus that acts as the initiators and completers of transactions. They are located at the bottom of PCIe's inverted tree topology. In a modern computer, PCIe endpoints have different functionality including graphics card, USB hubs or Wi-Fi adapters.

2.4.2 Architecture Layout

The PCIe architecture consists of four device layers. From top to bottom, these layers are the: Device Core, Transaction, Data Link and Physical Layer. Each layer is logically split, with one side handling inbound traffic, while the other handling outbound traffic. The PCIe layers are illustrated in Figure 2.4.

Like traditional network architectures, PCIe is based on packets where each layer deals with one part of the packet. As a packet traverses down the device layers, the data payload becomes encapsulated. The reverse

Figure 2.4: PCIe architecture device layers

process occurs when a packet is traversing up the device layers. Figure 2.5 illustrates the encapsulation of data for each device layer.

Figure 2.5: PCIe packet architecture

Device Core

The device core is the top-most layer in the PCIe architecture. It provides the core functionality of the device. Although it is not defined in the PCIe specifications, it is either the source or destination of all PCIe requests. On its transmit side, it provides the transaction layer with requests, informa-

tion about the transaction type, the destination address and the amount of data to transfer.

Transaction Layer (TL)

The TL has two responsibilities: to generate outbound packets and examine incoming packets. When generating outbound packets, the TL does this by creating a Transaction Layer Packet (TLP). This is comprised of a header that defines the transaction parameters and an optional End-to-End Cyclic Redundancy Check (ECRC) used for error conditioning. After creating a TLP, they are sent to the next layer down, the Data Link Layer. Inbound packets arriving from the Data Link Layer are removed of their TLP header and ECRC. The TL layer does this by decoding the packets header which the information is passed onto the Device Core logic for appropriate actions. The TL may also check for any packet errors during transmission by checking and reporting any ECRC error condition to the Device Core. TLPs can be grouped into four request categories:

- Memory read/writes
- I/O read/writes
- Configuration read/writes
- Messages

Each category stated above can either be a posted or a non-posted request. It is expected for non-posted requests; the receiver needs to send a packet back to the requester to inform when a transaction is complete. If there is no completion packet, then the transaction encountered an error. For posted transactions, no packets are sent from either the requester or the receiver. This means that the requester does not get feedback if the transaction is completed or if it has encountered an error. However, posted transactions do not incur the overhead of completion which therefore improves performance.

Data Link Layer (DLL)

The DLL is responsible for link management and provides TLP error correction and flow control. These responsibilities are accomplished by generating DLL packets (DLLP). Outbound TLPs are received by the DLL and are given a sequence ID and a Link Cyclic Redundancy Check (LCRC). The DLL transmitter records a copy of every TLP that has been sent in a replay buffer. Until the destination sends back a confirmation that it has received the packet in the form of an *Ack* DLLP with its sequence ID, the DLL transmitter will flush the TLP with the corresponding sequence ID. If the DLL receiver does not receive an *Ack* DLLP, it generates an internal *Nak* DLLP which is sent to the transmitter to not acknowledge the reception of a DLLP. The transmitter receives the *Nak* DLLP then resends the TLP with the corresponding sequence ID in the hopes of successfully sending a DLLP packet to the destination. This mechanism is called the *Ack/Nak* protocol which provides the error correction functionality of the DLL. A high-level DLL architecture is illustrated in Figure 2.6.

Figure 2.6: Data link layer architecture [4]

Physical Layer

At the bottom of the PCIe hierarchical layer is the Physical Layer. TLP and DLLP packets are forwarded down from the Transaction and Data Link Layer to the Physical Layer's transmitter for transmission over the PCIe's physical link. The receiver does the reverse process of forwarding TLP and DLLP packets to the DLL. In the PCIe specifications, the Physical Layer is split into two portions: The Logical and Electrical sub-blocks.

The Logical portion consists of digital logic to process packets for serial transmission as well as processing inbound packets destined for the DLL and TL. This consists of different elements that make up the transmit logic:

- Framing
- Byte striping
- Scramblers
- 8b/10b encoding
- parallel-to-serial converters

The receiver logic decodes inbound packets by doing the reverse process of the transmit logic.

The electrical sub-block provides the delivery mechanism along the physical link. On the transmit side, it converts the serial bit stream to electrical signals that will be sent along the link. The receiver portion recreates the bit stream from the electrical signals being received.

The overall transmit architecture of the PCIe physical layer is summarised in Figure 2.7. The receiving architecture is similar which handles the conversion from electrical signals on the PCIe link to a DLLP.

Framing

When TLPs and DLLPs arrive at the Physical Layer, they are clocked into a buffer where start and end characters are added. This is illustrated in

Figure 2.7: Physical layer transmit architecture [4]

Figure 2.5. By adding start and end characters on the ends of the packet, it allows detection of the start and end of a packet on the receiver side of the physical layer.

Byte Striping

If a PCIe port implements more than one lane, then for each consecutive outbound data they need to be routed onto consecutive lanes. This process is called byte striping, where one byte of the packet is placed on an active lane. The lanes of a PCIe link transmit simultaneously. As a result, the byte striping logic must be fast enough to accommodate for all the lanes transmitting at the same time. Figure 2.7 illustrates the byte striping logic

after the transmit buffer.

Scramblers

The intention of data scrambling in PCIe is to prevent repetitive patterns in the data stream. Repetitions in the data stream can create resonance tones which can interfere with adjacent PCIe lanes and surrounding signals. The aim of data scrambling is to spread the frequency spectrum to ensure that there are no single frequency components that can create these resonance tones. It is achieved through an algorithm comprised of a 16-bit Linear Feedback Shift Register (LFSR) which acts as a pseudo-number generator. Each output bit from the LFSR is XORed with each bit of the data being transmitted. A high level block diagram of a PCIe data scrambler is illustrated in Figure 2.8.

Figure 2.8: High level overview of PCIe data scrambling

8b/10b Encoding

Embedding a clock removes the need for a global clock to be sent in parallel with the data. The benefits of removing the need for a global clock is that it eases length matching constraints of PCIe data traces. In PCI, both data traces and clock signals must arrive at the same time or mismatch will occur. This is done by "snaking" traces which can complicate the PCB design and increase board space. At higher frequencies, mismatch becomes more evident where very small differences in traces can lead to multiple bits out of place. Therefore, the speed that PCI operates in is restricted to only 33 MHz. With an embedded clock, PCIe routing is simplified and allows higher frequencies to be used thus increasing the overall system bandwidth.

Implementing 8b/10b encoding maintains a DC balance by equalising the number of one bits and zero bits being transmitted. When a PCIe line is driven, the DC value for that line changes. If the signal average is predominantly one level or the other, it can lead to a lane being over charged. This degrades signal integrity at the receiver. From PCIe specifications, the 8b/10b format allows at most five bits with the same polarity. This provides enough 0-to-1 or 1-to-0 transition density to allow the clock to be recovered at the receiver. The encoding process partitions the 8-bit character into its 3-bit and 5-bit sub-block. It then gets encoded to a six-bit block and four-bit block respectively through a lookup table. Although the encoding process adds at least 25% of overhead, the benefits of having an embedded clock outweighs this disadvantage. A high level overview of the 8b/10b encoding process is illustrated in Figure 2.9.

Electrical Sub-Block

In contrast with the PCI architecture, PCIe uses differential signalling as opposed to single ended signalling, where all signals are referenced to ground. With a noisy ground plane, it can affect the signal integrity. To re-

Figure 2.9: High level overview of 8b/10b encoding process

duce the effects of noise, PCIe employs a high-speed signalling standard: Low-Voltage Differential Signalling (LVDS). With LVDS, information is transmitted using two complementary signals: D+ and D-. The advantage of removing a ground reference improves the signal integrity. This advantage is not due to differential signalling, but to the common practice of transmitting information on a balanced line: a transmission line consisting of a pair of conductors, each of which have equal impedance along their lengths, to ground and between the transmitter and receiver [28]. In the case of PCIe links, the characteristic impedance of a PCIe link is 100 Ω .

To prevent any DC offset appearing on the differential lines, AC coupling capacitors are needed on the transmitter lines. These capacitors block DC voltage on the receiver, allowing the receiver and the driver to be biased as per PCIe specifications. This simplifies the buffer design for PCIe devices as they do not receive signals with large DC offset.

2.4.3 Configuration of PCIe Devices

The first peripherals for computers required users to set configuration switches and jumpers for each card installed. This manual process frequently resulted in conflicting memory, I/O and interrupt settings between each peripheral. A major strength of PCI and PCIe was its automated configuration mechanism, which removes the need for manual user configuration. Configuration of devices are the same between PCI and PCIe. Both standards make use of the PCI configuration space, configuration requests and PCIe device enumeration process.

PCI Configuration Space

Each PCIe device has a PCI configuration space which is in total 256 bytes long which is used in the device enumeration process. The first 64 bytes of this space is the PCI configuration header, which consists of two types: type 0 and type 1 illustrated in Table 2.2 and 2.3 respectively. A type 0 header is used for a PCIe endpoint device, whereas a type 1 device is used by a root-complex device. Both configuration header types contain useful information for an OS.

The first four bytes in both type 0 and 1 configuration spaces contains the device and vendor ID. These are useful for the operating system as it provides a label that distinguishes between multiple PCIe devices and it is used in the enumeration process. The vendor ID is allocated by the Peripheral Component Interconnect Special Interest Group (PCI-SIG) to standardised multiple PCIe device manufacturers. The device ID is then allocated by the vendor.

Both type 0 and 1 configuration spaces contain base address registers (BARs). They provide a way for the host system software to determine the size of the address space requested by the device by reading bits 0 to 11 of the BAR. In bits 12 to 32, the host writes the allocated base address assigned from an operating system. A type 0 header has six possible BARs

Bits [31-24]		Bits [23-16]		Bits [15-8]		Bits [7-0]		Reg
Device ID				Vendor ID				0x00
Status				Command				0x04
Class Code						Revision ID		0x08
BIST	Header Type		Latency Timer		Cache Line Size			0x0C
Base Address Register 0 (BAR0)								0x10
Base Address Register 1 (BAR1)								0x14
Base Address Register 2 (BAR2)								0x14
Base Address Register 3 (BAR3)								0x1C
Base Address Register 4 (BAR4)								0x20
Base Address Register 5 (BAR5)								0x24
Cardbus CIS Pointer								0x28
Subsystem ID				Subsystem Vendor ID				0x2C
Expansion ROM Base Address								0x30
Reserved						Revision ID		0x34
Reserved								0x38
Max latency		Min Grant		Interrupt		Cache Line Size		0x3C

Table 2.2: PCI Configuration Space Header for a PCI Type 0 (Non-Bridge) Device

and thus six possible address spaces which could be used to map to different registers. In a type 1 configuration space, there are only two BARs available and thus two possible address spaces.

In the PCIe architecture, it takes the PCI configuration space and extends it to include PCIe exclusive functions. This resides in the PCIe capability structure which resides inside the 256 bytes of the PCI configuration space. The contents of the capability structure is detailed in Table 2.4

After the PCI configuration space is the extended configuration space for PCIe. Including the PCI configuration space, the total size of a PCIe configuration is 4 kB. The extended configuration space provides additional functionalities in PCIe. When PCIe was first introduced, there was not enough space for a new extended capability structure. This issue was addressed by extending the PCIe configuration space range from 256 Bytes to 4 kB. Keeping the PCI configuration space provides backward compatibility with the PCI configuration space, along with implementing new capabilities in the PCIe architecture such as advanced error reporting. Since PCI devices are constricted to the 256 byte configuration space, they do not

Bits [31-24]		Bits [23-16]		Bits [15-8]		Bits [7-0]		Reg
Device ID				Vendor ID				0x00
Status				Command				0x04
Class Code						Revision ID		0x08
BIST		Header Type		Latency Timer		Cache Line Size		0x0C
Base Address Register 0 (BAR0)								0x10
Base Address Register 1 (BAR1)								0x14
Secondary Lat Timer		Subordinate Bus #		Secondary Bus #		Primary Bus #		0x18
Secondary Status				IO Limit		IO Base		0x1C
(Non-Prefetchable) Memory Limit				(Non-Prefetchable) Memory Base				0x20
Prefetchable Memory Limit				Prefetchable Memory Base				0x24
Prefetchable Memory Base Upper 32 Bits								0x28
Prefetchable Memory Limit Upper 32 Bits								0x2C
IO Limit Upper 16 Bits				IO Base Upper 16 Bits				0x30
Reserved						Revision ID		0x34
Expansion ROM Base Address)								0x38
Max latency		Min Grant		Interrupt		Cache Line Size		0x3C

Table 2.3: PCI Configuration Space Header for a PCI Type 1 Device

Bits [31-24]	Bits [23-16]	Bits [15-8]	Bits [7-0]	Byte Offset
PCI Express Capabilities Register		Next Cap Pointer	PCI Express Cap ID	0x00
Device Capabilities				0x04
Device Status		Device Control		0x08
Link Capabilities				0x1C
Link Status		Link Control		0x10

Table 2.4: PCI Express Capability Structure of Endpoint Devices

have access to the new PCIe capabilities. The overall PCIe configuration space is illustrated in Figure 2.10.

Figure 2.10: PCIe configuration address space

Device Enumeration

After post power up or system reset, the PCIe topology only consists of Bus 0 in the root complex as shown in Figure 2.11. As there are no direct method for the host to determine which slots have a PCIe device connected, software on the host must scan the PCIe fabric to discover the topology and map how the fabric is populated. The process of discovering its topology is called the enumeration process.

The enumeration process is a depth first search. The host system goes down each PCIe bus until it finds a PCIe device. From there, the host travels back and does another depth search on another PCIe bus. To enumerate a PCIe device, the device ID and vendor ID is read. If there is no device in a PCIe slot, all the signals are pulled up resulting in an invalid vendor ID of FFFFh. The system host then recognises that the system slot

Figure 2.11: Typical PCIe topology with root complex bus 0 highlighted [4]

is empty. If a valid device and vendor ID is returned, then the host reads the available BARs to determine the size of the address space requested by the endpoint device. This information is stored between bits 0 and 11 on each BAR. Once knowing the requested BAR address space size, the operating system designates a base address and allocates the requested address space within the host system for that particular BAR. The host writes the allocated base address between bit 12 and 31 of a BAR. The overall BAR structure after an address has been assigned from an OS is illustrated in Figure 2.12.

Figure 2.12: BAR structure after a base address is assigned by an operating system

2.5 PXIe Signals

Evaluation of available instrumentation standards in Section 2.2 suggests that PXIe is the most suitable platform for this thesis. The PXIe standard combines the PCIe serial transmission protocol with the compact-PCIe form factor to form a robust platform. Along with PCIe signals, the PXIe standard has its own set of signals. This section discusses PXIe signals set out by the PXI System Alliance (PXISA) [5].

2.5.1 PXI Signals

The PXIe standard is the successor of the original PXI standard, which leveraged on PCI technology. Although the PXIe standard has higher bandwidth and transfer speeds, the PXI standard is still widely used. Because of this, the PXISA includes single-ended PXI signals to allow compatibility of older PXI modules in a PXIe chassis. These signals include a 10 MHz clock (PXI_CLK10), eight trigger bus lines (PXI_TRIG[0:7]) and trigger lines from a timing module to each slots within a chassis (PXI_START).

2.5.2 PXIe Synchronisation

The PXIe standard includes these PXI signals along with additional differential signals. These added signals allow development of high-speed applications with fast synchronisation features. To implement this, the PXIe standard uses three signals: PXIe_CLK100, PXIe_SYNC100 and PXIe_DSTAR.

The PXIe_CLK100 is a 100 MHz differential clock that can be utilised by any PXIe peripheral module. Accompanied with the clock is the PXIe_SYNC100 signal which is distributed to every peripheral module slots. This signal is in phase with the PXIe_CLK100 signal which allows a PXIe peripheral module to create a 10 MHz signal which is in phase with the PXI_CLK10 discussed previously. Essentially, this 10 MHz clock is required to send and receive triggers from PXI modules which utilises the PXI_CLK10 clock.

Along with clock signals, the PXIe standard implements three types of differential triggers. These are DSTARA, DSTARB and DSTARC. High-speed, high-quality clock signals can be distributed from a PXIe timing module through DSTARA. DSTARB is designed for trigger lines that are routed from the timing module to all peripheral slots. The DSTARC trigger lines is a combination of both DSTARA and DSTARB, which is designed for both high-speed and high-quality clock or trigger signals from the timing module.

The use of these signals requires a timing module which is proposed as part of future work. A timing module can accompany the proposed System Controller and Peripheral Module for applications which require synchronisation. With these synchronisation signals, it allows users to build more sophisticated instrumentation set-ups for different applications. The overall PXIe signal topology is illustrated in Figure 2.13

2.6 Operating System

The first computer were single-purpose systems, programmed on rows of mechanical switches or jumper wires on plug boards. Developed by Manchester University, the Atlas computer used a program which managed the allocation of processing resources called the Atlas Supervisor [29]. This was considered as the first recognisable modern Operating System (OS). An OS is the low-level software that supports a computer's basic functions, such as arranging tasks for the CPU to carry out and controlling peripherals. PXIe system controllers are usually shipped with a pre-installed OS. With commercial solutions, end users have the option to choose one of two well established OS; Windows or Linux. This section explores potential OS's that can be implemented on the proposed System Controller.

Figure 2.13: Distribution of PXIe and PXI signals in a PXIe instrumentation system. Slot 1 is a system controller slot. Slots 2 and 3 are PXIe/PXI peripheral module hybrid slots. Slot 4 is a combined PXIe timing module and peripheral module slot. Slots between 5 and 8 are PXI peripheral module slots. [5]

2.6.1 Windows

Usage of the Windows OS can be found in many applications, from personal computers to embedded electronic systems. Its wide-spread popularity makes its graphical user interface (GUI) intuitive to use and provides a large eco-system of programs for different applications.

Generally, PXIe system controllers are usually shipped with a Windows OS. The choice of using Windows has the potential to cater for a wide range of users. However, users need to purchase a Windows license to obtain future security, software and hardware updates which increases the overall purchase cost for a PXIe system controller.

Being a closed-source OS, embedded developers on Windows require a Windows Driver Kit to facilitate driver development. Although Microsoft provides a well-defined framework for device drivers, the software kit requires a large amount of hard drive space. Usually PXIe system controllers are shipped with only 80 GB of hard drive space. As the driver kit can be up to 40 GB, the amount of usable space after its installation becomes more restrictive, limiting the amount of data and other programs that can be installed on the controller.

2.6.2 Linux

Linux is an OS developed by contributors world wide. The defining component of Linux is its open source kernel, which was first released on September 17, 1991 by Linus Trovaldus [30].

The open nature of the OS has allowed it to be ported to a wide variety of devices. One of the motivations of this thesis is to provide an open system where users can develop their own hardware. Unlike Windows, Linux is an open source OS which does not have any formal framework. However, developers have access to the Linux's kernel source code, permitting development of custom drivers without additional software. This in turn frees up storage for other programs and data. For this reason, im-

plementing a Linux OS instead of a Windows solution provides flexibility and less storage constraints for developers on an open PXIe system.

2.6.3 Selection of OS

To install an OS on the module requires a storage solution. Found in most commercial PXIe system controller solutions, is a SATA hard drive storage medium containing the OS and other programs. With a Windows OS solution, a SATA hard drive is necessary. The interface between the silicon and the hard drive also requires proprietary drivers [31]. In addition, a Windows license is required to run the OS which does not fit into the open PXIe system architecture. As a result, a Windows OS solution is complicated which requires further expenses by a user for a functional solution.

An alternative solution is utilising an open source OS called Linux. Its open source nature allows a wide range of customisation including a solution for different storage mediums. An alternative to SATA is an SD card storage medium. A Linux solution can be customised for a SD card which provides plug-and-play functionality unlike traditional hard drives. This is a more convenient if users want to use their own custom Linux solution. Additionally, a Linux OS solution is smaller in size compared to a Windows solution. This reduces the cost of the storage medium required for the proposed System Controller.

With Linux, an OS image can be as small as 4 MB and can be stored on a cheap SD card. This solution is more cost effective and simpler than a Windows solution. Effectively, a 4 GB SD card is more than enough to implement a Linux platform. In addition, a Linux kernel already contains drivers for USB storage devices, making it readily expandable if users need extra space. In turn, using a Linux OS for the proposed System Controller is more suitable than Windows as it supports the open source nature of this project while being a simpler and more cost effective solution.

2.7 Integrated Circuit Technology

Part of the project requires development of a PCIe engine to meet the PXIe standard. As PCIe has a high data rate, IC technologies such as micro-controllers cannot meet the speed performance required. Alternatively, another IC technology that is more promising are microprocessors. Specifically, Acorn RISC Machine (ARM) based microprocessors which have been used in commercial PCIe devices [32, 33]. Although a microprocessor solution would permit a potential embedded Linux solution and thus meet the functional requirements for the proposed System Controller, such implementation is expensive and complicated as it would require interfacing with an external PCIe IC along with DDR memory. This is a complex implementation and customisation is restricted to software development.

Another IC solution are FPGAs (Field Programmable Gate Arrays) which consist of an array of programmable logic blocks of different types as well as some special I/O blocks such as PCIe or Ethernet engines. This includes general logic, memory and multiplier blocks. These blocks are surrounded by programmable routing fabric that allows the blocks to be interconnected. Every FPGA relies on programming technology that is used to control the programmable switches in the routing fabric. A widely used programming technology in FPGAs are static memory based [34]. Static random-access memory (SRAM) are distributed throughout the FPGA to provide reconfigurability. There are two primary uses for SRAM cells, to set the select lines to multiplexers and steer interconnect signals. The remaining SRAM cells are used to store data in look-up tables (LUTs) to implement logic functions.

Implementing an FPGA solution for a PXIe platform would allow users to develop their own custom hardware and software applications. However, it follows the same implementation complexity to that of a microprocessor solution. A PXIe-FPGA module still requires DDR memory and other components to be coupled with the FPGA. However, there exist

modular solutions which removes this complexity. They can be interfaced through board-to-board connectors which is simpler than developing an all-in-one solution. Additionally, some vendors of these modular FPGA solutions are pin compatible with other FPGA families. This means if the PXIe modules utilise a modular solution that is pin compatible with a different FPGA family, it removes the restriction of using a single FPGA family per PXIe module. With the aid of modular FPGA solutions, users can customise their own hardware without the need of developing or purchasing an entirely new PXIe module.

A requirement of the System Controller is to host an OS to allow users to interface with the open PXIe platform. With Xilinx FPGAs, a MicroBlaze soft processor could be utilised to host a Linux OS [35] with PCIe connectivity. However, the performance is rather poor and the development process is tedious as it requires development of build scripts and compilation of a Linux image. An alternative solution is to utilise a Xilinx Zynq System-on-Chip device [36] which contains an ARM microprocessor processing system (PS) and FPGA programmable logic (PL). This all-in-one solution is appealing as it allows both RTL hardware development on the PL along with software development on the PS. Xilinx has also developed a single toolset that enables development of embedded Linux solutions for the Zynq SoC. The toolset contains the necessary build scripts to automate the development of a Linux OS image.

Although a Peripheral Module could utilise a Zynq device, it is not a cost-effective solution as the Peripheral Module does not need to host an OS. An alternative FPGA is required that can host a PCIe endpoint engine while providing functionality towards the system. It is also preferable to use Xilinx FPGAs which removes the need of using multiple vendors and software tools. Comprising of Xilinx's current 7-Series FPGA lines are the Virtex-7, Kintex-7 and Artix-7 families. The Artix-7 family is appropriate for the Peripheral Module due to its low cost and on-board PCIe generation 2 hard-block. Although the other 7-Series families can be utilised for

this project, they are significantly more expensive than the Artix-7.

It is discovered that two IC technology could be utilised for the two proposed PXIe modules. Firstly, a microprocessor has been used in a couple of commercial solutions that implements PCIe connectivity. The pitfall with using a microprocessor requires an external interface IC to implement PCIe connectivity. In contrast with microprocessors, there exist modular FPGA solutions that can be utilised for both boards. This has the potential to simplify the hardware design of the proposed modules. Coupled with its hardware programmability, it is decided that FPGA technology is the most suitable option for the PXIe modules. The implementation of both modules are presented in Chapter 5.

2.8 Investigation of I/O Standards

An investigation of development FPGA boards is carried out to determine I/O standards that could be utilised with the proposed Peripheral Module. These development boards offer a range of I/O standards used in FPGA applications, allowing signals to be either obtained or driven.

One standard that is primarily used in FPGA development boards is the open Pmod interface [37]. Its simple design allows application specific modules to be connected to the Pmod connector and be interfaced through an FPGA. There are a wide range of Pmod modules that can be used in different applications. However, the Pmod interface is restricted to slower applications. Another I/O standard needs to be considered for faster applications which can fully utilise the high-speed PCIe serial bus bandwidth.

Another I/O standard that is common on development boards is the FPGA Mezzanine Connector (FMC), illustrated in Figure 2.14. Developed by a consortium of companies, the FMC is an ANSI standard that provides a mezzanine card form factor, connectors and modular interface to an FPGA on a host board. There are two different connectors board de-

signers can choose from that offers different levels of connectivity: A Low Pin Count (LPC) with 160 pins and a High Pin Count (HPC) connector with 400 pins. Both connectors support single-ended and differential signalling up to 2 Gb/s and signalling to an FPGA's serial interface at up to 10 Gb/s [38]. The large bandwidth allows companies to develop FMC cards for a variety of applications, including high-speed analogue, RF, image processing and interface peripherals. The FMC standard provides a better utilisation of the PCIe serial bus bandwidth in a PXIe platform compared to the Pmod standard.

Figure 2.14: An FMC HPC male (top) and female (bottom) connector [6]

Coupled with FPGA technology discussed in the previous section, these I/O standards can be utilised by the Peripheral Module. Users can develop their own software or hardware applications that interfaces with external modules compatible with these I/O standards. This allows the Peripheral Module to be customised for different applications, removing the need for users to buy expensive specialised PCIe solutions. The implementation of the explored I/O standards are later discussed in Chapter 5.

2.9 Existing Solutions

As of 2017, there are more than 65 PXIe alliance companies [39] that produce PXIe modules and chassis for different fields including medical, telecom, automotive industry and electronic manufacturing. In addition, development on the PXIe platform is also evident in research [40, 41, 42, 43, 44]. Although the PXIe standard is open, these projects utilise commercial solutions which tend to be expensive and non-customisable. The specialised nature of these PXIe modules further increases the cost of using the PXIe platform as different modules need to be purchased for different applications.

Justified in the previous section, the proposed PXIe modules will utilise a modular FPGA solution to meet their functional requirements along with introducing versatility to the open PXIe platform. There exists a small niche of PXIe modules that utilise FPGA technology. However, these devices tend to be more expensive than specialised commercial PXIe modules, with the cheapest FPGA solution costing USD \$6000 [7]. The PXIe-7961 offers a Xilinx Virtex-5 FPGA that provides customisable I/O for the LabVIEW environment [45]. Although it offers customisation, the cost of the module is not justifiable for engineers and researchers with a lower budget. In addition, the PXIe-7961 uses an obsolete Virtex-5 FPGA. The Virtex-5 is not compatible with Xilinx's *Vivado* development environment. The Virtex-5 is only compatible with Xilinx's obsolete *ISE* development environment, which can still be downloaded from Xilinx's website. *ISE* is not compatible with newer OS, which limits the usability of the program. In addition, *Vivado* offers a range of IP-cores which is not found in *ISE*. These IP-cores can be utilised instead of writing HDL code and thus reducing FPGA development time.

While the PXIe-7961 offers programmability, the design of the module is closed meaning that users do not have access to the schematics. This is usually the case with commercial PXIe modules, as their IP is valuable

Figure 2.15: PXIe-7961 peripheral module from National Instruments [7]

to the manufacturer. The open nature of this project allows users to have access to a high-speed platform with the potential to develop their own electronic hardware. The European Organisation for Nuclear Research [46], known as CERN, has been supporting the Open Hardware Repository [47] to facilitate electronic designers to collaborate on open hardware designs.

The *Simple PXIe FMC Carrier Board*, otherwise known as SPEXI, is an example of an open source PXIe peripheral module developed by CERN which is illustrated in Figure 2.16. The SPEXI is a PXIe peripheral module that has one FMC LPC female connector and features a Spartan 6 FPGA with only a single lane, generation one PCIe link. Unlike the PXIe-7961 mentioned previously, there are open electronic schematics along with

VHDL demos and the implemented PCB. However, the pitfall of the SPEXI is that it utilises an obsolete Xilinx Spartan-6 FPGA which is incompatible with the latest FPGA development software. In addition, the SPEXI project utilises a Gennum GN4124 PCIe x4 bridge [48] which provides the Spartan-6 with PCIe connectivity. An all-in-one FPGA solution containing an internal PCIe hard block is preferred as it removes the complexity of PCB development. Although the SPEXI solution uses an obsolete FPGA and an external PCIe hard block, the open schematics are useful to verify PXIe electrical and mechanical requirements. Overall, the total cost for the SPEXI is estimated to be NZD \$1000 from inspection of the bill-of-materials.

Figure 2.16: SPEXI FMC carrier from CERN [8]

Both the PXIe-7961 and the SPEXI offers FPGA programmability towards the PXIe platform, but utilises obsolete or unsourceable components. The PXIe700 (illustrated in Figure 2.17) is another PXIe FMC carrier

module, similar to that of the SPEXI. The difference being that it utilises a Kintex-7 which is part of Xilinx's current family line of FPGAs. The Kintex [49] is a mid-range FPGA that is used in the areas of high-speed telecommunications, including 3G and 4G wireless as well as PCIe Generation 3. However, the cost of a Kintex-7 FPGA is significantly more expensive than the Artix-7, which may not appeal to users looking for a more economical solution.

Figure 2.17: PXIe700 peripheral module from Sundance Technology [9]

PXIe-FPGA solutions that are mentioned so far are all peripheral modules. As of writing, there are currently no commercial PXIe system controller that utilises FPGA technology. Implementing hardware programmability on the System Controller further increases the versatility of an open PXIe platform. Users can develop their own RTL solutions that are not found on commercial system controllers without the need of developing additional hardware. National Instruments' range of system controllers [50, 51] uses Intel Processors, coupled with an integrated hard-drive con-

taining a Windows OS. It also features multiple I/O ports such as Ethernet and USB which could be implemented on the System Controller. The least expensive system controller solution from National Instruments is the PXIe-8100 at NZD \$1795. This system controller provides an achievable cost objective.

It is discovered that the largest drawback of existing PXIe peripheral module solutions is the use of obsolete components. The PXIe-7961 and SPEXI utilises a Kintex-5 and Spartan-6 FPGA respectively. A PXIe-FPGA solution that utilises a 7-Series device is the PXIe700. It features a Kintex-7 FPGA device, which is significantly more expensive than the previously mentioned Artix-7 device. Although each module has their drawback, their design implementation can be utilised for the Peripheral Module. There are no commercial system controllers that utilise an FPGA solution. However, the cost of the PXIe-8100 provides an achievable cost objective for the System Controller. With a detailed review of existing solutions, the objectives for both System Controller and Peripheral Module are laid out in Chapter 3 and 5.

2.10 Previous PXIe Modules for MRI Instrumentation Development

The School of Engineering at Victoria University of Wellington has developed three PXIe modules for MRI instrumentation development. These module designs can be utilised to develop solutions with latest market FPGA solutions, as well as lower cost components. This section reviews the previously developed PXIe modules.

2.10.1 RF Transceiver Board

The first custom PXIe module solution is a RF transceiver board, illustrated in Figure 2.18. It consists of a Spartan 6 with a PCIe generation 1

x1 link and an interface with high-speed ADC and DAC devices. The solution was configured with PCIe functionality and Direct Memory Access (DMA) logic to fetch and store data on the DDR3 memory. Although the implementation was successful, it utilises an obsolete Spartan 6 FPGA that cannot be used in Xilinx's latest FPGA development software. However, the implementation of an ADC/DAC could be utilised on the Peripheral Module. Its functionality is appealing for MRI applications, but its implementation is not versatile for different applications.

Figure 2.18: RF Transceiver Peripheral Module [10]

2.10.2 PXIe System Controller with Zynq 7045 SoM

Previously developed by Matthew Bourne from School of Engineering Victoria University, is a system controller implementation, illustrated in Figure 2.19. The implementation took on a modular approach and utilises a Zynq 7045 SoC MMP module from Avnet. By utilising a modular solution, the complexity of the system controller was significantly reduced and implementation of a Linux OS was simple using Xilinx's embedded solu-

tion toolset. On the fabric of the Zynq SoC is a PCIe root-port model that allows the memory reads and writes from the Linux OS to be sent to other peripheral modules. Although the solution is elegant, the implementation was costly. Firstly, the system controller utilises a mid-range SoC package which contributed a significant portion towards the overall development cost. A smaller Zynq SoC package can be used in the 7045 place which provides the same functionality but at lower cost. Secondly, the system controller utilises an expensive power module to power various components of the system controller. However, Xilinx's has released a power estimation tool which can be utilised to develop robust and economical power circuitry for the Peripheral Module.

Figure 2.19: Previously implemented PXIe system controller with Zynq 7045 [10]

2.10.3 PicoZed System Controller Implementation

Further work has been undertaken at the School of Engineering and a new system controller had been designed (illustrated in Figure 2.20) that builds on from the solution that utilises a Zynq 7045 SoC device. In its place, a Pi-

coZed modular solution is utilised which consist of a Zynq 7015 SoC. The smaller package is significantly more economical than the 7045 solution implemented on the previous system controller. In addition, the expensive power module is removed and replaced with discrete power components. However, the PicoZed based design was not physically implemented and still needed peer review before fabrication. Also, further verification of the hardware design is required to establish the requirements of the OS, driver design and the Peripheral Module hardware. The current implementation of the PCIe root port design and the PCIe driver design used in the 7045 are limited. The PCIe root-port only supports PCIe x1 link, while the driver is only limited to interfacing with one device. After verification of the hardware design, fabrication and firmware design of the PicoZed System Controller can commence.

Figure 2.20: PicoZed System Controller 3D rendering of designed board

Chapter 3

System Controller Hardware Design Review

Discussed in the previous chapter, a new System Controller had been designed by the School of Engineering. However, it has not been manufactured. In this chapter, the design of the System Controller is explored and validated before manufacture. The focus of reviewing the hardware design of the System Controller also establishes the requirements for OS, driver design and the Peripheral Module hardware design later in development. In turn, the developed System Controller is a one of two core modules required for an open PXIe system.

3.1 System Controller Mechanical Construction

The System Controller needs to meet PXIe mechanical specifications for it to be compatible with the PXIe-1062Q chassis. This section details the mechanical construction of the implemented System Controller.

3.1.1 Connectors

A connection is made between the PXIe-1062Q chassis and a module using a CompactPCIe connector [52]. There are three different types of backplane connectors that are utilised within the PXIe standard:

- Advanced Differential Fabric (ADF) Connector - A 90 pin connector used for transmitting PCIe signals through differential lanes. In peripheral slots, it provides PXIe signals such as DSTAR and a 100 MHz differential clock. Both of which can be used for synchronisation applications.
- Enriched Hard-Metric (EHM) Connector - A 60 pin connector which carries power (in peripheral slots) and PXI signals which will not be used in this thesis
- Universal Power (UPM) Connector - A 7 pin connector that supplies power in a system controller slot

Connectors on each modules are female connectors, while the 1062Q chassis' backplane utilises male connectors.

Many vendors provide the same connectors used in the PXIe standard. Most of which do not state if it is compatible with PXIe. It is important to choose verified PXIe connectors as unverified connectors may result in incompatibility issues. The two modules utilise connectors from ERNI Electronics [53], which are ensured to be compatible with the PXIe standard.

For the System Controller to be compatible with the 1062Q chassis, it needs to meet the PXIe system controller connector requirements. The System Controller module requires one EHM (XJ4), two ADF (XJ2 and XJ3) and one UPM (XP1) right angle female connectors which are illustrated in Figure 3.1.

As a System Controller, it needs to forward four PCIe x4 links and 100 MHz clocks to each of the four peripheral and timing module slots on the chassis. These PCIe signals are located on connectors XJ2 and XJ3. Power

Figure 3.1: PXIe System Controller backplane connectors [5]

rails are supplied to the System Controller through XP1. The physical implementation of the System Controller backplane connectors are illustrated in Figure 3.2 which conforms to the PXIe requirements for a system controller. Additionally, the pin outs of these connectors are verified with PXIe hardware specifications [5] to ensure that proper connections are being made between the System Controller and the backplane.

3.1.2 PCB Dimension

As per PXIe mechanical specifications [5], the System Controller needs to meet the 3U Eurocard form factor with a dimension requirement of 100 mm by 160 mm. For a module to fit within a PXIe slot, the thickness is required to be 1.6 ± 0.2 mm. Reviewing of the PCB dimensions concluded that it meets the dimension requirements as illustrated in Figure 3.3.

The PCB manufacturer provides a standard 1.6 mm, eight signal layer board illustrated in Figure 3.4. This information of the System Controller's layer stack verifies that the thickness of the board is within PXIe tolerance.

Figure 3.2: System Controller backplane connectors implementation

The layer stack is later used for the Peripheral Module as both designs utilise the same characteristic impedance requirement. This is discussed later in Chapter 5.

3.2 Verification of the PicoZed Interface

The design of the System Controller utilises a PicoZed module from Avnet (illustrated in Figure 3.5), consisting of a Zynq 7015 SoC. In this section, verification of the physical interface and the pinout are discussed.

3.2.1 Verification of connector positioning

Provided in the PicoZed 7015 hardware guide [12] is a detailed mechanical footprint for the module. This is illustrated in Figure 3.6, which consists of

Figure 3.3: System Controller hardware implementation with width and height measured on Altium

Figure 3.4: System Controller layer stack from PCB manufacturer

Figure 3.5: Avnet's PicoZed 7015 Module [11]

three 100 pin board-to-board connectors. The implemented footprint on the System Controller is cross validated with the dimension reference to ensure that the alignment of these connectors are correct. It is concluded that these connectors are in the correct positions and thus, no additional changes were made.

3.2.2 Verification of PicoZed Pinout

The pinout of the implemented PicoZed footprint needs to be verified to ensure correct functionality. The pinout of the PicoZed consists of mostly I/O, which some are connected to peripherals on the System Controller. These connections are later verified later in Section 3.4. However, the PicoZed requires various voltage input. Firstly, the PicoZed requires a 5 V voltage source across four pins of the connector. Additionally, FPGA bank 34 and 35 require an external voltage source for I/O applications. The design is later discussed in Section 3.4 The implemented design provides a voltage adjustable design that allows users to select different voltages for different I/O standards. The schematic and routing of the implemented System Controller are cross validated with the PicoZed footprint to en-

Figure 3.6: PicoZed 7015 footprint from Avnet [12]

sure that the 5 V and adjustable voltage source are connected to the correct voltage input pins. If these connections were incorrect, it may lead to the destruction of a PicoZed module.

Secondly, the PicoZed has 8 GTP transceivers which is used to implement PCIe connectivity. The connection between the PCIe x4 link and the PCIe bridge (later discussed in Section 3.3) is verified to ensure that the proper communication link exist between the PicoZed and PCIe bridge. Additionally, these transceivers requires a 1.0 V and 1.2 V source which are implemented correctly in the System Controller schematic.

Figure 3.7: System Controller PCIe connectivity schematic

3.3 PCIe Connectivity of the System Controller

As mentioned in the previous Section, the PicoZed contains four TX and four RX GTP transceivers that are used to implement PCIe connectivity. The architecture of the transceivers allows both modules to establish a PCIe x4 generation 2 link. However, a requirement for the System Controller is it must communicate with four PXIe banks. Accomplishing this requires the PCIe x4 link from the PicoZed GTP transceivers to fan out to four separate PCIe x4 links.

The PEX8624 PCIe Generation 2, 24-lane, 6-port Switch [54] is utilised in the System Controller to fulfil this fan out requirement. The PEX8624 is configured through strapping pins to set the input port and four output ports to PCIe x4 links. The output ports and their accompanied reference clock are then routed to their respective backplane connectors as shown in Figure 3.8.

3.3.1 System Controller Clock System

The System Controller is required to source all PXIe slots with a 100 MHz PCIe reference clock. The Peripheral Module's GTP transceivers utilise this reference clock and multiplies it up to 2.5 GHz using a phase-locked-loop (PLL) residing within the architecture. To meet PCIe generation 2 specifications, the reference clock is required to be 100 MHz $\pm$ 300 ppm. This is realised by utilising a DSC557-03 PCIe Generation 2 clock generator by Micrel [55], which outputs two 100 MHz reference clocks with a

An LMK00308 is utilised to buffer five separate clocks through banks A and B. Bank A has one output which is routed to the PEX8624 reference clock input, while bank B has four reference clock outputs for each PXIe slot within the chassis. Bank A is configured to High Current Steering Logic (HCSL) standard required by the PEX8624 design guide, whereas bank B is configured to LVPECL to comply with the PXIe standard [52]. The overall System Controller's clock system is illustrated in Figure 3.9.

3.4 Implemented Peripherals

The designed System Controller provides various on-board peripherals which can be utilised by a user for different applications. Peripherals consist of a microSD slot, UART, USB OTG and Ethernet connectivity. This section discusses the usage of these peripherals in an open PXIe platform.

Figure 3.9: System Controller Clock System

3.4.1 microSD

A microSD card form factor introduces non-volatile storage in the System Controller. Implementation of a microSD card begins with a female microSD socket, as to store images on a microSD card, the card must be removable from the System Controller. The socket is mounted at the front of the System Controller PCB, allowing the card to be inserted and removed from the chassis with ease. The microSD card location is illustrated in Figure 3.10.

The connection between the microSD card socket and the PicoZed are SPI 3.3 V data lines. They are connected to MIO pins 40 through 46 as the Zynq SoC specified these pins to be the dedicated microSD interface [56]. Connecting the SPI data lines also facilitates boot from a microSD card which is handled by the Zynq's internal boot manager. This is a convenient feature as it allows a Linux kernel or an FPGA image to be loaded during the Zynq SoC start up. The dedicated microSD MIO pins are configured to be a 1.8 V interface. The 3.3 V interface is translated to 1.8 V by utilising a Texas Instruments' TXS02612RTWR voltage level translator IC illustrated in 3.10. The overall schematic of the microSD card system is

Figure 3.10: microSD card interface

illustrated in Figure 3.11.

Figure 3.11: microSD card schematic

3.4.2 UART

The Universal Asynchronous Receiver Transmitter (UART) is implemented on the System Controller through a micro-USB female interface. The later developed Linux OS can utilise this interface to allow users to communicate with the System Controller through a terminal running on an external computer. From the female connector, a differential data line is routed to a USB-to-UART bridge conversion bridge with 100 Ω characteristic impedance. From the bridge, the RX and TX UART data lines are routed to a dedicated Zynq UART MIO pin.

3.4.3 USB OTG

Previously discussed in Chapter 2, the PXIe-8100 contains different I/O standards including USB connectivity. This allows users to use flash storage media and other peripherals such as a mouse or keyboard. Implemented on the System Controller is a USB On-The-Go (OTG) interface which allows the same connectivity found on commercial solutions. A micro-USB female interface is utilised rather than a type A USB due to space constraints at the front of the System Controller. The differential data line from the USB are routed to a dedicated USB OTG pin on the Zynq 7015 package.

3.4.4 Ethernet

An Ethernet connection can be utilised to download Linux images straight to the PicoZed without the need to boot up from an SD card. This can be utilised later during the development of a Linux OS.

Implementing Ethernet connectivity requires three components, an Ethernet PHY to implement OSI physical layer functions, magnetics for signal conditioning and a RJ45 physical connector. A high-level overview is illustrated below in Figure 3.12.

Figure 3.12: PicoZed's Ethernet implementation

The PicoZed module has an on-board Marvell 88E1512 PHY [57] while the magnetics and connector are in a single, 10/100/1000 Mbps RJ45 connector package. From the PicoZed schematics, the Ethernet PHY connections are broken out to the JX3 connector of the PicoZed as shown in Figure 3.13.

PETH_PHY_LED0	JX3-47	VCC0_15	ETH_PHY_LED0	VCC0_15	JX3-48	PETH_PHY_LED1
	JX3-49	GND	GND	GND	JX3-50	
PETH_MD1_P	JX3-51	ETH_MD1_P	ETH_MD2_P	ETH_MD2_P	JX3-52	PETH_MD2_P
PETH_MD1_N	JX3-53	ETH_MD1_N	ETH_MD2_N	ETH_MD2_N	JX3-54	PETH_MD2_N
	JX3-55	GND	GND	GND	JX3-56	
PETH_MD3_P	JX3-57	ETH_MD3_P	ETH_MD4_P	ETH_MD4_P	JX3-58	PETH_MD4_P
PETH_MD3_N	JX3-59	ETH_MD3_N	ETH_MD4_N	ETH_MD4_N	JX3-60	PETH_MD4_N

Figure 3.13: Ethernet data connections on PicoZed

The integrated magnetics package has four differential data lanes that interfaces back to the PicoZed's onboard Ethernet PHY. These data lanes are routed with 100 Ω characteristic impedance as per Texas Instruments' technical note on PCB layouts for Ethernet interfaces [58]. Connections of the integrated magnetics package to the PHY are shown in Figure 3.14.

3.5 System Controller Power System

To ensure that the System Controller is functional, the power circuitry is verified to ensure that system utilises the correct voltages. This is accom-

Figure 3.14: RJ-45 connector with integrated magnetics connections

panied with an estimation of the PicoZed power requirements during operation. In this section, the power system of the System Controller is explored.

3.5.1 PicoZed Power Requirements

Illustrated in Figure 3.15 is the System Controller power architecture. It is required to power the PicoZed and various on-board peripherals. The PicoZed is powered by a 5 V rail which is obtained by stepping down the chassis' 12 V rail. *MGTVCC* and *MGTVTT* are two separate power supplies for the PicoZed's GTP transceivers. For the transceivers to operate properly, *MGTVCC* and *MGTVTT* requires a 1.0V and 1.2 V source

respectively. Discussed later in Section 5.9.3, these power sources are obtained in that specific sequence by using a sequenced power supply. The PicoZed has two on board I/O banks which need to be supplied externally. Banks 34 and 35 implements the I/O standard LVDS25 which requires a 2.5 V source. Lastly, on-board components require either a 1.8 V and 3.3 V source.

Figure 3.15: System Controller power architecture

3.5.2 PicoZed Power Estimation

Xilinx has developed a tool called Xilinx Power Estimator (XPE) [59] that allows users of Xilinx's products to estimate the power consumption in different scenarios. With XPE, both modules are given estimation constraints that are beyond their normal operational capacity. The output from the XPE tool is an accurate worst-case power analysis, which provides useful information verify the power system on the System Controller.

The first step in using the XPE tool is to give the System Controller operating parameters. In this case, the XPE tool is given an unlikely scenario of 100% design utilisation on the PicoZed's Zynq 7015. This provides the

resulting estimation of the PicoZed's power draw during upper limits of operation. Knowing the upper limit requirements of the PicoZed, a robust power circuitry can be developed which minimises risks of the System Controller being under powered. In addition, XPE provides parameters for design activity and physical interfaces. The design activity is kept to default, while the physical interfaces are configured. These physical interfaces includes on-board DDR3 memory, PCIe generation 2 x4 and LVDS I/O signals. In addition all 50 available I/O on the PicoZed are used for estimation. Half of these available I/O pins are configured to writes, while the other half are configured to reads. The PicoZed's power estimation parameters are shown in Figure 3.16

XPE Quick Estimate - XC7Z015CLG485-1I

XC7Z015CLG485-1I

Conditions: Typical, Ambient=25C

Environment: 250 LFM

Voltage: ☒ Nominal ☐ Maximum

Design Activity

	Clock	Toggle	Enable
Logic	150 MHz	12.5 %	50 %
BRAM	150 MHz	50 %	25 %

Design Utilization

	Value	%
LUT	46200	100.0
FF	92400	100.0
BRAM	190	100.0
DSP	160	100.0

Physical Interfaces

	Number	Width	Rate
Memory: DDR3	1	36	800 Mb/s
GTP: PCIe Gen2	1	4	5 Gb/s
GTP:			Gb/s
LVDS: In 25 Out 25			667 Mb/s

OK Cancel

Figure 3.16: XPE settings for PicoZed power estimation

The XPE tool reported a total on-chip power of 4.13 W. Discussed in the previous section, the PicoZed is powered by a 5 V source derived from the chassis' backplane 12 V rail. A MAX15066 buck converter was chosen to step down the 12 V rail to 5 V as it can provide 4 A of continuous current

at 5 V. It therefore has power rating of $5 \times 4 = 20W$ which is around 5 times higher than XPE's worst-case total on-chip power.

The XPE tool also provides a power supply estimation for each of the voltage rails as shown in Figure 3.17. Key voltage sources are $V_{CCO}2.5V$, $MGTAV_{CC}$ and $MGTAV_{TT}$, which supply power to I/O banks and GTP transceivers respectively. This information infers that the System Controller needs to supply three voltage rails, 2.5 V, 1.0 V and 1.2 V, with worst-case continuous current at 0.378 A, 0.273 A and 0.241 A respectively. In addition, the PEX8624 PCIe switch power draw needs to be considered as it has a significant power requirement of 1.9 W at 2.5 V.

Power Supply		
Source	Voltage	Total (A)
V_{CCINT}	1.000	1.311
V_{CCBRAM}	1.000	0.016
V_{CCAUX}	1.800	0.442
V_{CCAUX_IO}		
$V_{CCO} 3.3V$	3.300	
$V_{CCO} 2.5V$	2.500	0.378
$V_{CCO} 1.8V$	1.800	
$V_{CCO} 1.5V$	1.500	0.247
$V_{CCO} 1.35V$	1.350	
$V_{CCO} 1.2V$	1.200	
-		
$MGTAV_{CC}$	1.000	0.273
$MGTAV_{TT}$	1.200	0.241
V_{CCPINT}	1.000	0.068
V_{CCPAUX}	1.800	0.010
V_{CCPLL}	1.800	0.003
V_{CCO_DDR}	1.500	
V_{CCO_MIO0}	1.800	
V_{CCO_MIO1}	1.800	
V_{CCADC}	1.800	0.020

Figure 3.17: PicoZed power supply estimation

From the outcome of the XPE tool, a power supply needs to supply a minimum power of 2.85 W, 0.27 W and 0.29 W for the 2.5 V, 1.0 V and

1.8 V rails respectively. Power sequencing must also be considered and is explained in detail later in Section 5.9.3. To meet power demands from these voltage banks while providing an integrated solution for power sequencing, an ADP5052 [60] is previously chosen to meet these two requirements. The package has 5 channels which can be sequenced in any order. Each channel can provide up to 4 A, which therefore exceeds the power requirements for these voltage rails.

3.5.3 PicoZed Power Sequencing

Power sequencing is an important aspect that needs to be taken account. Following recommended power sequencing from the module manufacturers, excessive current draw during start up can be avoided, which in turn prevents damage to devices [61].

From the PicoZed hardware manual[12], the recommended power sequencing is the following:

1. 1V
2. 1.2V and 1.8V
3. V_ADJ and 3.3V_reg

As mentioned in the previous section, the correct power sequencing is achieved using an ADP5052 five channel, quad buck regulator to provide an all-in-one sequencing solution. Power sequencing is achieved by using the previous voltage rail in the sequence as the enable signal. To enable the 1 V source, an on-board signal is utilised after a successful start-up of the Zynq SoC.

3.6 System Controller Fabrication Process

Electronic Manufacturing Contractors (EMC) Ltd in Lower Hutt is contracted to fabricate the System Controller PCB as well as populating the

module. The handover of the System Controller consists of sending an Altium PCBdoc. During the fabrication process, a couple of design choices are made.

The first issue with the manufacturing process is that a couple of IC pads are too narrow, as illustrated in Figure 3.18. As a result a soldermask dam could not be applied between IC pads to prevent accidental solder bridges. These areas without soldermask dams were analysed to ensure there are no solder bridges.

Figure 3.18: Affected narrow pads on System Controller hardware design

The second issue relates to the PEX8624 PCIe bridge utilised to fan out the PCIe link from the System Controller. During development, Broadcomm scheduled the IC for obsolescence [62] and as a result, was no longer in stock at major electronic vendors. However, the obsolescence was driven by change in environmental laws in many countries and is revised part was in the pipeline but not presently available. As a result, the overall functionality is not affected. EMC was able to source the older PEX8624 part from smaller electronic vendors, so no major delays were encountered.

3.7 Summary

In this chapter, the hardware design of the implemented System Controller is discussed. Verification of the design consists of checking its mechanical constraints, PicoZed connectivity and pinout, PCIe implementation and an evaluation of the power system. All aspects of the System Controller hardware design meets the PXIe standard which can be utilised later to develop the associated Linux OS and drivers. The System Controller was manufactured and the physical implementation is illustrated in Figure 3.19. There are no solder bridges on affected pads with no soldermask dams.

Figure 3.19: Built System Controller module

Chapter 4

Prototyping and Proof of Concept

Before hardware development of the modules, a prototype system was developed to verify PCIe logic and pin constraints for later hardware and firmware design. The goal of the prototype set-up is to verify PCIe connectivity and data transmission with existing hardware. As the PXIe standard uses a PCIe serial bus for communication between modules, two Xilinx development boards with PCIe capabilities were used to closely emulate a PXIe system. This chapter details the components used in this prototyping set-up, an overview of their respective FPGA PCIe design and a verification of the design.

4.1 Components

Two Xilinx FPGA development boards are utilised to closely emulate a PXIe system. One of which represents a system controller, while the other represents the peripheral module within a PXIe system.

Available development boards on-hand are the Avnet Zynq-7000 Mini-ITX [13] and PicoZed 7015 with a carrier card [14], which both utilise a Zynq SoC device. These two boards represent the System Controller and Peripheral Module respectively. By utilising these two development boards, it forms an identical PCIe topology found in the PXIe standard.

Illustrated in Figure 4.1 and 4.2 are the development boards used for the prototyping set-up.

Figure 4.1: Zynq Mini-ITX development board from Avnet, emulating the PCIe system controller [13]

4.2 Test Set-up

Illustrated in Figure 4.3 is the prototyping set-up. The Mini-ITX is installed into a chassis along with its dedicated ATX standard power supply. A 24-pin connector is used to supply power to the Mini-ITX development board. The PicoZed with its carrier card is not able to fit within the chassis. Therefore, a PCIe male-to-female extender is used which allows the Mini-ITX to interface with the PicoZed external to the chassis.

Figure 4.2: Zynq PicoZed with carrier card from Avnet, emulating the PXIe peripheral module [14]

Accompanied with each development board are their respective FPGA design. Application note XAPP1171 from Xilinx [63] is utilised to develop a simple PCIe root-port and endpoint on each of the respective FPGA development boards. Both FPGA firmwares are developed and exported into a bitstream file from Xilinx's Vivado Design Suite. These bitstreams can be stored on a SD card and loaded onto a Zynq SoC FPGA fabric.

A goal of the open PXIe system is to provide an interface through

Figure 4.3: Zynq Mini-ITX development board in chassis (1), PCIe extender (2) and PicoZed with Carrier Card (3) in prototype set-up.

an Linux OS. A simple Linux image which contains PCIe command line tools and the PCIe root-port FPGA firmware is developed for the Mini-ITX using the Xilinx’s Petalinux toolset. Details of the Linux OS and FPGA firmware are discussed later in Chapter 6.

With the prototype configured, the PCIe connectivity for both systems are tested in the following section. This consists of using commands on the Linux command line to verify if a FPGA PCIe-endpoint can be seen on a Linux OS.

4.3 PCIe Connectivity Testing

A Linux image with the necessary PCIe tools is generated using Petalinux and stored onto a microSD card. The Mini-ITX development board is configured to boot the Linux image from microSD card. To communicate with the Mini-ITX, a UART connection is established with an external computer

running a serial console. Concurrently, the PCIe endpoint image is stored on another microSD card and loaded onto the PicoZed.

After the boot-up process, a PCIe endpoint can be seen from a *lspci* output illustrated in Figure 4.4. The entry contains the correct configuration for the FPGA endpoint design. This includes the selected type of device, "Memory controller", and the device ID, "7011". In addition, it shows the entries of the PCIe configuration registers which indicates the capabilities of the PCIe endpoint.

```
01:00.0 Memory controller: Xilinx Corporation Device 7011
Subsystem: Xilinx Corporation Device 0007
Flags: bus master, fast devsel, latency 0, IRQ 91
Memory at 60000000 (32-bit, non-prefetchable) [size=2K]
Capabilities: [40] Power Management version 3
Capabilities: [48] MSI: Enable- Count=1/1 Maskable- 64bit+
Capabilities: [60] Express Endpoint, MSI 00
Capabilities: [100] Device Serial Number 00-00-00-01-01-00-0a-35
Kernel driver in use: xpcie
Kernel modules: xpcie
```

Figure 4.4: PicoZed endpoint *lspci* output from Linux command line

4.4 Summary

By developing a prototype PCIe system, the PCIe logic and pin constraints can be verified. In addition, the development of the FPGA firmwares and Linux image demonstrates the design flow in Vivado and Petalinux. These are later used to develop the overall firmware for both modules. The verification of PCIe connectivity between the development boards utilises the command line interface of the Linux OS. Using the *lspci* command, the PCIe endpoint can be seen along with its correct PCIe endpoint configurations. With a successful test on the prototype system, the development of the Peripheral modules' hardware, software and firmware can begin.

Chapter 5

Peripheral Module Hardware Design

5.1 Overview

Justified in earlier, the PXIe standard is chosen to be the instrumentation platform for the modules. This chapter discusses the hardware objectives for the Peripheral Module and its respective PCB design.

5.2 Open PXIe System Hardware Objectives

A typical PXIe system consists of four main components: chassis, system controller, timing module, and peripheral module. As this thesis focuses on forming the foundation of an open PXIe system, illustrated in Figure 5.1, the two core modules will be a System Controller and a Peripheral Module. The modules will take advantage of FPGA's high speed and programmable nature, allowing the platform to be customised for a range of applications.

Discussed in the previous chapter, various PXIe modules were previously developed by the School of Engineering at VUW. The design of Pe-

Figure 5.1: Proposed open PXIe hardware architecture

Peripheral Module can be leveraged for this thesis. The Peripheral Module will utilise a 7-Series device from Xilinx, rather than a Spartan-6 found on the RF transceiver peripheral module. Although the I/O functionality of the previous peripheral module is appealing, the implementation is limited to specific applications. A new approach is taken to allow the Peripheral Module to be versatile.

The role of the Peripheral Module is to provide functionality to the open PXIe platform. To achieve this, the design of the Peripheral Module needs to accommodate various I/O standards such as FMC and Pmod, which are explored in the previous chapter. By utilising these standards, users can customise the Peripheral Module for different applications by attaching FMC and Pmod cards that are available. These cards can be interfaced by developing RTL logic on the FPGA. Along with functionality, the FPGA on-board the Peripheral Module will need to interface with the backplane's PCIe link which originates from the System Controller. This allows the System Controller to communicate the Peripheral Module through PCIe.

Although the chassis forms the overall structure of a PXIe system, a

new open chassis is not a focus of this thesis and can be implemented as part of future work. Instead, an objective for both modules is to be compatible with a PXIe-1062Q chassis by National Instruments. The 1062Q chassis is sourced as it meets the PXIe standard and provides the necessary slots for the proposed System Controller and Peripheral Module.

A timing module is used to synchronise different PXIe modules using clock signals and trigger lines through an external interface or through DSTAR lines running along chassis backplane. The synchronisation utility of a timing module is an important feature of a PXIe platform. However, it is not necessary for a basic PXIe instrumentation system and thus, the timing module is not a focus of this thesis. As a timing module can be implemented as part of future work, the Peripheral Module needs to be able to source the future synchronisation clocks and trigger lines.

Along with hardware implementation, the both modules need to be sufficiently in-expensive to appeal to a wide range of potential users. To be considered cost effective, each module needs to be less expensive than their existing solution counterpart. For the System Controller, the PXIe-8100 from National Instruments is the most comparable commercial system controller solution, with a price tag of NZD \$1795. Whereas the SPEXI with a production cost of about NZD \$1000, it is the most comparable solution for the Peripheral Module.

In summary, the hardware and cost objectives for the Peripheral Module are:

- A PXIe-FPGA general purpose Peripheral Module that is compatible with a PXIe-1062Q chassis and implements PCIe, FMC and Pmod connectivity
- More economical to produce in comparison to their existing solution counterpart

The overall architecture of each module that fulfils these objective are depicted in Figure 5.2.

Figure 5.2: Peripheral Module high-level architecture

5.3 Peripheral Module Mechanical Implementation

Identical to the System Controller mechanical constraints, the Peripheral Module needs to meet the 3U Eurocard form factor. The form factor has a dimension requirement of 100 mm by 160 mm with a thickness of 1.6 ± 0.2 mm. Additionally, the Peripheral Module needs to implement two female connectors to interface with the backplane of a PXIe chassis. This section discusses the connectors implemented on the Peripheral Module and the development of a skeleton PCB.

5.3.1 Connectors

A connection is made between the PXIe-1062Q chassis and a module using a CompactPCIe connector [52]. There are three different types of backplane connectors that are utilised within the PXIe standard:

- Advanced Differential Fabric (ADF) Connector - A 90 pin connector used for transmitting PCIe signals through differential lanes. In peripheral slots, it provides PXIe signals such as DSTAR and a 100

MHz differential clock. Both of which can be used for synchronisation applications.

- Enriched Hard-Metric (EHM) Connector - A 60 pin connector which carries power (in peripheral slots) and PXI signals which will not be used in this thesis
- Universal Power (UPM) Connector - A 7 pin connector that supplies power in a system controller slot

Connectors on the Peripheral Module are female connectors, while the 1062Q chassis' backplane utilises male connectors.

Illustrated in Figure 5.3, the Peripheral Module requires two right angle female connectors: one EHM (XJ4) and one ADF (XJ3).

Figure 5.3: PXIe Peripheral Module backplane connectors [5]

Power rails are supplied to the Peripheral Module through the XJ4 connector. The PCIe link is supplied to the Peripheral Module through the XJ3

connector along with PXIe signals including DSTAR lines, and synchronisation clocks.

The Peripheral Module utilises the same ERNI Electronics backplane connectors found on the System Controller as discussed in Section 3.1.1. This ensures that the connectors are compatible with the PXIe standard.

5.3.2 Peripheral Module Skeleton PCB

With the backplane connector requirements verified for the Peripheral Module, a skeleton PCB template is provided in Altium PCB Design Software [64] that meet the chassis' 3U Eurocard form factor requirement for a Peripheral Module. The skeleton board is used to verify the positions of the connectors and the dimension of the PCB to ensure it meets PXIe mechanical standards. Figure 5.4 illustrates the final skeleton board design for PXIe Peripheral Module.

Figure 5.4: PXIe Peripheral Module skeleton PCB

The skeleton Peripheral Module PCB provide the mechanical foundation for electronic circuitry to be implemented.

5.4 Interfacing with a Modular FPGA Solution

Discussed earlier, utilising modular FPGA solutions is more feasible than building an PXIe module with an embedded FPGA. The Peripheral Module act as a carrier card that mates with its FPGA solution through board connectors. This arrangement is more simplistic than an all-in-one solution and allows the carrier card to provide power along with interfacing signals to and from the FPGA solution. By utilising modular FPGA solutions, users will be able to customise the hardware and RTL logic on the Peripheral Module.

5.4.1 Peripheral Module FPGA Selection

Unlike the System Controller, the Peripheral Module does not need to host an OS. Instead, it only needs to implement a PCIe interface. An FPGA solution is suitable as it allows users to utilise the remaining resources left over after implementing core functionality for custom RTL development. The existing counterpart of the Peripheral Module is the SPEXI module from CERN. It utilises an obsolete Xilinx Spartan-6 FPGA which is not compatible with the latest Vivado development environment. A 7-Series FPGA is therefore required by the Peripheral Module to take advantage of utilising Vivado's IP blocks. The Artix-7 100T package is chosen as it is the least expensive package with PCIe hard-block residing within the FPGA fabric.

The TE0712 modular FPGA solution from Trenz Electronics (illustrated in Figure 5.5) is chosen as it housed the Artix-7 100T with appropriate GTP transceivers for PCIe. Additionally, the TE0712 is part of Trenz Electronics small form factor 4x5 cm modular solutions. The 4x5 cm solution line

consists not only the Artix-7 100T, but also larger Artix, Kintex and Zynq FPGA/SoC. All of these 4x5 modules are pin compatible with one another. By designing the Peripheral Module around the TE0712, users can utilise any 4x5 solution from Trenz Electronics, bringing versatility to the module and open PXIe platform. Due to pin constraints of the on-board connectors, the TE0712 does not break out all its I/O lines and thus reduces the number of usable I/O. This inherent disadvantage of the module however, does not obstruct the fulfilment of the Peripheral Module's objectives.

Figure 5.5: Trenz Electronics' TE0712 Module [15]

5.4.2 Peripheral Module Board-to-Board Connectors

The TE0712 differs from the PicoZed and many other modules as it has two 100-pin and one 60-pin high-speed hermaphroditic B2B connectors as illustrated in Figure 5.6. Unlike traditional male to female connector mating, hermaphroditic connectors can be mated with the same hermaphroditic connectors. However, this needs to be taken into account in the hardware schematic of the Peripheral Module. The pin numbering of the hermaphroditic connector footprint is mirrored to ensure right connections are being made to the TE0712 from the Peripheral Module.

With a modular solution chosen for the Peripheral Module, it simpli-

Figure 5.6: TE0712 hermaphroditic board-to-board connectors (highlighted red)

fies the overall hardware design. The Peripheral Module utilises a TE0712 from Trenz Electronics. The next step is to place the connectors in a suitable location on the Peripheral Module to prevent obstruction while optimising routing for high-speed signals. This is discussed in the following section.

5.5 High Speed Design

The Peripheral Module will utilise high-speed signals that require extra precautions during signal routing. Reflections on signal lines arise due to impedance mismatches leading to signal attenuation, distortion and ringing. All of which reduces the overall signal integrity. For such signals, meeting characteristic impedance along with implementing high-speed routing guidelines are required. This section details placement of the FPGA modular solution, building of the layer stack meeting characteristic impedance and implementation of routing guidelines.

5.5.1 PCB Layout

Before routing high-speed signals on the Peripheral Module, the location of the TE0712 and FMC connector is finalised. These components have

critical signals that need to be implemented first to minimise track distance. The module's high-speed signals route length are restricted to 6 inches as recommended by Texas Instruments in their routing application guide [16].

The critical components of the Peripheral Module are the FMC connector and the TE0712 module. The FMC connector has up to 34 LVDS high-speed signals and a gigabit clock that is interfaced with the TE0712 module. Additionally, the TE0712 module has a PCIe x4 link that needs to be established between the module and backplane. These signals make up all the high-speed signals on the Peripheral Module. The layout arrangement for the Peripheral Module is shown in Figure 5.7. This arrangement ensures that connections between the FMC, TE0712 and backplane are minimised. In addition, the FMC connector is placed in the middle of the board to ensure that FMC cards can fit within the chassis when installed on the Peripheral Module.

Figure 5.7: Peripheral Module layout arrangement

5.5.2 PCB Layer Stack

The complexity of implementing multiple high-speed signals and power circuitry suggested an eight-layer PCB is most appropriate. The main advantage with an eight-layer PCB allows for simpler routing strategies on different signal and power layers. In the case where a layer becomes congested with other signals, a less congested layer can be utilised, which is simpler than changing the routing strategy. In addition, if an area had multiple components that uses the same voltage rail, a power layer can be used for a split plane. This split plane can power these common components by using a single polygon pour.

The PCB manufacturer provide a standard 1.6 mm, eight signal layer board which met the PCIe recommended board size of 1.6 ± 0.2 mm. The PCB layer stack description of the board is illustrated in Figure 5.8. It consists of 3.9 FR-4 dielectric material, along with copper planes that can be utilised as internal voltage planes. Both System Controller and Peripheral Module are given the same layer stack as they use the same characteristic impedance required by some signals. This is discussed further in the next section.

5.5.3 Characteristic Impedance

PCIe and LVDS signals are required to meet characteristic impedance. These standards require a single-ended and differential impedance of $50 \pm 10 \Omega$ and $100 \pm 10 \Omega$ respectively. To meet these requirements, the track width of these signals needs to be determined using microstrip and stripline equations [65]. Table 5.1 are the required parameters needed to determine the width for both microstrip and stripline equations. These values are obtained by the PCB manufacturer.

The microstrip topology applies to signals on the top or bottom layer of the PCB. Substituting the parameters from Table 5.1 and various track width values into the microstrip single-ended equation, a track width of

Figure 5.8: Layer stackup for the System Controller and Peripheral Module

Table 5.1: Impedance parameters for the Peripheral Module

Parameter	Value
Dielectric constant ϵ_r	3.9
Dielectric thickness h (stripline case)	0.13 mm
Dielectric thickness h (microstrip case)	0.18 mm
track thickness t	0.036 mm

0.16 mm is chosen as this gave an impedance of 58.76Ω . This meets the single-ended impedance requirement while keeping tracks thin as possible to simplify routing. With a chosen track width, the microstrip differential equation is used to determine the track separation h for differential signals. Separation is kept as narrow as possible to route between components that have fine-pitch pins. A separation of 0.15 mm is chosen as it results in a differential impedance of 98.88Ω , which meets the differential impedance requirement for top and bottom layer signals.

For internal differential signals, the stripline equation is used. The same approach is taken to determine the width and separation of a differential signal using stripline single-ended and differential signals. This

results in a track width value of 0.12 mm, with a single-ended impedance value of 52.88 Ω . The track width is then substituted into the differential stripline equation. It is found that a track separation of 0.18 mm yielded a differential impedance of 95.44 Ω , thus meeting differential characteristic impedance requirements for internal signals.

5.5.4 Routing Guidelines

High speed signals are most likely to impact or to be impacted by other signals. Therefore, the best approach when designing a system with high-speed signals is to route them first. This ensures that routing rules can be followed without interference from other signals. Texas Instruments has released a high-speed signal routing application note [16] which is applied to relevant high-speed PCIe and LVDS I/O signals found on the Peripheral Module.

High speed signals must consider the location of split-planes. If a high-speed signal is routed across split, as illustrated in Figure 5.9, it can lead to addition of signal jitter and signal attenuation, and thus should be avoided.

Figure 5.9: An example of routing over a split plane [16]

From the application note, it is recommended that they are routed over a solid ground reference plane. For each signal layer of the PCB, a ground split plane is defined on an adjacent copper plane. This ensures high-

speed signals are routed over a ground reference with no splits. Figure 5.10 illustrates an example of a ground reference plane. The shaded ground reference plane is located on layer two copper plane. It encompasses the LVDS and PCIe high-speed signals routed on the adjacent top signal layer.

Figure 5.10: An example of a ground reference plane (highlighted) on layer two

Most high-speed signals utilised in both boards are differential signals. The two lines must be routed to the same length to ensure components of each signals reach the destination at the same time. Without length matching, the differential signal could be received incorrectly. Figure 5.11 illustrates a matched differential signal. To ensure the two lines are length matched, the shorter line out of the two are given "zig-zag" routing. This increases the length of the line to a point where it matches the routing length of the longer line.

Another source of reflection in high-speed signals are surface-mount devices (SMDs). The pads on SMDs can introduce discontinuities which

Figure 5.11: An example of a length matched differential signal (highlighted blue) on the FMC Peripheral Module

can degrade the signal quality. However, all PCIe TX lanes requires AC coupling capacitors to meet PCIe specifications. From the TI application note, it recommends using 0402, 100 nF capacitors. This guideline is applied to all PCIe TX lanes. In addition, these capacitors are placed symmetrically on differential signal lines to further minimise reflection as illustrated in Figure 5.12.

5.6 PCIe Hardware Design

The TE0712 has the physical capability to implement PCIe Generation 2 link. This is accomplished on both modular solutions through a PCIe hard block and GTP transceivers. Interfacing between the System Controller and Peripheral Module within a PXIe chassis requires a reference clock and data lanes to be connected from the GTP transceivers to the backplane.

Figure 5.12: An example of AC coupling capacitor placement to reduce potential reflection [16]

As PCIe is a high-speed serial bus standard, appropriate high-speed PCB routing techniques, discussed in the previous section, are implemented to ensure signal integrity.

This section overviews the implementation of PCIe hardware on the Peripheral Module.

5.6.1 GTP Architecture

On board both the Zynq 7015 and the Artix-7 100T are four GTP transmitters (TX) and four GTP receivers (RX). GTP transceivers [17] allows high-speed serial data to flow to and from a data trace on the PCB to FPGA logic. For a PCIe application, the GTP architecture implements the PCIe physical layer as discussed in Section 2.4. Both TX and RX GTP architecture are illustrated in Figure 5.13. Both architectures are made up of a physical coding sub-layer (PCS) and a physical media access (PMA). In the TX architecture, parallel data from FPGA logic flows from the PCS to the PMA, where it is converted into serial data to data traces. In the RX architecture, the opposite conversion takes place. Serial data from PCB traces flows through the PMA, converted to parallel data in the PCS and into FPGA logic.

Figure 5.13: GTP TX and RX transceiver architecture [17]

Within the TX and RX PCS is an 8b/10b encoder and decoder respectively. The PCIe standard require transmitted 8-bit data words to be mapped to 10-bit symbols, otherwise known as 8b/10b encoding. An advantage of using 8b/10b encoding in PCIe is to achieve DC balance to prevent potential distortion of the signal, while obtaining precise data bit location as both data and clock are sent in one signal. The end result of the 8b/10b encoded signal is that the receiver can extract PCIe data and an embedded clock from one signal.

The GTP's PMA utilises two main blocks when implementing PCIe. These are the pre-emphasis and driver block. After parallel data is converted to serial data, the pre-emphasis block of the TX's PMA increases the magnitude of specific frequencies within the signal to improve the overall signal-to-noise ratio (SNR). Within the RX's PMA, it removes this processing within its de-emphasis block. The PMA's driver block either outputs or receives serial data on PCB traces. This process on the PMA requires a reference clock which control the generation of serial and parallel clocks used by the PMA [17]. The GTP pins on the chosen FPGA/SoC solutions need to source an external clock which is later discussed in the following section.

5.6.2 PCIe Implementation on the Peripheral Module

Unlike the System Controller, the Peripheral Module is only required to connect a PCIe x4 link that originates from the System Controller. To achieve this, the GTP receive and transmit differential data lines are connected to the PCIe x4 link through the XJ3 backplane connector. In addition, GTP transceivers of the Peripheral Module require a 100 MHz clock, which is again provided by the XJ3 backplane connector originating from the System Controller. The use of a common clock synchronises the System Controller and Peripheral Module as they both reside in the same PCIe topology. The implemented Peripheral Module's PCIe hardware schematic is illustrated in Figure 5.14.

5.7 Clock System

The PXIe standard allows for synchronisation between multiple modules and even separate chassis through a connection to a PXIe timing module. The timing module provides high-quality clocks and trigger lines to implement multi-device synchronisation. These signals may be directly con-

Figure 5.14: Schematic of Peripheral Module's PCIe connections

nected from the timing module using a SMA cable or obtained through the PXIe DSTAR lines running along the chassis backplane. Although this feature is not a requirement for the thesis, building a foundation simplifies potential future synchronisation implementation.

The design of the Peripheral Module includes a clock system that introduces high-quality clocks, which are then distributed throughout each module and can be utilised by the on-board FPGA. This section details the design of the clock system for the Peripheral Module.

5.7.1 Peripheral Module Clock System

On the Peripheral Module are three clock domains users can choose for synchronising their applications. These are the PXIe chassis on-board 100 MHz reference clock, the 100 MHz clock from a system timing board and

a clock source which can be sourced from the PXIe DSTAR line originating from the timing board.

A LMK00306 is used on the Peripheral Module to fan out the input clock to different parts of the module. It is a six output differential clock buffer, but only three outputs is required. However, the package only has two differential clock inputs. The third input only takes in a single ended signal. As the three usable clock domains on the Peripheral Module are differential, one of them is required to be converted to a single ended clock source. The chosen clock is the PXIe 100 MHz reference clock. This is because the quality of the other clocks need to be preserved, removing the choice of converting them to a single ended clock. A SN65LVELT23 LVDS to LVTTL package is used for the conversion.

The three outputs from the LMK3006 is routed to a FPGA clock compatible pin, a Si5338 clock generator on-board the TE0712 module and to a SMA clock output. This arrangement illustrated in Figure 5.15 allows users to build FPGA applications that require synchronisation with one of the three available clock domains. All of these clocks are configured to LVDS25 by default to ensure that signal standards used throughout the Peripheral Module design are consistent.

Figure 5.15: Peripheral Module Clock System

The clock output from the Peripheral Module needs to be converted

from a LVDS signal to a single-ended signal. Figure 5.16 illustrates the implemented circuitry which uses minimal components while preserving board space. It consist of a RF transformer to provide common mode rejection, and two high-pass filters to remove low frequency noise from a 100 MHz signal.

Figure 5.16: Conversion of LVDS to single-ended signal for SMA output

5.8 Peripherals

As discussed in Chapter 2, various Xilinx FPGA development boards were investigated to determine what I/O standards users are utilising in their projects. Most of these development boards utilise Pmod and FMC I/O standards, giving user access to third-party modules. To utilise these modules, their respective standards are incorporated onto the Peripheral Module. There are other peripherals both the System Controller and the Peripheral Module can utilise in their respective design. These are explored within this section, along with their hardware implementation.

5.8.1 UART

In embedded development, a communication interface is important which aid during development and testing. A UART interface is chosen as electronic circuitry to support UART functionality is minimal, yet it is an important feature that is used in later firmware development. The supporting circuitry consists of a USB-to-UART bridge, a steering diode array package and a female mini-usb connector.

The USB protocol encodes its data using NRZI (non-return-to-zero) line coding [66]. The USB-to-UART bridge allows this protocol to be either converted to/from UART serial data. Data lines are susceptible to transients caused by electrostatic discharge (ESD) and other induced voltages. Decoupling capacitors on the UART voltage lines could be used to act as a charge pump to negate the effects of induced voltages. However, it may not deal with fast transients effectively, which may cause damage to the surrounding circuitry. A steering diode array SOT-23-6 package is chosen to deal with faster voltage transients while minimising PCB space.

The female mini-USB is conveniently placed at the entrance of the 1062Q chassis, allowing users to easily access the connector while the Peripheral Module is installed. This ensures that there is a distance between the female mini-USB connector and an FMC card to prevent potential mechanical obstruction. As part of the USB standard, the two data lines from the female connector forms a differential pair. The pair from the bridge to the female connector are routed with $90\ \Omega$ target impedance to meet the USB 2.0 specifications.

5.8.2 Pmod Interface

Implementing a Pmod interface on the Peripheral Module allows users to utilise Pmod modules in their application. The standard is simple to implement consisting of a twelve-pin connector. Ten of these pins are dedicated to I/O while the remaining two are for VCC and ground. Some

Pmod modules utilise LVDS lines [67, 68, 69]. To incorporate these modules, all Pmod I/O pins are routed with a $100\ \Omega$ characteristic impedance back to the TE0712 FPGA module.

5.8.3 Joint Test Action Group (JTAG)

The Peripheral Module require a connector to program FPGA, software and firmware. A JTAG interface is placed the Peripheral Module to achieve this. Using a JTAG interface utilises more sophisticated debugging tools within the Vivado suite such as in-software clock frequency measurements, virtual input-output (VIO) and Integrated Logic Analyser (ILA). These features will aid in the development and debugging of hardware applications on the Peripheral Module.

5.8.4 FPGA Mezzanine Carrier (FMC)

An FMC connector provides an interface for the Peripheral Module to utilise available FMC cards from manufacturers. The FMC standard comes in two connector sizes, Low Pin Count (LPC) and High Pin Count (HPC). Each offers a different maximum user-defined connectivity. The LPC is chosen over the HPC for the Peripheral Module as the TE0712 does not have enough I/O pins to fully utilise the HPC form factor.

With the TE0712 module being able to utilise all FMC LPC signals, the next step is to consider the footprint of an FMC card on a Peripheral Module. A single-width FMC card on a PXIe Peripheral Module should be fully contained within the PXIe chassis without obstructing any internal parts. Illustrated below in Figure 5.17 is the mechanical dimensions of a single ended FMC card.

The best placement for the female FMC connector on the TE0712 carrier board is to align its center with the carrier board horizontal median. This prevents a FMC card to interfere with any internal parts of the chassis. In addition, the FMC connector is also placed close to the TE0712 mod-

Figure 5.17: Mechanical dimensions of a single width FMC card [18]

ule to reduce routing distance of differential signals and thus minimising additional track impedance and capacitance.

Also required by the FMC standard is a 12 V and 3.3 V source which are routed along with the differential signals.

5.9 Power Design

After considering critical signals, the next step is to develop the power circuitry for the Peripheral Module. Development of the Peripheral Module

Table 5.2: Backplane current capacity for each slot in PXIe-1062Q chassi [2]

Slot	+5 V	+3.3 V	+12 V
System Controller	15 A	15 A	30 A
Peripheral Module	6 A	6 A	1 A

power circuitry consists of three steps: determining the power requirements, estimating the potential power draw of the TE0712 during operation and the TE0712 respective power sequence requirements. This section discusses the design of robust power circuitry for the Peripheral Module.

5.9.1 Power Requirements

The TE0712 solution require a 3.3 V voltage source. In addition, there are also two I/O banks on the FPGA which requires an external voltage source.

The PXIe-1062Q provides various voltage rails with high maximum continuous currents to each slot in the chassis as shown in Table 5.2. This ensures that the chassis is able to provide sufficient power to both modules during operation.

Figure 5.18 illustrates the proposed power architecture of the Peripheral Module. The TE0712 is powered by the 3.3 V rail derived from the chassis backplane. On the TE0712 are two separate I/O banks, 15 and 16, which need to be powered separately from the *VIN* input. The proposed functionality for these banks is to drive LVDS25 FMC signals. Therefore, these banks requires a 2.5 V rail, which is obtained by stepping down the 3.3 V rail. In addition to the TE0712, various peripherals on the carrier card require a 1.3 V and 1.8 V sources. These are again sourced from the 3.3 V rail from the chassis.

Figure 5.18: Peripheral Module power architecture

5.9.2 TE0712 Power Estimation

When designing the power circuitry for the Peripheral Module, it is important to determine an estimate power consumption during operation. An estimation of the power usage determines what power components are required for functional TE0712 operation.

Estimating the TE0712 power consumption follows the same procedure discussed in Section 3.5.2. The parameters used for estimating the power consumption of the TE0712 is illustrated in Figure 5.19. The estimation is again given an unlikely scenario with all the FPGA resources being used along with driving all FMC differential signals and implementing a PCIe Generation 2 x4 link.

With these test parameters, XPE concluded a total on-chip-power of 4.088 W. In the Peripheral Module power architecture, the 3.3 V rail from the chassis' backplane is used as the voltage source for the TE0712 module. From Table 5.2, the 3.3 V rail has a current rating of 20 W which exceeds the estimated total on-chip-power by a factor of 5. The large power overhead already supplied from the backplane simplifies the design of the power

The dialog box is titled "XPE Quick Estimate - XC7A100TFGG484-2LE". It contains several sections for configuring the power estimation:

- XC7A100TFGG484-2LE**
  - Conditions: Typical, Ambient=25C
  - Environment: 250 LFM
  - Voltage: ☒ Nominal ☐ Maximum
- Design Activity**

	Clock		Toggle		Enable	
Logic	150	MHz	12.5	%	50	%
BRAM	150	MHz	50	%	25	%
- Design Utilization**

			%
LUT	63400		100.0
FF	126800		100.0
BRAM	270		100.0
DSP	240		100.0
- Physical Interfaces**

		Number	Width	Rate	
Memory	DDR3	1	36	800	Mb/s
GTP	PCIe Gen2	1	4	5	Gb/s
GTP					Gb/s
LVDS	In 17 Out 17			667	Mb/s

Buttons: OK, Cancel

Figure 5.19: XPE settings for TE0712 power consumption estimation

circuitry on the Peripheral Module.

Figure 5.20 shows the maximum current draw for each voltage rail estimated by XPE. The I/O estimation is based on LVDS25, as it is the standard utilised by FMC I/Os. The maximum estimated current draw on the FPGA's 2.5 V voltage rail is 0.257 A. The 2.5 V rail from the carrier card therefore must provide a minimum of $0.25 \times 2.5 = 0.65W$ of power at 2.5 V. With this information, a LP3878 800 mA voltage regulator [70] is chosen to supply the 2.5 V rail on the Peripheral Module as it provides a factor of 3 safety margin.

5.9.3 Power Sequencing

Unlike the System Controller power sequencing design discussed in Section 3.5.3, the TE0712 has less stringent start up requirements. The technical reference manual [15] recommends that all the power sources to be

Power Supply		
Source	Voltage	Total (A)
V _{CCINT}	1.000	1.653
V _{CCBRAM}	1.000	0.017
V _{CCAUX}	1.800	0.428
V _{CCAUX_IO}		
V _{CCO} 3.3V	3.300	
V _{CCO} 2.5V	2.500	0.257
V _{CCO} 1.8V	1.800	
V _{CCO} 1.5V	1.500	0.250
V _{CCO} 1.35V	1.350	
V _{CCO} 1.2V	1.200	
-		
MGTAV _{CC}	1.000	0.299
MGTAV _{TT}	1.200	0.245
-		
-		
-		
-		
-		
V _{CCADC}	1.800	0.020

Figure 5.20: TE0712 power supply current draw

powered up simultaneously which removes the need for a sequence power supply solution. Separate voltage regulators are used to set up the voltage rails, which are all sourced by the 3.3 V power supply from the PXIe backplane. This in turn powers up all voltage rails simultaneously when the 3.3 V rail from the backplane is supplied to the module.

5.10 Fabrication of the Peripheral Module

EMC Ltd. is again contracted to fabricate the Peripheral Module. The PCBdoc containing the Peripheral Module PCB design is handed over along with the bill of materials associated with the module. During the fabrication process, the same problem occurred concerning with some IC pad spacing being too narrow. These pads are accounted so they can be

inspected for solder bridges once the boards arrive. Additionally, an incorrect device package is provided in the BOM for the LP3878 voltage regulator discussed in Section 5.9.2. An 8-WDFN package is utilised which did not fit the SOT23-8. Another order was required for the correct voltage regulator package.

5.11 Summary

In this chapter, the hardware design of the Peripheral Module is discussed. The steps taken when designing the module started with the mechanical requirements. The Peripheral Module meets the 3U Eurocard form factor along with their respective backplane connector requirements. This is followed by selecting its modular FPGA solution. The design choice of using a modular solution rather than building a standalone PXIe-FPGA module removes the complexity by only focusing on interfacing with the module. The Peripheral Module utilises various high-speed signals such as PCIe and LVDS lines. Proper high-speed routing and PCB guidelines are implemented to minimise loss in signal integrity. With most critical signals routed, various peripherals are implemented to meet the module's functionality objectives. Using the XPE tool, a robust power circuitry is developed for the Peripheral Module. With the hardware development completed, the FPGA and software firmware can be developed for both implemented System Controller and Peripheral Module. The developed Peripheral Module (illustrated in Figure 5.21) is received a few days before the conclusion of the project. There are no solder bridges with the affected pads that did not utilise a solderdam mask and the SOT23-8 LP3878 are fitted correctly. The overall hardware schematics is provided in Appendix A.

Figure 5.21: Populated Peripheral Module PCB with AD9467 FMC card

Chapter 6

Firmware Development

To achieve the desired functionality of the developed modules, firmware is required. This chapter discusses the design of the System Controller and Peripheral Module's respective firmware for the open PXIe system.

6.1 Open PXIe System Firmware Objectives

After arranging fabrication of the System Controller and Peripheral Module, their associated firmware is devised. The System Controller firmware consists of three parts. Firstly, a PCIe root-port on FPGA fabric to enable memory read and write transactions to the Peripheral Module. Secondly, a Linux OS to provide a user interface and a software development environment. Finally, a custom driver to interface the root-port in the PL from the Linux OS. This is accompanied with an API to abstract the low-level driver to simplify software development.

The prior work within the School of Engineering at VUW resulted in firmware, a Linux OS and device driver for the Zynq 7045 but was limited to a single PCIe lane and only one peripheral module.

As for the Peripheral Module, the firmware contains logic to implement a PCIe endpoint to facilitate communication with the System Controller. Additionally, the Peripheral Module has an FMC connector on-

board. To demonstrate usability of the Peripheral Module's FPGA fabric on top of its PCIe connectivity logic, an RTL interface is developed for an FMC card. A MicroBlaze soft-core processor is utilised to control an FMC card through SPI using in-built libraries. This is simpler than developing an RTL SPI controller.

In summary, the firmware objectives for each module are:

- Develop PCIe root-port logic on FPGA fabric for the System Controller to provide PCIe x4 communication with peripheral modules
- Develop a Linux image to allow the System Controller to host an embedded OS
- Build a PCIe driver associated with the Linux OS to support multiple peripheral modules
- Simplify software development on the System Controller by wrapping the PCIe driver with an API
- Develop PCIe endpoint logic on FPGA fabric for the Peripheral Module to communicate with the System Controller
- Develop an RTL interface to interface with an FMC card
- Ensure that there is enough left over resource utilisation ($\leq 50\%$ for all resources) for custom user RTL applications.

With the objectives set out for both modules, their proposed firmware architecture are illustrated in Figure 6.1.

6.2 Xilinx Toolset

There are two parts to developing the modules' firmware. This consists of developing the FPGA logic for both modules, and an embedded Linux

Figure 6.1: High-level firmware architecture

OS for the proposed System Controller. Both parts require specific tools to build and deploy their respective solution. This section highlights the tools used in the thesis and their justification for their usage.

6.2.1 Vivado Design Suite

As the project utilises an Artix-7 FPGA and Zynq SoC, the Vivado Design Suite [71] is chosen as it supports these devices. Vivado is software produced by Xilinx for synthesis and analysis of hardware description language (HDL) designs for their 7-Series devices. In addition, it is coupled with an SDK environment to develop and run bare-metal code on either the Zynq's ARM processor, or an FPGA utilising a MicroBlaze soft-core processor. This is useful when interfacing with an FMC card which is to be discussed in Section 6.5.1. An alternative to Vivado is the ISE environment which was also developed by Xilinx. However, ISE does not support the newer 7-Series devices[72]. The lack of support within the ISE environment runs the risk of making firmware development arduous on both boards.

6.2.2 Petalinux

Petalinux [73] produced by Xilinx, is a toolset that allows customisation, building and deployment of embedded Linux solutions on Xilinx's processing systems. Apart from Petalinux, the Yocto project [74] offers a standardised build environment for embedded systems. It has support for multiple computing architectures including ARM. Development of a Linux kernel using Yocto is time consuming as it provides a wide range of kernel customisation. Although customisation may be useful, the project only requires a basic embedded OS solution. Petalinux is an embedded Linux toolset for Xilinx devices. Its simplistic nature accelerates design productivity. This is achieved through Xilinx tool integration, which allows a kernel to be developed and deployed in a few minutes. As compatibility and ease of development are two factors that minimises the project risks, Petalinux is chosen over the Yocto project.

6.3 Advanced eXtensible Interface (AXI) Bus

Packaged with Xilinx's Vivado Suite are AXI IP HDL soft-cores. As part of the *Advanced Micro Controller Bus Architecture* (AMBA) open standard, AXI is a family of micro controller buses that targets high performance, high clock frequency systems[75]. Xilinx has adopted the AXI protocol to increase productivity, availability and flexibility. Productivity is achieved by standardising the AXI interface, allowing developers to only learn a single protocol for FPGA development. As AXI is an industry-standard, a large consortium of IP providers are available that support the AXI protocol. With different types of AXI interfaces, the standard is flexible for different applications [76]. These are:

- *AXI4* is for memory mapped interfaces and allows burst of up to 256 data transfer cycles with just a single address phase.
- *AXI4-Lite* has a small logic footprint and is a simple interface for single transaction memory mapped interface.
- *AXI4-Stream* allows unlimited data burst size. These interfaces do not have an address phase and are therefore not considered to be memory mapped.

The following sections discusses the use of IP FPGA blocks from Xilinx and Analog Devices. All of which utilise an AXI interface that allows communication between either the Zynq's ARM processor or MicroBlaze soft-core processor. Each AXI block within a Vivado design have a unique AXI address which is used in AXI transactions.

6.4 System Controller FPGA Firmware Design

The System Controller uses a Zynq 7015 SoC as its silicon device. Its hybrid nature permits running an embedded Linux OS on its ARM cores

while interfacing RTL logic within the FPGA fabric. The requirement for the System Controller firmware is to develop a Linux-based solution on the PS, while interfacing with a PCIe root-port implementation on the PL of the Zynq. The interface between the two systems is a device driver loaded on the OS. Through this driver, the OS can send and receive data with the proposed PXIe Peripheral Module endpoint. This section discusses the system controller FPGA firmware design choices.

6.4.1 Zynq Processing System

Xilinx has provided their Processing System (PS) interface around the Zynq SoC [77] as an IP core. It is utilised in the firmware design of the System Controller to enable peripherals to connect the PS and is a requirement for later OS development. Configuration of the core consists of enabling peripherals which are used in the System Controller. These are the microSD card, USB UART and Ethernet interface. This configuration is later used to build the device tree required for development of the Linux kernel image. Figure 6.2 shows the Zynq PS configuration for these interfaces.

6.4.2 PCIe Connectivity

On-board both the Zynq SoC and Artix FPGA is an integrated PCIe hard block which implements the Data Link, Transaction and Physical Layer which makes up the PCIe protocol [78]. In order for transactions to be sent from the AXI space on the System Controller to PCIe, an *AXI Memory Mapped to PCIe*, or *AXI-PCIe*, is utilised to convert between the two standards. With this block, it handles TLP packet generation and extraction to/from the PCIe hard block.

In the proposed firmware for the System Controller, the Zynq PS, and thus the Linux OS, interfaces with the AXI-PCIe block through the AXI bus. A transaction initiated from the Linux OS is converted into an AXI memory read or write transaction internally within the PS. Once the AXI-

Figure 6.2: Peripheral configuration in Zynq PS core

PCIe block receives an AXI transaction, it is translated into a TLP in the *Slave Bridge* and sent to the PCIe hard block to convert the TLP to electrical signals on the PCIe physical link. In addition, the AXI-PCIe block also translate PCIe memory read and write request TLP packets within the *Master Bridge* to AXI transactions. A high-level architecture of the AXI-PCIe block is depicted in Figure 6.3.

Configuration of the AXI-PCIe core for the System Controller firmware consists of selecting the type of device, link width and BAR configuration. Within the System Controller firmware, the core is configured to a *root port* device, to support operations carried out by a host. In the previous implementation of the root port, it is limited to a PCIe x1 link. The design for the new System Controller expands this link width to x4 which is configured

Figure 6.3: High-level AXI-PCIe architecture [19]

in the AXI-PCIe *link width size* setting. The AXI-PCIe is configured with a 1 GB, 32 bit BAR to map the entire DDR memory to the PCIe address space as depicted in Figure 6.4. This potentially allows a PCIe endpoint, such as a PXIe Peripheral Module, to have access to the entirety of the System Controller's DDR memory to store and download data.

Base Address Registers

Base Address Registers (BARs) serve two purposes. Initially, they serve as a mechanism for the device to request blocks of address space in the system memory map. After the BIOS or OS determines what addresses are to be assigned to the device, the Base Address Registers are programmed with these addresses and the user implemented PCI-PCI bridge must use this information to perform address decoding.

☐ BAR 64 bit Enabled

BAR 0 Options

☒ Bar0 Type Memory

Size 1 Gigabytes

AXI BAR MAP 0 Options

AXI-PCIe BAR Translation 0x00000000

Figure 6.4: System Controller Zynq PCIe BAR configuration

In a Zynq architecture, DDR memory can only be interfaced through a memory interface controller within the PS. To allow the AXI-PCIe block in the PL to interface with this memory controller, a slave AXI port on the Zynq PS needs to be utilised. However, the Zynq PS provides two ports

that interfaces with the memory controller. The first available option is to utilise a *General Purpose* (GP) port. The AXI-slave GP port allows direct access to the memory controller on the PS from the PL. However, the simplistic architecture of this port limits the speed of data transfer to the DDR memory. An alternative is the *High-Performance* (HP) port. Like the GP port, it has direct access to the memory interface controller but features FIFO buffers with bandwidth management [56]. This allows higher transfer speeds from the PL to DDR memory. As the HP interface offers a higher speed access to DDR memory, it appeals for high-speed applications. Because of this reason, the HP port is chosen as the interface between the AXI-PCIe block to DDR memory.

Overall, the design of the System Controller PCIe root-port allows memory read and write transactions to the PXIe Peripheral Endpoint. This is achieved using a Zynq PS and AXI-PCIe block. The design utilises an HP port to potentially allow the Peripheral Module to have fast access to the System Controller's DDR memory in high-speed applications. To initiate memory read and write transactions from the proposed Linux OS, a driver is required to interface with the PL logic. The design of this driver is later discussed in Section 6.7.

6.5 Peripheral Module FPGA Firmware Design

The silicon device chosen for the FMC carrier peripheral module is an Artix-7 FPGA. Mentioned in Chapter 1, the objective of the peripheral module is to provide I/O capabilities in an open PXIe ecosystem. In addition, the Peripheral Module requires a communication channel with the System Controller. To fulfil these requirements, the FPGA firmware requires logic to interface with the System Controller's PCIe root-port and RTL logic to interface with FMC cards. Discussed in this section is the FPGA firmware design for the Peripheral Module that fulfils these two requirements.

6.5.1 FMC ADC Interface Example

Mentioned in Chapter 1, the School of Engineering at VUW developed a peripheral module that functions as an ADC to be used for MRI imaging. To showcase the Peripheral Module in an MRI setting, potential FMC cards used for the thesis are restricted to ADC functionality. However, users can develop their own RTL design on the Peripheral Module's FPGA to interface with any FMC cards.

The design consists of a high speed ADC interface and PCIe interface to allow high speed data acquisition applications. A 16-bit analogue-to-digital (ADC), AD9467 FMC card by Analog Devices [20] is chosen to meet this requirement. It is controlled by a Microblaze processor through a Serial Peripheral Interface (SPI). Although there are other ADC FMC cards recommended by Xilinx, Analog Devices has provided open-source Verilog soft-cores [79] and drivers [80] that interfaces with the AD9467 FMC card. Using these soft-cores simplifies the FPGA firmware development for the Peripheral Module. The overall Peripheral Module FPGA firmware including both AD9467 and PCIe connectivity is illustrated in Figure 6.5.

Figure 6.5: Overall FPGA firmware architecture for Peripheral Module including AD9467 and PCIe connectivity

On-board the AD9467 FMC card, an analogue input is converted into

a 16 bit digital representation. Data is clocked out of the AD9467 on the rising and falling edge of a 250 MHz Data Clock Output (DCO). The waveform for this process is depicted in Figure 6.6.

Figure 6.6: AD9467 timing diagram [20]

This technique is called Double-Data-Rate (DDR), where data transfers are occurring on both rising and falling clock edges. The AD9467 HDL soft-core provided by Analog Devices, extracts the interleaved bits and rebuilds the original 16 bit digital signal.

Shown in Figure 6.7 is part of the AD9467 HDL interface RTL schematic that implements DDR bit extraction for an LVDS pair. Both components of the differential signal are buffered into a single-ended signal and placed into an Input Double-Data-Rate (IDDR) primitive. An IDDR is an I/O resource which splits each LVDS pair into two data bits on every clock cycle of the DCO as illustrated in Figure 6.8. With all captured data bits, the original 16-bit digital representation is rebuilt and outputted into Analog Device's DMA soft-core controller.

A DMA controller soft-core from Analog Devices is utilised to transfer the 16 bit digital representation to the TE0712 local DDR memory. The DMA controller takes in 16 bit data from the HDL interface and is connected directly to a DDR memory interface. The drivers provided can set

Figure 6.7: Bit extraction RTL schematic

Figure 6.8: IDDR primitive timing diagram [21]

the start destination address in the *CURRENT_SRC_ADDRESS* register in the DMA soft-core. From here, the DMA controller writes the 16 bit data

Illustrated in Figure 6.9 is the simplified data flow from the AD9467 FMC card to the TE0712 DDR memory. It utilises three soft-cores. Firstly, the AD9467 interface is used to translate the DDR signal into a 16 bit data. Secondly, a DMA controller interfaces with the DDR memory interface through posted AXI transactions. Thirdly, the DDR memory interface receives the 16 bit data and transfers it to DDR memory. This design allows a digital representation of an analogue signal to be stored in DDR memory. The following section consists of configuring the AD9467 C-code drivers on the Microblaze.

6.5.2 Microblaze Firmware Development

The AD9467 can be controlled through its SPI bus. Potentially, an RTL solution can be developed that interacts with the AD9467 through this bus. However, an existing open driver solution given by Analog Digital is more attractive, which reduces the need to develop RTL. Execution of the drivers requires a processor, which the Artix-7 on the TE0712 module does not have unlike the PicoZed. A soft-core processor is therefore needed to run the AD9467 driver code.

Xilinx designed a soft-core microprocessor for Xilinx's FPGA line called the Microblaze with its entirety implemented in the fabric of Xilinx FPGAs. Much like the Zynq PS, C or C++ code can be executed to interface with the FPGA fabric. Development and compilation of software for the Microblaze takes place in Xilinx's Software Development Kit (SDK) environment.

Analog Devices provides the AD9467 drivers as C source code. For the drivers to be executable on the Microblaze, it needs to be first compiled using a compiler system called GNU compiler Collection (GCC). GCC translates the C source code into assembly language in the form of an object code file. The contents of the object code file contain the source code in a binary format, but it is not directly executable by the microprocessor. This is because the translated object code file is not yet cross-compiled with object code files of used libraries in the source code. A linker script is used to collate all the required library object code files with the output of the GCC compiler. The result of the linker script is an Executable and Linkable Format (ELF) file, which can be executed on the Microblaze. The benefit of using the Xilinx's SDK environment is that compilation and generation of an ELF file is automated, simplifying the whole process. If done manually, there are four different executables that GCC needs to call along with the development of a linker script to link all the necessary object files.

With a generated ELF file, the drivers can be executed by the Microblaze. This completes the AD9467 FMC card interface on the Peripheral

Module. What is required next in the Peripheral Module's firmware is PCIe connectivity, which is discussed in the next section.

6.5.3 PCIe Connectivity

Mentioned in Section 6.4.2, the proposed System Controller firmware design adopts the Xilinx AXI-PCIe IP core to interface with the PCIe hard block within the PicoZed fabric. For the Peripheral Module to receive PCIe memory read and write transactions from the System Controller, it utilises a AXI-PCIe core in an endpoint configuration.

From the previous section, a DMA engine is utilised to store 16 bit captures from the ADC directly into DDR memory. Another DMA engine is utilised on the Peripheral Module's FPGA firmware to allow fast data transfers from its DDR memory to the System Controller. The difference between the AD9467 system and the PCIe system is that the PCIe system utilises a DMA controller from Xilinx, rather than the IP core from Analog Devices. This is because the controller from Analog Devices is not compatible with the AXI-PCIe block.

The proposed PCIe-DMA sub-system on the Peripheral Module FPGA firmware consists of the DMA controller being an AXI master of the AXI-PCIe and the DDR memory interface. In this arrangement, the DMA controller acts as medium for transferring of data between the DDR memory and AXI-PCIe block. The System Controller can write to the source and destination address registers of the DMA controller and initiate a transfer. This is discussed in Section 6.7. The overall PCIe-DMA subsystem is illustrated in Figure 6.10.

With a PCIe-DMA architecture, the AXI-PCIe block needs to be configured to allow the System Controller to access the DMA engine to initiate transactions. Through the PCIe BARs configuration, BARs can be mapped to an AXI address space. The AXI address map for the Peripheral Module is illustrated in Table 6.1. This address arrangement ensures enough

Figure 6.10: PCIe-DMA implementation

Table 6.1: Peripheral Module AXI address space

AXI Peripheral	Offset Address	Range	High Address
AXI-PCIe	0x81000000	16 kB	0x81007FFF
AXI-DMA	0x81008000	16 kB	0x8100BFFF
AXI-BRAM	0x8100C000	32 kB	0x8100FFFF

address space is allocated for each AXI peripheral. The addresses are non-overlapping because if the addresses were overlapped, it may result in the AXI-peripherals to be non-functional.

With this arrangement, BAR0 PCIe to AXI translation is configured to 0x81000000 with a size of 64 kB. This allows the System Controller to access any of the three AXI peripherals listed above if it knows the offset. For example, if the System Controller wants to write to *DMA_SRCADDR* register within the DMA controller, the AXI-DMA offset is 0x00008000 and a register offset of 0x18. Therefore, the address the System Controller needs to write to is 0xXXXX8018. The four MSB values represents the BAR0 address for the Peripheral Module. This is stored within the Linux's system mapped addresses table during initialisation of the Peripheral Module. This is further discussed in Section 6.7.

The PCIe-DMA sub-system and the AD9467 interface logic make up the Peripheral Module FPGA firmware. This provides the module to interface with an AD9467 ADC while allowing the System Controller to grab ADC data stored on DDR memory through a DMA controller. Overall,

the Peripheral Module's FPGA firmware is designed around System Controller's PCIe driver.

6.6 Linux OS Development

In traditional PXIe systems, user interactions are done through a System Controller's OS. An embedded Linux OS is chosen for the proposed System Controller as it is compatible with the Zynq SoC. A core component that makes up the Linux OS is the Linux kernel. The kernel is replaceable software within the OS that interfaces with hardware such as the CPU and memory devices. In the System Controller, the kernel will act as the communication medium between the FPGA fabric and the OS.

The Petalinux toolset is used to develop the Linux image. Building a Linux image consists of importing an FPGA bitstream from Vivado, configuring the kernel with applications and modules, configuring the kernel and rootfile system and finally, building the Linux image. The Petalinux toolset is wrapped in an API which simplifies and automates most of the Linux image development process.

After creating a new Petalinux project, the System Controller's FPGA bitstream from Section 6.4 is imported through the *petalinux-config* command. Importing the FPGA bitstream generates a data structure called a device tree that describe hardware attributes to the Linux kernel. These hardware attributes are the blocks used in the FPGA design. An example is the device tree entry for an AXI interconnect block, illustrated in Listing 6.1. Within this entry are properties inherent to the AXI interconnect. The two main properties are the *compatible* and *reg*. From the example listing, the *compatible* property identifies the bus controller as a *simple-bus*, indicating it is a simple memory-mapped bus with no driver. The *reg* property describes the address and its length of a register set for the interconnect.

The configuration stage consists of customising the kernel and root file system. To access Linux PCIe tools such as `/textitlspci`, the kernel needs to

```

ps7_axi_interconnect_0: amba@0{
    #address-cell = <1>;
    #size-cells = <1>;
    compatible = "xlnx,ps7-axi-interconnect-1.00.a", "
simple-bus";
    ranges;
    ps7_afi_0: ps7-afi@f8008000{
        compatible = "xlnx,ps7-afi-1.00.a"
        reg = <0xf8008000 0x1000>;
    };
};

```

Listing 6.1: An example of an device tree entry

be configured to provide PCIe bus support. This configuration is shown in Figure 6.11. The root file system within a Linux OS contains all user space components. In this thesis, the configuration of the root file system includes the PCIe driver (section 6.7 and its API section 6.8).

Figure 6.11: Enabling PCIe support during kernel configuration

The final step is to generate a Linux image using the *petalinux-build* command. This image can be copied onto a microSD card. Booting the PicoZed from a SD card is configured through a dip switch on the mod-

ule. The boot up process of the Linux image consists of two steps. Firstly, the Zynq PS loads the Linux kernel which verifies the hardware interface, followed by loading the user-space. From here, the user has access to their application built during the configuration of the rootfs, along with initialising drivers. The development of the PCIe drivers will be discussed in the subsequent section.

6.7 Linux PCIe Driver

The Peripheral Module FPGA firmware is discussed in 6.5.3. It utilises a PCIe-DMA sub-system that laid the framework for the PCIe driver design. This section focuses on the design of the PCIe driver, which consists of PCIe device instantiation and interfacing with the Peripheral Module's PCIe-DMA sub-system to transfer data back to the System Controller. With the previous implementation of a PCIe driver, it is limited to interfacing with one peripheral module. A entirely new driver is discussed in this section which potentially expands the PCIe connectivity to multiple devices.

To perform data transfers to and from the Peripheral Module, a PCIe driver, with the name "xpcie", is developed for the System Controller. The driver is located in the kernel space of the Linux OS, which provides an interface to hardware on the System Controller module. The class of the driver is a *character (char) driver*. As the Peripheral Module can be accessed like a file, a char driver is the most suitable driver class. Other driver classes are *block devices* and *network interfaces*. These classes do not match the functionality of the Peripheral Module and hence are not chosen for the driver design.

The requirements of char style driver are to implement the following system calls:

- probe

```

static int xpcie_probe(struct pci_dev *pdev, const struct
pci_device_id *id)
static void xpcie_remove(struct pci_dev *pdev)
MODULE_DEVICE_TABLE(pci, xpcieids)
static int __init xpcie_init(void)
static void __exit xpcie_exit(void)sub

```

Listing 6.2: PCIe functions

```

static struct pci_driver xpcie_driver = {
    .name = xpciename,
    .id_table = xpcieids,
    .probe = xpcie_probe,
    .remove = xpcie_remove,
};

```

Listing 6.3: PCIe driver structure

- remove
- __init
- __exit

System calls are requests made to the Linux kernel. The above system calls are hardware services requests that are made to a device, which in this case, is the Peripheral Module. These system calls are implemented as shown in Listing 6.2.

A standard PCIe driver structure, is utilised to map these implemented system calls to the generic system call functions, illustrated in Listing 6.3. PCIe drivers are required to have an *init* and *exit* function. These functions handle the registration/de-registration of the device in the kernel address space. These system calls are called upon during the execution of kernel command *pci_register_driver* and *pci_unregister_driver*.

The *probe* function is called to start the registration of the Peripheral Module into the OS. This consists of enabling the device to make it visible to the user and mapping all the PCIe endpoint BARs to the OS virtual

```
struct file_operations xpcie_fops = {  
    .read = xpcie_read ,  
    .write = xpcie_write ,  
    .unlocked_ioctl = xpcie_ioctl ,  
    .open = xpcie_open ,  
    .release = xpcie_release ,  
    .llseek = xpcie_llseek ,  
};
```

Listing 6.4: File operations structure

memory. Discussed in Section 6.5.3, the Peripheral Module is configured with one 64 kB BAR. It has a translation base address of `0x81000000` which allows the System Controller to access the DMA controller and perform either memory read or writes on Peripheral Module's DDR memory. The registration process maps the BAR address into the kernel memory map which is used later to write to the DMA controller registers.

After the PCIe registration process, the next step is to represent the device within the kernel. Devices are shown in the `/dev` directory of the Linux file system. Each device holds a device number pair, consisting of a major and minor part. The major number represents the driver associated with the device while the minor number is used to specifically represent the actual device within the kernel. The `alloc_chrdev_region` is used to allocate device number pair for the peripheral module.

Once the allocation is completed, the character device can be initialised using the `cdev_init` followed by a call to the kernel to represent the device using `cdev_add`. These functions are defined in `include/linux/cdev.h` of the Linux file system. The initialisation process requires a `file_operations` structure defined in 6.4. Discussed before, character drivers are required to implement four system calls. The `file_operations` structure maps each of these system call to their implemented function. These are utility functions that are used with the Peripheral Module. Memory reads and writes to the Peripheral Module can initiated using the `read` or `write` system calls. With

```

//registration
xpcie->bar_len[i] = pci_resource_len(xpcie->pdev, i);
xpcie->bar[i] = pci_iomap(xpcie->pdev, i, xpcie->bar_len[i])
;

//allocation of major and minor device numbers for the
Peripheral Module
int rc = alloc_chrdev_region(&xpcie->cdev_num, 0, 1,
xpciename);

xpcie->major = MAJOR(xpcie->cdev_num);
xpcie->minor = MINOR(xpcie->cdev_num);

//mapping the file operations to the device
cdev_init(&xpcie->cdev, &xpcie->fops);
xpcie->cdev.owner = THIS_MODULE;
xpcie->cdev.ops = &xpcie->fops;

//initialising device
rc = cdev_add(&xpcie->cdev, xpcie->cdev_num, 1);
xpcie->dev_node = device_create(xpcie->class, NULL, xpcie->
cdev_num, NULL, "%s", devname);

```

Listing 6.5: PCIe driver registration and allocation

the completion of the allocation process, the device node can be created to allow the user to access the device. This is done by calling the *device_create* command.

Listing 6.5 shows the work flow to registration, allocation and initialisation of a character driver and the device it is controlling.

With an initialised device represented within the kernel space and the Peripheral Module's BAR address in the kernel memory map, memory reads and writes can be performed on the Peripheral Module. This consists of writing to three registers on the AXI-DMA controller within the Peripheral Module's FPGA firmware. These registers are *CDMA_SRCADDR*, *CDMA_DSTADDR* and *CDMA_BTT*. In the case where the System Controller performs a memory read using the *read* system call, the desired DDR memory address is written to *CDMA_SRCADDR* followed by the

AXI-PCIe address being written to *CDMA_DSTADDR*. The transaction is then initiated by writing the data length to the *CDMA_BTT*, where BTT stands for "bytes to transfer". Writing to this register automatically starts the data movement from the DDR memory address to the AXI-PCIe within the Peripheral Module. The AXI-PCIe sends the data back to the System Controllers root-port which is then stored in a read buffer in kernel space. A *copy_to_user* system call can be used to move the data from kernel to user space. For a memory write to the Peripheral Module's DDR memory, the addresses written to *CDMA_SRCADDR* and *CDMA_DSTADDR* are reversed. This results in a data transfer from the AXI-PCIe block to DDR memory.

Overall, the driver design allows the System Controller to interface with the Peripheral Module PCIe-DMA sub-system to retrieve data stored on DDR memory. Although the source code for the drive will be provided to users, an API is required to abstract these low-level system calls which simplifies the design of user applications within the proposed open PXIe platform. This will be discussed in the following section. In the next chapter, the driver is tested and evaluated on a development board along with the developed System Controller.

6.8 Application User Interface (API) for Users

Discussed in the previous section is the design of a PCIe driver. Development of a driver for a Linux OS requires an understanding of the Linux architecture. The average user may not have this technical background to develop their own applications on an open PXIe system. A solution to this problem is to develop an API. The API provides an abstraction layer between the kernel and user space. The API consists of clearly defined methods which use system calls to request a service from the Linux kernel. The usefulness of an API simplifies the series of commands needed to control a PXIe peripheral module to a single function.

Before using functions from the API, the Linux environment needs to be set-up correctly. This consists of loading the PCIe driver which handles the registration, allocation and initialisation of the Peripheral Module mentioned in the previous section. This is done by using the *insmod* command within the command line on the compiled drivers. Once the driver is loaded, an *xpcie* device is displayed in */dev* folder of the Linux file-system. A *open* system call returns a file descriptor required for reading and writing to the Peripheral Module.

With a file descriptor, the System Controller can initiate memory reads and writes to the Peripheral Module. The API provides methods *write_data* and *read_data* that facilitates data transaction. These methods uses the kernel's *write* and *read* command, which are mapped to the PCIe driver functions previously highlighted in Listing 6.4. Both functions require three parameters: a file structure, user space buffer and an address. The user space buffer stores the data to be sent in write transactions and data obtained from read transactions. However, the kernel requires access to the user space buffer. Dereferencing a user-supplied buffer provides an entry point for user applications to have direct access to memory, compromising the security of the system. Therefore, the *write_data* and *read_data* utilises system calls *copy_from_user* and *copy_to_user* respectively to maintain the security of the kernel.

Provided in the API is a reference to *ioctl* commands to the PCIe endpoint. These commands configure different parameters of transfers provide statistics of transfers that has previously occurred. A summary of the available *ioctl* commands are shown in table 6.2.

Listing 6.6 shows an API use case example of a loopback function. Data stored in a pre-defined buffer *w_data* is written to a DDR memory address on the Peripheral Module. That data is then retrieved from the same memory address using the *read_data*, which stores the data into a pre-defined buffer *r_data*. The speed of this transaction in MB/s can be obtained using an *ioctl* command with a *TRANS_SPEED* specifier. The transfer time is cal-

Table 6.2: Overview of the open PXIe platform API

Command	Description
<code>open_dev(const char *path)</code>	Opens and initialise device
<code>close_dev(const char *path)</code>	Closes device and returns allocated memory
<code>read_data(int dev, unsigned char *read_buf, int addr)</code>	Reads contents from device at address <code>addr</code> then stores it in <code>read_buf</code>
<code>write_data(int dev, unsigned char *data, int addr)</code>	Writes data to device at address <code>addr</code>
<code>iotctl(int fd, TRANS_TIME, 0)</code>	Get previous transfer time
<code>iotctl(int fd, TRANS_SPEED, 0)</code>	Get previous transfer speed

culated by recording the start and end time of the transaction. By dividing the size of the transfer (bytes) with the transfer time, the transfer speed can be obtained.

The API solution for an interface between the user applications and the low-level PCIe driver was discussed in Section 6.7. It employs clearly defined methods which use system calls to interact with the PCIe driver in kernel space. Providing an API to users removes the requirement of having knowledge of the Linux architecture to develop applications on the proposed open PXIe system. This opens the platform to a wide range of potential users with at least some basic coding experience.

6.9 Summary

Discussed in this chapter was the development of the both proposed PXIe modules FPGA firmwares using Xilinx's Vivado Design Suite. The overall FPGA design is provided in Appendix B and C. The System Controller FPGA firmware consists of an AXI-PCIe block in a *root-port* configuration. This is coupled with the Zynq PS through an HP AXI interface, to allow the Peripheral Module to store data in the System Controller's DDR memory. Similarly, the Peripheral Module FPGA firmware consists of the same AXI-PCIe block. However, it is configured to a *end-point* module.

```
/* open device */
fd = open("/dev/xpcie_1");

/* write to DDR memory on peripheral module with buffer
   containing data */
write_data(fd, (unsigned char *)&w_data, 0x00100000);

/* read from DDR memory on peripheral module endpoint */
read_data(fd, (unsigned char *)&r_data, 0x00100000);

/* get previous transfer speed */
printf("Last transfer ran at %d MB/s\n", ioctl(fd, TRANS_SPEED, 0)
);
```

Listing 6.6: API use case example of a loopback function

Along with the AXI-PCIe, a DMA controller from Xilinx is also utilised. Together, they form the *PCIe-DMA* sub-system on the Peripheral Module. This allows the System Controller to initiate memory transactions directly towards the Peripheral Module's DDR memory without MicroBlaze intervention. The PCIe connectivity implemented for both modules allow PCIe communication between the two. This fulfils the functional data channel requirement within the 1062Q.

Another requirement for the Peripheral Module is to provide I/O capabilities within the platform. This is achieved by utilising IP cores and drivers from Analog Devices to store 16-bit data from an AD9467 ADC FMC onto the DDR memory of the Peripheral Module.

The communication channel is solidified through a PCIe driver configured within the developed Linux OS. The driver allows the System Controller to perform memory reads and writes to the Peripheral Module from the Linux environment. To facilitate user development, an API is also developed to aid users in building custom applications for the open PXIe platform without having technical knowledge on Linux architecture.

Upon completion, the respective firmwares are then tested on devel-

oped modules. This is discussed in the following chapter.

Chapter 7

Design Evaluation and Analysis

With both PXIe modules and their respective firmware designed and constructed, they are evaluated against their respective objectives presented in Section 5.2 and 6.1. The development of the System Controller and Peripheral Module are achieved in principle through the hardware design presented in Chapter 3 and 5. However, their compatibility with a PXIe-1062Q chassis must be evaluated, along with their communication channel and API objectives. This chapter discusses the evaluation of the hardware and firmware designs for the two modules.

7.1 Mechanical and Electrical Testing and Evaluation of the Developed Modules

The two modules are designed to meet the PXIe mechanical specifications [5] discussed in Section 5.3. To test mechanical and electrical compatibility, the fully developed boards are connected to the 1062Q chassis, illustrated in Figure 7.1. Both modules are able to fit into their respective slots and both mated successfully with their backplane connectors. Additionally, the AD9467 FMC card is connected to the Peripheral Module to test for obstruction. There is no obstruction between the FMC card and the Sys-

tem Controller, along with chassis itself. This indicates that the mechanical construction of both modules meet the PXIe mechanical specifications outlined in Section 5.3. However, it requires some force to remove the modules from the chassis. Commercial PXIe modules use a lever that aids in removal from the chassis. This could be added in a future revision of both modules.

To confirm that both modules met the PXIe electrical specifications, the chassis is powered up with the modules in their respective socket. Their status LEDs are evaluated as it determines if the on-board power circuitry is working correctly. With all LEDs being lit, it indicates that all the voltage rails are functional. The *power_good* LEDs are also analysed on board the modular solutions. These LEDs are lit indicating that the FPGA/SoC modules are powered up correctly. On board the AD9467 FMC card are two LEDs for the 12 V and 3.3 V rails as per the FMC standard. These LEDs are also lit indicating that the interface between the FMC card and Peripheral Module is functional.

Mechanical and electrical testing of the two modules within the 1062Q chassis concluded that both modules meets PXIe specifications and thus fulfilling a hardware objective for this thesis. In addition, the FMC interface on the Peripheral Module is functional and the mated pair did not have any obstruction within the chassis.

7.2 FPGA Design Evaluation

The firmware development for both System Controller and Peripheral Module is discussed in the previous Section. In this section, their respective firmware is evaluated based on their routing implementation, resource utilisation and functionality through simulations. These areas of evaluation determine how the FPGA will behave before testing in hardware along with providing information of potential improvements that could be made to the FPGA firmware.

Figure 7.1: Developed System Controller (1) and Peripheral Module (2) in PXIe-1062Q chassis

7.2.1 Static Timing and Routing Evaluation

After developing the FPGA firmware, the next step is to place and route the logic onto the FPGA resources. This occurs in Vivado's implementation stage and is done automatically. Although manual routing of the FPGA logic is possible in Vivado, the time it would take is not feasible for the thesis.

Timing is an important parameter that can be used to evaluate the implemented FPGA design. In digital systems, timing needs to be considered when designing FPGA firmware. When timing is violated, metastability may be introduced into the system, which can cause the FPGA to exhibit strange behaviour in the logic and thus hinder functionality. Xilinx's Vivado suite has provided tools to evaluate timing which can be used for routing evaluation of the FPGA designs for both the System Controller and Peripheral Module.

In a synchronous digital system, data is advanced through a circuit

every tick of the clock signal. This is enforced by flip-flops, which moves the input to its output when instructed by the clock. Two key parameters that determines timing violations are setup and hold slack. The setup time is the amount of time an input to a flip-flop needs to be stable before the clock's active edge. The setup slack is the difference between the desired setup time and the actual setup time. If the setup slack is positive, there is enough time for the data to be captured by the flip-flop. To meet timing requirements, the setup slack should be positive. If the value is negative, the data misses the time to be captured and must wait for the next clock cycle. The hold slack is similar, it is the difference between the desired hold time and the actual hold time, where the hold time refers to the minimum amount of time required for an input to be stable after the clock's active edge. Again, a positive hold slack is required to meet timing. A negative hold slack indicates that the input changed too soon for capture.

System Controller's FPGA Firmware Timing and Routing Evaluation

For each input of a flip-flop within the System Controller FPGA design, a Worst Setup Slack (WSS) and Worst Hold Slack (WHS) is derived within Vivado's implementation stage. To meet the timing constraints of the design, WSS and WHS must be equal or greater than 0 ns for reasons mentioned previously. In the System Controller's FPGA firmware, Vivado reported a WSS and WHS of -1.459 and 0.019 ns respectively. Using the timing analysis tool, ten logic paths are discovered that did not meet the setup timing requirement.

The affected routing paths are shown in Figure 7.2. Two reasons that could result in a negative WSS is clock domain crossing (CDC) and routing congestion. CDC is traversal of a signal from one clock domain to another. Traversal between clock domains requires either a two-stage flip-flop in the destination domain or the use of a FIFO to remove metastability and minimising setup time. However, this is not the case for the System Controller's FPGA firmware as there is only one clock domain in the overall

design.

As CDC is ruled out, it leaves routing congestion as the culprit. Figure 7.3 illustrates the affected paths. The wide fan-out is usually attributed to the routing implementation navigating around areas of congestion. These areas of congestion can be highlighted as illustrated in Figure 7.4, where the shaded areas are FPGA resources that has a utilisation greater than 70%. This leads to more compact routing from Vivado's routing implementation and thus congestion. This correlates to the location of the affected paths shown in the previous figure. Although optimising routing can be reduced in HDL code, Figure 7.3 indicated that these affected paths are within the AXI-PCIe block. As the AXI-PCIe logic is a closed IP core, further RTL optimisation cannot be done. However, testing of the firmware on the developed System Controller (detailed later) confirmed that PCIe connectivity is still functional.

To test if the congestion is the factor for the negative WNS, the System Controller FPGA firmware is implemented on a Zynq 7045 device which has four times as much logic resources compared to the 7015. With more resources, there will be less compact routing and therefore, less congestion. Depicted in Figure 7.5 is the routing implementation on the 7045. The arrows represent the failed timing paths on the System Controller FPGA firmware, which are implemented on the Zynq 7045. It illustrates that these paths are less fanned out as depicted before. After running a timing analysis, Vivado reported a WNS and WHS of 0.28 and 0.03 ns, meeting timing requirements. As the 7045 did not have significant congestion around the AXI-PCIe RTL logic, it can be concluded that the 7015 smaller logic footprint resulted in routing congestion, leading to WNS not being met.

Peripheral Module's FPGA Firmware Timing and Routing Evaluation

Timing analysis is also performed for the Peripheral Module's FPGA firmware. The result of the implementation stage reported a WSS and WHS time of

Figure 7.2: System Controller's FPGA paths that did not meet setup time

Figure 7.3: Paths that did not meet setup time in AXI-PCIe RTL schematic

Figure 7.4: Areas of congestion with utilisation greater than 70%

0.927 and 0.029 ns respectively. As these values are positive, the Peripheral Module's FPGA firmware meets timing constraints. Analysis and evaluation of the FPGA firmware indicated that the System Controller has significant congestion which results in timing violations being reported in Vivado. Although the Peripheral Module did not report any timing Violations, congestion is present on the Artix-7 100T model which may be due to its smaller floor size. A possible remedy for the congestion is to use devices in the series with larger FPGA fabric, such as the Artix 200T model.

7.2.2 Resource Utilisation

Although resource minimisation is not an objective, it affects user development within the FPGA fabric. With more FPGA space available after core functions are implemented, the more resources are available to the

Figure 7.5: System Controller's FPGA paths that failed timing represented on Zynq 7045

user to develop their own FPGA applications. Resource utilisation is more significant on the Peripheral Module as the FPGA fabric is the main development environment. Whereas on the System Controller the main development environment resides in the Linux OS. However, users will still be able to develop on the FPGA fabric of the System Controller. This section discusses the resource utilisation that implements the core FPGA logic for basic PXIe functionality.

7.2.2.1 System Controller's FPGA Firmware Resource Utilisation

The fully routed FPGA firmware of both System Controller and Peripheral Module consumed the resources shown in Table 7.1. The System Controller is more aimed towards users developing on the Linux OS rather than the FPGA fabric. However, there is minimal resource utilisation as portrayed in Table 7.1, providing flexibility for potential FPGA development on the System Controller.

Table 7.1: System Controller top 3 FPGA firmware resource utilisation on the Xilinx Zynq 7z015clg485-1

Resource	Used	Available	Utilisation (%)
Slice registers	17936	92400	19.41
Slice LUTs	20349	46200	44.05
Block RAM	19	95	20.00
LUT Flip Flop Pairs	22057	46200	47.74

Table 7.2: Peripheral Module top 3 FPGA firmware resource utilisation on the Xilinx Artix-7 7a100tfgg484-2

Resource	Used	Available	Utilisation (%)
Slice registers	29932	126800	23.61
Slice LUTs	25727	63400	40.58
Block RAM	31	135	22.96
LUT Flip Flop Pairs	13105	63400	20.67

7.2.2.2 Peripheral Module's FPGA Firmware Resource Utilisation

Unlike the System Controller, the FPGA resource utilisation is an important factor on the Peripheral Module. After implementing the PCIe connectivity logic, the remaining FPGA resources can be utilised by the user to develop their own FPGA applications. The resource utilisation of a peripheral module with PCIe connectivity shown in Table 7.2 is minimal, with each of these resources being less than 50 %. The minimal impact on the FPGA resources after implementing the core functionality of the Peripheral Module provides a large leeway for users to utilise its fabric.

It is shown that FPGA resource utilisation are less than 50% across or resource types. Although the primary development environment resides within the Linux OS on the System Controller, there is a large leeway for user FPGA application if required. A similar resource utilisation is found on the Peripheral Module which lead to the same conclusion. However, the versatile hardware design of the Peripheral Module is compatible with

any 4x5 cm boards from Trenz Electronics due to their pin-compatible nature. If users wish to opt for a larger FPGA package, Trenz Electronics has an 4x5 cm Artix-7 200T module [81] which has twice as many resources compared to the 100T module used in this project.

Overall, both modules' FPGA firmware implements their core functionality to meet their respective objectives. After implementation, resource utilisation impact on the FPGA fabric is shown to be minimal, allowing user FPGA applications to be developed on both modules. With each FPGA firmware evaluated for its timing and resource utilisation, their PCIe connectivity are simulated in Vivado to verify functionality. This is discussed in the following section.

7.2.3 Functional Verification

Initial testing of both PXIe modules' PCIe connectivity are conducted within Vivado's simulator environment. Two test-benches are developed for both PXIe modules that implements logic to monitor the operation of both systems. Each test-bench utilises an AXI-PCIe block example design [19] in either root-port or endpoint configuration as discussed in Chapter 6. Along with the AXI-PCIe block, a root-port and endpoint model from Xilinx [19] are also utilised to test the AXI-PCIe in either configuration. Verification of each test-bench model consists of memory read and writes transactions through a virtual PCIe generation 2 x4 link which is also provided by Xilinx as part of the root-port and endpoint models.

System Controller's FPGA Firmware Functional Verification

Illustrated in Figure 7.6 is the test-bench architecture for the System Controller. The simulation process is a loop-back test which consists of an AXI master initiating first a memory write and then a memory read to the stored address on the endpoint. To monitor PCIe transactions on the System Controller as if it was in a physical system, it consists of an AXI-PCIe

block with the same configuration found in the System Controller FPGA firmware. The AXI-PCIe is connected to an endpoint model through a PCIe generation 2 x4 link. The simulation process consists of data transactions between the AXI-PCIe root-port and the endpoint model.

Figure 7.6: Testbench for AXI-PCIe in root-port configuration

Initiating a data transfer from the root-port to the endpoint involves an AXI master generating traffic for the AXI-PCIe to process through its slave port. The value `0xa4a4a4a4` is written to the *wdata* slave input while asserting high on *wvalid* to validate the written data. The AXI-PCIe block wraps this data into a TLP and sends it to the endpoint through the PCIe link. On-board the endpoint model, the data is extracted from the TLP and stored in a register *tdata*. The received data on the endpoint is verified to be the sent data as illustrated in Figure 7.7. This confirms the memory write functionality of the AXI-PCIe in root-port configuration sending TLPs to a PCIe endpoint. Figure 7.8 illustrates the PCIe transceivers toggling during the transaction.

After the memory write transaction, the AXI master then initiates a memory read transaction to the endpoint. The value `0xa4a4a4a4` is wrapped in a TLP on the endpoint and sent through the PCIe link. The AXI-PCIe de-wraps the TLP and stores the received data in the *READ_DATA* register. Figure 7.9 verifies the received data on the root-port to be the sent data from the endpoint. This completes the loop-back test and shows the

Figure 7.8: PCIe transceivers toggling during transmission

correct functionality of the AXI-PCIe block in root-port configuration.

Peripheral Module's FPGA Firmware Functional Verification

A separate test-bench is also developed for the AXI-PCIe configured as an endpoint device which represents the PXIe Peripheral Module FPGA firmware design. The AXI-PCIe block is connected to a root-port model [19] from Xilinx which functions as the System Controller through a PCIe generation x4 link. The model contains a TX and RX engine for transmission and reception of TLPs, and a test program which stimulates the AXI-PCIe endpoint. Unlike the System Controller's test-bench, the AXI-PCIe needs to be configured before transactions can take place over the PCIe link. During the simulation, the root-port model keeps a log of events that occurs. The simulation consists of:

- The system reset gets de-asserted to allow the AXI-PCIe endpoint to start up and take on the root-port reference clock.
- Configuration transactions performed by root-port to enumerate and configure endpoint
- Separate memory write/read to/from the AXI-PCIe endpoint

At the $4.995\ \mu\text{s}$ time stamp, the system reset is de-asserted to allow the endpoint to start up and lock with the root-port reference clock as illustrated in Figure 7.10. Enumerating and configuring the endpoint consists of multiple memory read and write transactions to the endpoint's BAR. An example of this transaction is reading the endpoint device and vendor ID (*7025* and *10ee* respectively) which is illustrated in Figure 7.11.

Once configuration is successful, transactions can take place over the virtual PCIe link. This consists of a separate memory read and the write transaction from the root-port model to the AXI-PCIe endpoint. Figure 7.12 and 7.13 illustrates the endpoint device successfully receiving and

Figure 7.11: Reading Device and Vendor ID from the endpoint's BAR on sixth row of waveform diagram

transmitting the data payload respectively. Along with the successful synchronisation with a root-port reference clock and configuration of the AXI-PCIe block, the test-bench for the Peripheral Endpoint indicates the successful functionality of its PCIe connectivity.

These simulations conducted verified the AXI-PCIe block in both root-port and endpoint configuration. As the AXI-PCIe block are utilised in both proposed modules FPGA firmware, the next step is to test their functionality in a prototype system which is discussed in the next section.

7.3 Testing of an ADC Firmware using Prototype System

Part of the Peripheral Module FPGA firmware design consists of interfacing with an AD9467 ADC. With functional I/O stipulated as an objective for the Peripheral Module, the ADC functionality must be evaluated. A test set-up is built that comprises of a 20 MHz EZ Digital FG-7020A function generator, connected to the *analog signal in* SMA connector of the AD9467 FMC card which resides on the Peripheral Module.

The testing procedure consists of acquiring signals over a frequency range of between 100 KHz and 20 MHz is chosen, which evaluates the performance of the ADC, while being within the operating range of the function generator and well under the Nyquist frequency of $250/2 = 125\text{MHz}$. This ensures that the sampled waveform meets the Nyquist-Shannon sampling theorem [82], which states for a perfect reconstruction of an analogue waveform, the sampling rate should be at least twice the rate of the signal rate.

The sampled data is obtained through a UART interface which can then be plotted and analysed. Figure 7.14 illustrates a 100 kHz sine wave that is captured by the AD9467 ADC. Knowing the sample rate of the ADC, multiple maximum or minimum points of a sine wave are chosen to calculate

Figure 7.12: Verification of root-port sending 0x61626364 to AXI-PCIe endpoint as part of memory read transaction

Figure 7.13: Verification of root-port receiving 0x61626364 from AXI-PCIe endpoint memory write transaction

Δt , as depicted in Figure 7.14. The frequency can therefore be calculated through as a quick check to verify correct operation.

Figure 7.14: 100 KHz sine wave capture with an AD9467 ADC consisting of 2048 data points

This test proves the functionality of the AD9467 FPGA design and furthermore that I/O functionality is established. The following section incorporates the AD9467 FPGA design with the PCIe-DMA engine on the endpoint to allow data from the ADC stored on DDR to be sent through PCIe in the prototype system.

7.4 Driver and API Verification

Discussed in Chapter 4, a Linux image with PCIe command line tools is developed to verify the PCIe connectivity in a prototype system. The same procedure is carried out to verify the developed driver and API on the developed PXIe modules. In addition, an example user application is developed which is tested on the prototype system.

7.4.1 Loading of the PCIe driver on a Linux OS

The pre-configured Linux image contains the developed PCIe driver. Through the UART terminal, the current directory is navigated to the location of the compiled drivers. This is stored in the `/lib/modules/3.17.0-xilinx/extra` directory. The compiled driver is an object file called `xpcie.ko`. To load the driver, the Linux command `insmod` is called. This starts the registration, allocation and initialisation process as discussed in Section 6.7. Once the process is complete, the System Controller can send memory reads and writes to the Peripheral PCIe endpoint. The result of loading the PCIe driver is illustrated in Figure 7.15

```
root@peta_pcie_rc:~# insmod /lib/modules/3.17.0-xilinx/extra/xpcie.ko
Registering pci driver testing
xpcie 0000:01:00.0: enabling device (0140 -> 0142)
xpcie 0000:01:00.0: BAR[0] mapped at 0x80960000 with length 65536.
xpcie 0000:01:00.0: Failed to enable MSI interrupts
xpcie 0000:01:00.0: Setting up xilinx axi cdma engine...Successful
Registration complete
```

Figure 7.15: Result of loading PCIe driver on Linux OS

The Peripheral Module is given a bus number, a device number and a function number portrayed as `0000:01:00.0`. After loading the drivers, the BAR0 address of the Peripheral Module is mapped into the kernel memory map. As discussed in the previous chapter, the BAR0 to AXI space allows the System Controller to access the Peripheral Module's AXI address map. The System Controller can access the DMA controller and perform memory reads and writes directly to the endpoint's DDR memory. As shown, the driver implements MSI interrupts, but it is not operational in the Xilinx's Linux kernel version 3.17.0. However, a future update of the kernel may solve the issue. Until this happens, legacy interrupts are used in its place.

Like in the prototype system discussed in Chapter 4, the command `lspci` is executed to verify the Peripheral Module PCIe configuration. Illustrated in 7.16, the Peripheral Module is allocated with the entry `000:01:00.0` by the


```

01:00.0 Memory controller: Xilinx Corporation Device 7024
Subsystem: Xilinx Corporation Device 0007
Flags: fast devsel, IRQ 167
Memory at 60800000 (64-bit, prefetchable) [disabled] [size=64K]
Capabilities: [40] Power Management version 3
Capabilities: [48] MSI: Enable- Count=1/2 Maskable- 64bit+
Capabilities: [60] Express Endpoint, MSI 00
Capabilities: [100] Device Serial Number 00-00-00-00-00-00-00-00

```

Figure 7.16: Peripheral Module endpoint *lspci* output from Linux command line

kernel on the System Controller. The output of *lspci* indicates that the endpoint is a Xilinx device serving as a “Memory Controller” device, with *BAR0* memory mapped to *0x60000000* within the kernel memory map. This is then followed by a list of PCIe capabilities the endpoint supports, along with the kernel driver it is using. As illustrated, the Peripheral Module successfully utilises the developed *xpcie* drivers.

In addition to the developed Peripheral Module, the System Controller PCIe connectivity is also tested with a commercial PXIe-6535 I/O module from National Instruments [83]. Through the *lspci* command tool, the PXIe-6535 can be seen as illustrated in Figure 7.17. It is classified as a *Signal processing controller*, with one BAR being mapped to *0x60400000* in the kernel address space.

```

09:00.0 Signal processing controller: National Instruments PXIe/PCIe Device
Subsystem: National Instruments PXIe-6535
Flags: fast devsel, IRQ 167
Memory at 60400000 (32-bit, non-prefetchable) [disabled] [size=64K]
Capabilities: [40] Power Management version 2
Capabilities: [48] MSI: Enable- Count=1/1 Maskable- 64bit+
Capabilities: [58] Express Endpoint, MSI 00
Capabilities: [100] Device Serial Number 00-80-2f-00-00-a8-58-1a

```

Figure 7.17: PXIe-6535 *lspci* output

The result of this test verifies the PCIe connectivity of both FPGA firmware within the PXIe chassis. From this test, the developed *xpcie* drivers can verify the Peripheral Module. Data transactions between the two modules is discussed in Section 7.4.2.

```

printf("Starting AD9467 capture\n");
w_dw = 0x1;
address = 0x100000;
write_data(fd, (unsigned char *)&w_dw, address, 4);

printf("Wait until capture is done\n");
while(1){
    read_data(fd, (unsigned char *)&r_dw, address, 4);
    if(r_dw == 0){
        printf("AD9467 capture complete\n");
        break;
    }
}

printf("Retrieving AD9467 captured data\n");
for(i=0; i<1024; i+=8){
    address = 0x8000000 + i;
    read_data(fd, (unsigned char *)&r_dw, address, 4);
    printf("Read from address 0x%x: 0x%x\n", address, r_dw);
}

```

Listing 7.1: Example user application utilising API

7.4.2 Example User Application

An example user application is developed for the prototype system that encompasses the firmware design of both boards as shown in Listing 7.1. The aim of the user application is to show all aspects of the FPGA and Software firmware design of both modules. The result of this application verifies all hardware and firmware objectives set out in Chapters 5 and 6. The application uses the set-up depicted in Figure 4.3 with an AD9467 FMC card connected to the Peripheral Module.

The user application is a C program that is configured into the Linux kernel on the System Controller. The application leverage on the developed API to remove the complexity of the user application. To initiate a capture of the AD9467, the *write_data* command is used to write *0b1* to the peripheral module DDR memory, at the address *0x100000*.

The Microblaze firmware on the Peripheral Module runs a polling routine which continuously checks the address `0x100000` as illustrated in Figure 7.18. When a `0b1` appears after being sent from the System Controller, the Microblaze can start the AD9467 capture process through a SPI command. This is implemented by using Analog Devices provided AD9467 drivers. By default, ADC captures are 1024 bits in size, but can be further increased or decreased. However, precautions are taken when increasing the capture size to prevent overwriting the control address at `0x100000`. This is a limitation of the example user application. However, it can be mitigated by setting up a register primitive in FPGA fabric instead of using DDR memory. Through Analog Devices's DMA core, the sampled data is directly stored at base address `0x0` without any intervention from the Microblaze. After the capture process, the Microblaze resets the value in address `0x100000` to `0b0`.

While an analogue signal is being captured on the Peripheral Module, the System Controller also polls the Peripheral Module's DDR address `0x100000`. When it sees a value of `0b1`, the System Controller knows the data capture process has been completed. The next stage is for the data to be retrieved by the system controller. Using the `read_data` from the API, each sample can be read from the endpoint's DDR memory in 16 bit increments. The overall System Controller polling routine is illustrated in Figure 7.19.

Discussed in Section 6.8, the API implements *ioctl* commands which can be utilised to measure PCIe transfer speeds. A log of data transfers is obtained with an average read and write speed of 100 MB/s. As the FPGA firmware for both System Controller and Peripheral Module implements PCIe x4 generation 2, the average throughput is expected to be around 2.0 GB/s as per the PCIe version 2.0 specifications. This low throughput may be due to the large overhead on the driver design. Along with PCIe protocol overhead, throughput is reduced as a result. A future solution may consist of optimising with the current driver design or to leverage on

Figure 7.18: Peripheral Module polling routine

proven open-source PCIe drivers for the AXI-PCIe block. This is discussed as part of future works in Section 8.2.

This example user application verifies the FPGA firmware on both modules as well as the driver and API software. From this application, it can be shown that transfer of sampled data from the Peripheral Module can be sent to the System Controller through PCIe. Users can extend this simple demo with their own custom applications written for the System Controller.

Figure 7.19: System Controller capture-polling routine

7.5 Overall Cost

Discussed in Chapter 2, the commercial PXIe counterparts for the System Controller and Peripheral Module are the PXIe-8100 and SPEXI, with a cost of NZD \$1795 and NZD \$1000 respectively. These modules provides a relevant cost benchmark for the two developed modules of the thesis. The cost objective for both developed PXIe modules is proposed in Chapter 5. They are required to be more economical than their counterparts, making the basic PXIe platform not being prohibitively expensive. The cost for the developed System Controller and the Peripheral Module is summarised in Table 7.3. For each board, it consists of developing an eight-layer PCB, the cost of electronic components and stencils for SMD component soldering. All of these are necessary expenditure to fully develop a functional PXIe

Table 7.3: Component cost breakdown for developed System Controller and Peripheral Module

Item	System Controller	Peripheral Module
Materials	\$577.38	\$243.40
PCB	\$159.39	\$26.99
Labour	\$132.62	\$47.52
FPGA/SoC Module	\$364.26	\$387.26
Total (NZD)	\$1233.65	\$705.17

module.

The total cost for the System Controller and Peripheral Module is NZD \$1233.65 and NZD \$705.17 respectively. These prices include their respective FPGA/SoM solution which is needed for a fully functional PXIe module. In comparison to their existing solution counterpart, the developed System Controller is 31% cheaper than the PXIe-8100 system controller from National Instruments, while the developed Peripheral Module is 29% cheaper than the SPEXI. Although further reductions could be made by using different components, the current cost of each modules satisfies their respective cost objective presented in Section 5. With a developed open platform, the biggest gains will be made when comparing to commercial PXIe peripheral modules. The price range of these modules starts at from NZD \$2000 [84] to NZD \$53000 [85]. An open framework of the developed platform allows users to develop their own custom solutions without the need of purchasing high-end PXIe modules. As a result, the developed open PXIe instrumentation platform can be used to facilitate custom hardware and software development for high-speed applications with far less set-up cost compared to commercial solutions.

7.6 Summary

Review of both developed PXIe modules along with their associated firmware, all objectives have been met. The cost objectives for both modules are

exceeded, being significantly less expensive than their existing solution counter parts. Although data transfer between the two modules is achieved, the measured throughput is less than anticipated when comparing to PCIe Generation 2 specifications. The throughput can be increased through either optimisation of the developed driver or utilising an open-source PCIe driver with proven expected throughput. With a functional open PXIe platform built, its current state allows users to utilise high-speed instrumentation for a range of projects and experiments, at a fraction of the cost of commercial PXIe solutions.

Chapter 8

Conclusions

This chapter reviews the work completed on the developed PXIe System Controller and Peripheral Module, provides suggestions for future work and a review of the project with reference to the hardware and firmware objectives outlined in Chapters 5 and 6.

8.1 Review

Chapter 1 and 2 outlined the need for an open and customisable PXIe platform. The project focused on developing a basic, but functional PXIe platform consisting of a System Controller and a Peripheral Module. Built on the PCIe bus standard, it provides high-speed transfers which is suitable for a wide range of applications including MRI. Coupled with FPGA technology, an open and customisable PXIe implementation is developed. In Chapter 2, a review of existing commercial PXIe-FPGA solutions are shown to be prohibitively expensive for low-end users. There exists an open solution called the SPEXI from CERN that has a more suitable price tag of NZD \$1000. However, it uses an obsolete Spartan 6 FPGA which is not compatible with the latest Vivado environment. As Vivado offers useful IP-cores that implement PCIe and DMA functionality, using either a Series-7 FPGA or Zynq SoC is a must for this project.

It was discovered that there exist modular FPGA and SoC solutions from Avnet and Trenz Electronics. A PicoZed Zynq 7015 SoC module from Avnet is utilised for the System Controller module. A Zynq solution is chosen to later simplify Linux OS development while being compatible with Vivado. For the Peripheral Module, A TE0712 Artix-7 FPGA module from Trenz Electronics is chosen as it provided a customisable solution for users to develop RTL on fabric. They are utilised to simplify both PXIe modules hardware implementation. Before starting hardware development, the PCIe topology in the PXIe standard is verified on two development boards in Chapter 4. With successful PCIe connectivity, Chapter 5 discusses the hardware implementation of the Peripheral Module. Firstly, a skeleton PCB is used to meet the mechanical constraints of the PXIe-1062Q. This is then followed by implementing the required board-to-board connectors along with the backplane's interface.

Further hardware development consisted of routing high-speed signals, development of a clock system and implementing peripherals. High-speed signals consists of PCIe lines and LVDS signals. These signals are implemented as differential signals and are required to meet a characteristic impedance of $100\ \Omega$. After routing of critical signals and implementing various peripherals, robust power circuitry was developed for the Peripheral Module with the aid of the XPE tool.

After completion of the hardware fabrication of both PXIe modules, Chapter 5 outlines each PXIe modules respective FPGA, software and firmware design. Developing the firmware after developing the hardware solidifies pin constraints for the FPGA firmware. For the System Controller, its firmware consists of two parts. Firstly, a PCIe root-port implementation is configured in the Zynq's FPGA fabric. This allows memory reads and writes to be sent to the Peripheral Module. An AXI-PCIe IP-core from Xilinx is utilised to achieve this. Secondly, a Linux OS is developed to meet the System Controller's objectives. This provides an environment for the user to develop applications to interface with the Peripheral Module in a

PXIe system. In addition, an associated PCIe driver and API are developed to aid in user development within the Linux OS.

The Peripheral Module firmware only consists of FPGA logic that implements a PCIe endpoint. It consists of two main blocks, an AXI-PCIe interface and a DMA controller. The AXI-PCIe soft-core is configured as an endpoint. To handle PCIe to AXI translation, a single BAR is implemented with a translation of `0x81000000`. This allows the System Controller to directly access the DMA controller on-board the Peripheral Module and perform memory reads and writes to its DDR memory. After implementing the Peripheral Module's PCIe-DMA engine, there are left-over resources users could use for custom FPGA applications. As the TE0712 is board compatible with other 4x5 cm boards from Trenz Electronics, users have the choice of swapping out the TE0712 for a larger FPGA package.

Discussed in Chapter 6 are the evaluation of the developed modules' hardware design. All hardware objectives for both modules laid out in Chapter 2 are either met or exceeded. The developed modules are compatible and fully functional within a 1062Q chassis, and the PCIe connectivity is proven in a later example user application. The cost objective is introduced in Chapter 2 and has been met. Additionally, the total cost of each module is more economical than their existing counter part by a significant margin. The System Controller is 31% cheaper than the PXIe-8100 system controller from National Instruments, while the Peripheral Module is 29% cheaper than the SPEXI from CERN.

Also discussed in Chapter 6 are the evaluation of the modules' respective FPGA and software firmware. The firmware objectives for each module are laid out in Chapter 4. Each firmware objectives are met. Evaluation of the FPGA firmware consist of static timing and routing evaluation, resource utilisation and functional verification. Firstly, timing is met on the Peripheral Module's. However, on the System Controller, the set up slack is negative due to large congestion near the PCIe hard block in the fabric. Implementing the same firmware on the larger Zynq 7045 device solved

the set up slack timing constraint due to less congestion. Later testing of the System Controller's firmware resulted in successful PCIe transfers, which removes the need to improve the set up slack timing. Secondly, resource utilisation are analysed for both modules. This is an important factor as it determines how much resources are left after implementing core functionality, reflecting the versatility nature of the two modules. Resource utilisation for both firmware designs are less than 50%, which therefore allows for additional RTL development. Lastly, verification of the FPGA firmware consists of utilising Xilinx's PCIe root-port and end-point models to interface with the AXI-PCIe block. Functionality is proven within Vivado's simulator, consisting of memory reads and writes.

An example user application is developed to encompass the overall project implementation. The set-up consists of the developed modules placed within the 1062Q chassis and an AD9467 FMC card mated with the Peripheral Module's FMC connector. After successful testing on a prototype system, the application is tested on the developed boards. On the System Controller, an example program is developed that initiates an ADC capture on the Peripheral Module. Once the capture process is completed, data is transferred from the Peripheral Module's DDR memory to the System Controller and is displayed through a UART terminal. The example user application verifies all the hardware and software objectives laid out in Section 5.2 and 6.1. With read and writes of speeds of 100 MB/s, the PCIe x4 generation 2 link implemented is not fully saturated due to overhead. Further optimisation could be done to remove the driver overhead or, another solution is to utilise proven open-source PCIe drivers.

8.2 Future Work

With a developed open PXIe platform, it currently uses a commercial PXIe-1062Q chassis from National Instruments. To fully complete the open platform, a suitable future work is to develop an open PXIe chassis. As the

PXIE standard is open, the development of a custom chassis is achievable. The potential chassis will need to implement the correct male backplane connectors for either a system controller or peripheral module, along with PCIe links from the system controller to each slot of the chassis and implementing the power rails.

As it currently stands, both developed PXIE modules are mechanically compatible with the 1062Q chassis. However, a significant amount of force is required to pull each module from their respective slot. Found on commercial solutions are front-mounted levers which can be utilised in a future design to make removal of the developed modules less forceful.

An important aspect of the PXIE standard is that it provides synchronisation features which can be applied to a range of projects. The developed System Controller and Peripheral Module allows for external inputs for synchronisation. Their respective clock systems allow signals from a timing module to be utilised by either the Artix FPGA or Zynq SoC. A custom timing module can be developed as part of future work to fully complete the open PXIE platform along with a custom chassis. The potential timing module would have high-quality clock sources and trigger lines running through an external SMA output and the DSTAR lines on the backplane. A potential use-case for a timing module is to provide a trigger to perform data acquired DMA transfers. Currently, data acquisition and DMA transfers are instigated from the System Controller by providing write commands to the Peripheral Module. However, a timing module can be used instead and trigger the Peripheral Module to perform a acquisition and PCIe transfer to the System Controller. This removes the need for the System Controller to constantly poll a specific address on the Peripheral Module as discussed in Chapter 6. By removing the polling nature, it frees up more resources on the ARM processor which can be used for other tasks.

Demonstrated in Chapter 5, the developed PCIe driver is tested in an example user application. Utilising ioctl commands from the API, trans-

fer speeds during reads and writes are measured to be 100 MB/s which did not meet PCIe x4 generation 2 expected transfer speed. Although the driver code could be further optimised, open-source drivers such as the Reusable Integration Framework for FPGA Accelerators (RIFFA) [86] is an alternate solution that could be incorporated into the open PXIe platform. This solution offers proven 80% saturation of PCIe links along with an API which potentially allows users to communicate to the Peripheral Module using either C/C++, Python and Java programming languages. This solution offers a more versatile software development environment compared to the currently implemented PCIe driver and API.

8.3 Summary

The objectives set out in Section 5.2 and 6.1 have been met or exceeded. With a total cost of the System Controller and Peripheral Module NZD being \$1233.65 and \$705.17 respectively, the developed modules are more economical than their existing solution counter parts. It provides a functional, economical alternative to commercial PXIe solutions, while being open for custom hardware and software development.

Both developed modules are mechanically and electrically compatible with the 1062Q chassis from evaluation. As a result, PCIe data transfers within the chassis can be initiated from the Linux OS on the System Controller. The result verifies all the objectives relating to the System Controller. However, potential driver optimisation or porting of an open-source driver can be done as future work to increase throughput. In addition, the Peripheral Module is tested with an AD9467 FMC card in an example user application which verifies the I/O capabilities and PCIe connectivity on the Peripheral Module. Furthermore, a usable API is developed with the aim to simplify user development. The hardware schematic, FPGA firmware and PCIe driver is available on a Github repository [87].

In summary, the open PXIe platform illustrated in Figure 8.1 consists

of a low cost and functional System Controller and Peripheral Module. Although further improvements could be done, all objectives have been met or exceeded, allowing users to utilise a high-speed instrumentation platform to develop their own PXIe hardware and software applications for a range of applications.

Figure 8.1: open PXIe platform with developed System Controller, Peripheral Module and PXIe-1062Q chassis from National Instruments [3]

Bibliography

- [1] (2018, February) CT-400 Modular 13-Slot VXIBIS Mainframe. VTI Instruments. [Online]. Available: <http://www.vtiinstruments.com/Products-Services/VXI/CT-400.aspx>
- [2] N. PXIe, "1062q 8 slot, 3u pxi express chassis with ac-up to 3 gb/s hardware platform," *National Instruments Corp., Vaudreuil-Dorion, Quebec, Canada*.
- [3] Mliu92. (2015) Example pci express topology. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/1/1c/Example_PCI_Express_Topology.svg
- [4] R. Budruk, D. Anderson, and T. Shanley, *PCI express system architecture*. Addison-Wesley Professional, 2004.
- [5] "Express hardware specification, august 22, 2005 http://www.pxisa.org/userfiles/files/specifications/pxiexpress.hw_spec_r1," *PDF Last accessed April*, vol. 24, 2015.
- [6] P. Psurek. (2015) Samtec fmc hpc male and female connector. [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/d/d4/Samtec_ASP-134488-01_male_FMC_HPC_connector_PP008729.jpg
- [7] "PXIe-8821," 2018. [Online]. Available: <http://www.ni.com/en-nz/support/model.pxie-7961.html>

- [8] "Simple PXIe FMC Carrier Board SPEXI," 2018. [Online]. Available: <https://www.ohwr.org/projects/spexi/wiki>
- [9] "PXIe700," 2018. [Online]. Available: <http://www.sundance.technology/som-carriers/pxi-boards/pxie700/>
- [10] M. Bourne, R. Dykstra, and S. Obruchkov, "An open source pxie platform for mri instrumentation development," in *2017 International Society for Magnetic Resonance in Medicine (ISMRM)*, April 2017.
- [11] I. Avnet, "PicoZed," Avenet, Inc., April 2017. [Online]. Available: <http://zedboard.org/product/picozed>
- [12] "Picozed hardware user guide," Avnet, 2017. [Online]. Available: <http://zedboard.org/sites/default/files/documentations/5279-UG-PicoZed-7015-7030-V1.7.pdf>
- [13] I. Avnet, "Zynq mini-ITX," Avenet, Inc., April 2017. [Online]. Available: <http://zedboard.org/product/mini-itx-board>
- [14] —, "PicoZed Carrier Card," Avenet, Inc., April 2017. [Online]. Available: <http://zedboard.org/product/picozed-fmc-carrier-card-v2>
- [15] "TE0712 technical reference manual," Trenz Electronics, 2017. [Online]. Available: <https://wiki.trenz-electronic.de/display/PD/TE0712+TRM>
- [16] "High-Speed Interface Layout Guidelines," 2017. [Online]. Available: <http://www.ti.com/lit/an/spraar7g/spraar7g.pdf>
- [17] "7 Series FPGAs GTP Transceivers," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug482_7Series_GTP_Transceivers.pdf

- [18] "ANSI/VITA 57.1 FPGA Mezzanine Card (FMC) Standard," VITA, 2010.
- [19] "AXI Memory Mapped to PCI Express (PCIe) Gen2 v2.6 LogiCORE IP Product Guide (PG055)," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_pcie/v2_6/pg055-axi-bridge-pcie.pdf
- [20] A. Devices, "Ad9467: 16-bit, 200 msps/250 msps analog-to-digital converter," 2011.
- [21] Xilinx, "7 Series FPGAs SelectIO Resources," 2017.
- [22] M. Bourne, "Development of a PXI Express Peripheral Module and Data Transfer Platform," Master's thesis, Victoria University of Wellington, Wellington 6140, 2013.
- [23] L. Hong, "Development and application of the pxi technology [j]," *Measurement & Control Technology*, vol. 6, p. 015, 2006.
- [24] (2010, November) Short Tutorial on VXI. National Instruments. [Online]. Available: <http://www.ni.com/white-paper/2899/en>
- [25] (2018, February) PXISA Sponsor Members. PXISA. [Online]. Available: <http://www.pxisa.org/Members/Roster.aspx>
- [26] L. D. Specification, "Rev. 1.4 (2011)."
- [27] W. Stallings, *Computer organization and architecture: designing for performance*. Pearson Education India, 2000.
- [28] C. Young, "The new penguin dictionary of electronics," 1979.
- [29] T. Kilburn, R. B. Payne, and D. J. Howarth, "The atlas supervisor," in *Proceedings of the December 12-14, 1961, eastern joint computer conference: computers-key to total systems control*. ACM, 1961, pp. 279–294.

- [30] L. Torvalds, "Linux history," *Linux International*, 1999.
- [31] I. M. Missing Link Electronics, "Zynq storage extension," 2014.
- [32] V. Technologies, "VAB-820," Via Technologies, November 2016. [Online]. Available: <https://www.viatech.com/en/boards/pico-itx/vab-820/>
- [33] "QUANTUM-iMX6 SBCs," 2018. [Online]. Available: <https://www.diamondsystems.com/files/binaries/QuantumiMX6datasheet.pdf>
- [34] I. Kuon, R. Tessier, and J. Rose, "Fpga architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008.
- [35] "MicroBlaze Linux," 2018. [Online]. Available: <http://www.wiki.xilinx.com/Build+kernel>
- [36] X. Zynq, "7000," *Zynq-7000 all programmable soc overview, advance product specification-ds190 (v1. 2) available on: http://www.xilinx.com/support/documentation-/data sheets/-ds190-Zynq-7000-Overview. pdf, August, 2012.*
- [37] D. Incorporated, "Digilent pmod specification."
- [38] R. Seelam, "I/o design flexibility with the fpga mezzanine card (fmc) wp315," 2009.
- [39] "PXIe Membership Roster," 2018. [Online]. Available: <http://www.pxisa.org/Members/Roster.aspx>
- [40] M. Ruiz, J. Vega, R. Castro, D. Sanz, J. López, G. De Arcas, E. Barrera, J. Nieto, B. Gonçalves, J. Sousa *et al.*, "Iter fast plant system controller prototype based on pxie platform," *Fusion Engineering and Design*, vol. 87, no. 12, pp. 2030–2035, 2012.

- [41] E. Barrera, M. Ruiz, S. López, D. Machon, and J. Vega, "Pxi-based architecture for real-time data acquisition and distributed dynamic data processing," *IEEE transactions on nuclear science*, vol. 53, no. 3, pp. 923–926, 2006.
- [42] Z. Wang, Y. Shang, J. Liu, and X. Wu, "A labview based automatic test system for sieving chips," *Measurement*, vol. 46, no. 1, pp. 402–410, 2013.
- [43] A. G. Volkov, K. Baker, J. C. Foster, J. Clemmons, E. Jovanov, and V. S. Markin, "Circadian variations in biologically closed electrochemical circuits in aloe vera and mimosa pudica," *Bioelectrochemistry*, vol. 81, no. 1, pp. 39–45, 2011.
- [44] A. G. Volkov, M.-R. Pinnock, D. C. Lowe, S. G. MaResha, and V. S. Markin, "Complete hunting cycle of dionaea muscipula: consecutive steps and their electrical properties," *Journal of plant physiology*, vol. 168, no. 2, pp. 109–120, 2011.
- [45] L. U. Manual, "National instruments," *Austin, TX*, 1998.
- [46] "CERN Homepage," 2018. [Online]. Available: <https://home.cern/>
- [47] "Open Hardware Repository," 2018. [Online]. Available: <https://www.ohwr.org/projects>
- [48] "Gennum 4124 PCIe Bridge," 2018. [Online]. Available: <http://downloads.semtech.com/extranet/document/55047>
- [49] "Kintex-7," 2018. [Online]. Available: <https://www.xilinx.com/products/silicon-devices/fpga/kintex-7.html>
- [50] "PXIe-8100," 2018. [Online]. Available: <http://www.ni.com/en-nz/support/model.pxie-8100.html>

- [51] "PXIe-8135," 2018. [Online]. Available: <http://www.ni.com/en-nz/support/model.pxie-8135.html>
- [52] "CompactPCI Express Specification," 2018. [Online]. Available: https://www.picmg.org/wp-content/uploads/PICMG_EXP_0_R2_0.pdf
- [53] "ERNI Electronics," <http://www.erni.com/en/home/>, 2017.
- [54] "24-Lane, 6-Port Express Gen 2 Switch," 2017. [Online]. Available: <https://www.broadcom.com/products/pcie-switches-bridges/pcie-switches/pex8624>
- [55] "DSC557-03 Crystal-less Two Output PCIe Gen1/2/3 Clock Generator," 2018. [Online]. Available: <http://ww1.microchip.com/downloads/en/DeviceDoc/DSC557-03%20Datasheet%20MKQBPD12091701-2.pdf>
- [56] I. Xilinx, "Zynq-7000 all programmable soc technical reference manual-ug585 (v1. 10)."
- [57] "Marvell Alaska 88E1512," 2017. [Online]. Available: https://www.marvell.com/transceivers/assets/Alaska_88E1512-001_product_brief.pdf
- [58] "AN-1469 PHYTER Design and Layout Guide," 2013. [Online]. Available: <http://www.ti.com/lit/an/snla079d/snla079d.pdf>
- [59] "Xilinx Power Estimator," 2017. [Online]. Available: <https://www.xilinx.com/products/technology/power/xpe.html>
- [60] "5-Channel Integrated Power Solution with Quad Buck Regulators and 200 mA LDO Regulator," 2017. [Online]. Available: <http://www.analog.com/media/en/technical-documentation/data-sheets/ADP5052.PDF>

- [61] "Power-supply sequencing for FPGAs," Texas Instruments Inc., 2014. [Online]. Available: <http://www.ti.com/lit/an/slyt598/slyt598.pdf>
- [62] "Product Discontinuation Notification," 2018. [Online]. Available: https://www.mouser.com/PCN/Broadcom.Limited_PDN_06092017_DCSG_03.pdf
- [63] "XAPP1171 PCI Express Endpoint-DMA Initiator Subsystem," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/application_notes/xapp1171-pcie-central-dma-subsystem.pdf
- [64] "Altium PCB Design Software," 2018. [Online]. Available: <http://www.altium.com/>
- [65] T. C. Edwards and M. B. Steer, *Foundations for microstrip circuit design*. John Wiley & Sons, 2016.
- [66] U. S. B. Specification, "Revision 2.0," 2000.
- [67] "Pmod NIC100: Network Interface Controller," 2018. [Online]. Available: <https://store.digilentinc.com/pmod-nic100-network-interface-controller/>
- [68] "Pmod AD1: Two 12-bit A/D Inputs," 2018. [Online]. Available: <https://store.digilentinc.com/pmod-ad1-two-12-bit-a-d-inputs/>
- [69] "Pmod AD2: 4-channel 12-bit A/D Converter," 2018. [Online]. Available: <https://store.digilentinc.com/pmod-ad2-4-channel-12-bit-a-d-converter/>
- [70] "Micropower 800mA Low Noise 'Ceramic Stable' Adjustable Voltage Regulator for 1V to 5V App," 2017. [Online]. Available: <http://www.ti.com/product/LP3878-ADJ>

- [71] "Vivado Design Suite," Xilinx, Inc., 2017. [Online]. Available: <https://www.xilinx.com/products/design-tools/vivado.html>
- [72] "XCN13012: Product Discontinuation Notice for Development Systems Products," Xilinx, Inc., 2013. [Online]. Available: https://www.xilinx.com/support/documentation/customer_notices/xcn13012.pdf
- [73] "Petalinux Tools," Xilinx, Inc., 2017. [Online]. Available: <https://www.xilinx.com/products/design-tools/embedded-software/petalinux-sdk.html>
- [74] O. Salvador and D. Angolini, *Embedded Linux Development with Yocto Project*. Packt Publishing Ltd, 2014.
- [75] "AMBA AXI and ACE Protocol Specification," Arm Holdings, 2011. [Online]. Available: http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf
- [76] "AXI Reference Guide," Xilinx, Inc., 2012. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/axi_ref_guide/v13_4/ug761_axi_reference_guide.pdf
- [77] "Processing System 7 v5.5," Xilinx, Inc., 2017. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/processing_system7/v5_5/pg082-processing-system7.pdf
- [78] "7 Series Gen2 Integrated Block for PCI Express (PCIe)," 2018. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/pcie_7x/v3_3/pg054-7series-pcie.pdf
- [79] "Analog Devices HDL Reference Designs," 2018. [Online]. Available: <https://github.com/analogdevicesinc/hdl>

- [80] “Analog Devices AD9467 Drivers,” 2018. [Online]. Available: <https://github.com/analogdevicesinc/no-OS/tree/master/AD9467-FMC-EBZ>
- [81] “TE0712 200t,” 2018. [Online]. Available: <https://shop.trenz-electronic.de/en/TE0712-02-200-2C-Micromodule-Artix-7-XC7A200T-2C-4x5cm-standard-footprint-con-temp.-range>
- [82] C. E. Shannon, “Communication in the presence of noise,” *Proceedings of the IRE*, vol. 37, no. 1, pp. 10–21, 1949.
- [83] “PXIe-6535,” 2018. [Online]. Available: <http://www.ni.com/en-nz/support/model.pxie-6535.html>
- [84] “PXIe-2567,” 2018. [Online]. Available: <http://www.ni.com/en-nz/support/model.pxie-2567.html>
- [85] “PXIe-5644,” 2018. [Online]. Available: <http://www.ni.com/en-nz/support/model.pxie-5644.html>
- [86] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, “Riffa 2.1: A reusable integration framework for fpga accelerators,” *ACM Transactions on Reconfigurable Technology and Systems (TRETs)*, vol. 8, no. 4, p. 22, 2015.
- [87] “Open PXIe Platform,” 2018. [Online]. Available: https://github.com/VUWinstrumentation/open_PXIe_platform

Appendices

Appendix A

Peripheral Module Hardware Schematics

A

B

C

D

A

B

C

D

Title

Revision

Sheet of

Drawn By:

Size

Number

A4

Date:

File:

5/12/2017

F:\Google Drive\U_FMC_carrier_top.SchDoc

4

JTAG

CLOCK BUFFER

Table 4. Differential Output Buffer Type Selection

CLKoutX_TYPET	CLKoutX_TYPED	CLKoutX_BUFFER TYPE (BANK A or B)
0	0	LVPECL
0	1	LVDS
1	0	HCSL
1	1	Differential (H+Z)

EXTERNAL CLOCK 100M INPUT

DS to SS

PMOD

UART

CLOCK OUT

PMO

ITAG

FMC LPC

FMC_SIGNALS

IIC VOLTAGE TRANSLATION

JTAG CHAIN

Table 4. Truth Table for the ADG819

IN	Switch S1	Switch S2
0	On	Off
1	Off	On

TE0712_TDO

TE0712_JTAG_TDO
TE0712_TDO

TE0712_JTAG_TDO

JTAG_TDI
 JTAG_TDO
 JTAG_TCK
 JTAG_TMS

JTAG_TDI
 JTAG_TDO
 JTAG_TCK
 JTAG_TMS

JTAG

Title		
Size	Number	Revision
A3		
Date	5/12/2017	Sheet of
F:\Goatle Drive - JPMC - Schibbe - Dannon Br		

INPUT POWER

Need ferrite choke and cap. for LP filter. Removes non-DC components. Filter design from TE0701.

BANK VOLTAGE SELECTION

Default bank voltage for I5 and I6 should be 2.5V (for LYDS) and 3.3V for bank I3

Shunt jumper can be found here: <https://www.digkey.com/product-detail/-DA33M0580-ND2-071621>

1.8V POWER

Footprint from Altium vault alternative

From TE0701 Schematic pg17

1.3V POWER

2.5V POWER

STATUS LED

FMC_VADJ CIRCUIT

Only implementing 4 lane PCIe

- Nominal values used, all dimensions are mm.
- Generic footprints used.
- Top layer component height note derived from Cross Section, see Compact PCI P10MG 2.0 R3.0.pdf, Page 47.
- Bottom layer component height assumes no solder side cover and complies with section 4.1.8, see Compact PCI P10MG 2.0 R3.0.pdf, Page 39, 47.
- Backstop is not specified in P10MGEXPRESS_HW_SFPC_M1.PDF nor implemented in this template.
- All other mechanical requirements derived from CompactPCI-Express specification.

Sheet Title PXie PERIPHERAL 3U Top Sheet		Altium	
Project Title PXie PERIPHERAL 3U		1B7 Shuanghui Level 3 - Building 3 No.165 Linying Road Shangbai 200355, China	
Size: A4	Items:	Revisions:	
Date: 5/12/2017	Time: 8:09:05 PM	Sheet 6 of 3	

A

B

C

D

Sheet Title PXIe PERIPHERAL_XI3 90Pin			
Project Title PXIe PERIPHERAL 3U			
Size: A4	Item:	Revision:	
Date: 5/12/2017	Time: 8:09:06 PM	Sheet * of *	
File: CON_PXI_Express_PERIPHERAL_XI3_90PIN.SchDoc			

Altium
IBP Shanghai
Level 3 - Building 3
No.168 Linhong Road
Shanghai 200335, China

Layer	Name	Material	Thickness	Constant	Board Layer Stack
1	Top Overlay	Solder Resist	0.39mil	3.5	
2	Top Solder	Copper	1.38mil		
3	Top Layer	2x IS400 1080 FZ01	5.12mil	3.9	
4	Dielectric1	Copper	1.38mil		
5	L2	IS400 0.18 35/35	7.09mil	3.9	
6	Dielectric2	Copper	1.38mil		
7	L3	3x IS400 1080 FZ01	7.68mil	3.9	
8	Dielectric3	FR-4	11.8mil		
9	L4	Copper	1.38mil	3.9	
10	Dielectric4	3x IS400 1080 FZ01	7.68mil	3.9	
11	L5	Copper	1.38mil		
12	Dielectric5	IS400 0.18 35/35	7.09mil	3.9	
13	L6	Copper	1.38mil		
14	Dielectric6	2x IS400 1080 FZ01	5.12mil	3.9	
15	L7	Copper	1.38mil		
16	Bottom Layer	Solder Resist	0.39mil	3.5	
17	Bottom Solder				
18	Bottom Overlay				
19					

Appendix B

System Controller FPGA Firmware

Appendix C

Peripheral Module FPGA Firmware

Appendix D

2017 16th International Conference on Field-Programmable Technology (ICFPT)

An Open Source PXIe Ecosystem based on FPGA Modules

Andrew Ang*, Matt Bourne* and Robin Dykstra*

*School of Engineering and Computer Science

Victoria University of Wellington, Wellington, New Zealand 6140

Email: robin.dykstra@vuw.ac.nz

Abstract—PXIe is a modular instrumentation standard used in building electronic test equipment and modular laboratory instruments. Built on the PCIe bus standard, it provides high-speed transfer rates suitable for a wide range of applications. However, PXIe is inherently complex, usually contains proprietary IP and the solutions provided by vendors are often prohibitively expensive for some projects. To overcome these barriers, an open source PXIe ecosystem has been developed. This consists of a PCIe I/O IP block for peripheral boards together with an associated device driver and application programmers interface (API) for the PCIe root complex hosted by the system controller. The design is capable of performing system controller initiated DMA transfers over PCIe in both directions. To further simplify system development, much use has been made of Xilinx based System on Module (SOM) units available from several vendors. Our example solution consists of (a) a PXIe system controller housing a Zynq based SOM mounted on a carrier board and running Linux and (b) a peripheral board based on an Artix SOM mounted on a carrier board and also hosting an FPGA Mezzanine Card (FMC) connector for further expansion.

I. INTRODUCTION

Developed in 2005 [1], Peripheral eXtensions for Instrumentation express (PXIe) adapts the serial computer expansion bus standard PCIe into the PXI form factor. The adoption of PCIe increased the PXI backplane bandwidth from 132 MB/s to 6 GB/s while maintaining compatibility with older PXI hardware. The standard was introduced for measurement and automation applications that require high transfer rates. Inherently, PXIe platforms and modules contain proprietary intellectual property (IP). System developers have no choice but to purchase off the shelf solutions or, if developing their own hardware, still have to pay for IP. The release of high performance FPGA devices from vendors such as Xilinx and the availability of open-source Linux operating systems has made it convenient to develop an open PXIe ecosystem.

A general PXIe system is shown in figure 1. It is composed of three modules: system controller board, system timing board and at least one peripheral module. The system controller board provides a user interface through an operating system (OS), allowing control with any modules within the system. Synchronisation between multiple devices is carried out by the system timing board using a star trigger and system reference clock. Peripheral modules provide specific functionality for a wide range of applications such as oscilloscopes, waveform generators and spectrum analysers [2].

Fig. 1: General PXIe system

II. CORE ARCHITECTURE DEVELOPMENT

Development of the ecosystem consisted of developing the hardware and software components for both system controller and peripheral modules. On each peripheral module, FPGA fabric is used to implement a direct memory access and programmed input/output (DMA/PIO) engine which interfaces with the root complex of the host system. In our case the host is based on the Xilinx Zynq series system-on-chip (SoC). The management of data transfer between the host and an endpoint is provided by a device driver for the host operating system, which in this case is Linux.

Fig. 2: Stack model of PXIe architecture

Finally an API layer was developed to abstract the systems calls between user and kernel space, allowing a simple method of I/O transaction between the host and the endpoint.

Fig. 3: PCIe CDMA Subsystem

A. Peripheral PCIe CDMA Subsystem

To perform data transfers over a PCIe link, a DMA/PIO engine was designed and incorporated into the programmable logic of the endpoint. The design uses multiple IP blocks to build the overall endpoint system. This consists of a central DMA engine (CDMA)[3] and an *AXI memory mapped to PCIe* block [4]. The overall PCIe CDMA subsystem design is based on the Xilinx application note 1171 [5] and is shown in figure 3.

1) *AXI memory mapped to PCIe*: This block performs the PCIe packet decoding. It contains both a slave and master AXI port so that the host system can send and receive data over PCIe. Addressing a PCIe device requires it to be mapped to the host system's I/O address space. The PCIe endpoint informs the address mapping and writing/reading configuration through base address registers (BAR). Two endpoint BARs that are configured on the endpoint *AXI memory mapped to PCIe* block to address to endpoints as the CDMA controller and provide a scatter-gather (SG) port.

2) *AXI CDMA*: This block controls DMA between a memory-mapped source address and a memory-mapped destination address using the AXI4 protocol. This performs memory mapped DMA with a given source and destination address. The inclusion of SG provides off-loading CPU management tasks to hardware automation when requesting for non-segmented memory in the operating system. This further decreases processor utilisation on the root complex side of the PCIe link. To facilitate dynamic translation SG DMA data transfers, a translation BRAM is added to the endpoint engine to temporarily store PCIe address translation vectors. This implementation allows address translation of AXI address to a Linux's 64-bit address space.

3) *System controller root-complex engine*: Within the PXIe system, the system controller board houses the Zynq based root complex. It allows generation of PCIe transactions on behalf of the processor within the Zynq device. Using PetaLinux tools from Xilinx [6], a Linux kernel was developed for the system controller board which incorporates the provided AXI-PCIe bridge driver. What is required is the connection of the *AXI Memory Mapped to PCIe* block to the ARM processors

of the Zynq device. In this case, the *AXI Memory Mapped to PCIe* block is configured to be a root complex, thus allowing generation of PCIe traffic. A simplified diagram of the system is provided by Xilinx [7] in figure 4.

Fig. 4: Simplified diagram of root-complex system

B. Device Driver Design

Xilinx has provided an open source driver for its CDMA engine [8]. It provides the base interface between the PCIe and CDMA engine, which is then wrapped into a single PCI style character driver.

1) *Initialising access to a device*: A character (char) device can be accessed in the file system of the operating system. It is instantiated through the terminal command *insmod*, and is then managed as part of a virtual file system which is traditionally mounted at */dev* in Unix systems [9]. A device is represented as an open *file* in the Linux kernel space which is associated with two structures: *inode* and *pci_driver*. An *open* function is defined within the driver to create the two data structures. *struct pci_driver* contains a *id_table* field that tells the kernel what PCIe devices it supports and callbacks to properly initialise and remove a device. A pointer of the *pci_driver* is then registered to the PCI core using *pci_register_driver*.

2) *Direct memory access (DMA)*: Setup of the CDMA engine occurs when creating an instance of the device on the host system. An instance of the CDMA engine is created as a field of the *struct xpcie_dev*. DMA read/writes occur by setting the *source* and *destination* registers on the endpoint CDMA block. It is then followed by writing the number of bytes to the *bytes to transfer* register.

C. Application Programming Interface (API)

The API provides an abstraction layer between the kernel and user space. It consists of clearly defined function calls which use system calls to request a service from the Linux kernel. These functions include establishing a connection, writing and reading data to a particular PCIe endpoint. A summary of each function and their example use case is provided in table I and listing 1 respectively.

1) *Opening and closing a device*: The API removes the complexity of initialising and closing a device through the *open_dev* and *close_dev* functions. It requires the pathname to the device which can be found in the */dev* folder of the OS after instantiating the char driver.

2) *Reading and writing*: Read and write is abstracted in the API through the *read_data* and *write_data* functions. The use of system calls *copy_to_user* and *copy_from_user* are used to maintain the security of the kernel from unintentional user access.

3) *I/O control (ioctl)*: Also provided in the API is access to *ioctl* commands to the PCIe endpoint. These commands configure different parameters of transfers and provide statistics of transfers that has previously occurred. A summary of the available *ioctl* commands are shown in table I.

TABLE I: API reference

Command	Description
<code>open_dev(const char *path)</code>	Opens and initialise device
<code>close_dev(const char *path)</code>	Closes device and returns allocated memory
<code>read_data(int dev, unsigned char *read_buf, int addr, int size)</code>	Reads up to <i>count</i> bytes of contents from device at address <i>addr</i> then stores it in <i>read_buf</i>
<code>write_data(int dev, unsigned char *data, int addr, int size)</code>	Writes <i>count</i> bytes of data to device at address <i>addr</i>
<code>ioctl(int fd, TRANS_TIME, 0)</code>	Get previous transfer time
<code>ioctl(int fd, TRANS_SPEED, 0)</code>	Get previous transfer speed
<code>ioctl(int fd, SET_DMA, 0)</code>	Set transfers to DMA mode
<code>ioctl(int fd, SET_PIO, 0)</code>	Set transfers to PIO mode

D. Test setup

Development of the engine for the endpoint module and the device driver on the system controller board was done on a prototype system consisting of three development boards: ZC706 [10], PicoZed[11] and a Zynq Mini-ITX [12]. The latter is configured to be the root-complex while the ZC706 and PicoZed were acting as PCIe endpoints. A Texas Instrument XIO3130 4-port, 4-lane PCIe switch [13] was used to connect the host system to the PCIe endpoints. The setup is shown in figure 5. Testing of the DMA/PIO engine and driver yielded transfers through PCIe upwards of 100 MB/s.

III. PXIE HARDWARE IMPLEMENTATION

The next stage of development of the ecosystem consisted of developing custom PXIE system controller and peripheral boards for a National Instruments PXIE-1062Q chassis[14]. It is our intention to develop an open PXIE backplane as part of future work. Component distributors such as Avnet have developed modular system-on-module (SoM) solutions that contain a Xilinx FPGA or Zynq device coupled with other components such as memory, communications interfaces as well as a connector to interface to host carrier boards. By

```

/* open device */
fd = open_dev("/dev/xpcie_1");

/* write to DDR memory on Zynq endpoint with buffer
   containing data */
write_data(fd, (unsigned char *)&w_data, 0x100000, size);

/* read from DDR memory on Zynq endpoint */
read_data(fd, (unsigned char *)&r_data, 0x100000, size);

/* get previous transfer speed */
printf("Last transfer ran at %d MB/s\n", ioctl(fd,
    TRANS_SPEED, 0));

/* close device */
close_dev(fd);

```

Listing 1: API use case example of a loopback function

Fig. 5: Prototyping setup

utilising these SoMs, the complexity of the board designs are significantly reduced.

A. System controller board

The Xilinx Zynq is a system-on-chip (SoC) which integrates software and hardware programmability into one package. This unique integration provides an elegant solution in developing a system controller board. This is the bridge that interfaces the user with the PXIE system. The user can develop their own programs in the user space and use the API to interface with any peripheral module within the system. The fabric of the Zynq device contains the AXI PCIe block that enables data transfer through PCIe. A custom Eurocard sized PXIE system PCB was developed based on the Avnet PicoZed Zynq module. To comply with the PXIE standard, a PEX8624 PCIe Gen 2 switch[15] was used to fan out the PicoZed's PCIe x4 link to the required 4 ports of 4 lanes each. The block diagram of the PCB functionality and the physical board are shown in figure 7a and 8a.

B. Peripheral board

In contrast to the Zynq, the Artix only has programmable logic. The Artix FPGA device is used for the peripheral board

as it does not need to have an on-board OS. The Artix-100T fabric contains a hard block PCIe peripheral unit to implement PCIe connectivity. The complete PCIe endpoint implementation utilises 43% of the Artix-100T slices. Users can utilise the remaining slices for developing custom FPGA firmware to interface with an FMC module. With a wide variety of FMC modules on the market, the peripheral module can be customised for different applications, which in turn, expands the capabilities of an open PXie ecosystem. However, it is our intention to develop application specific peripheral modules. A Eurocard sized PXie peripheral PCB was developed based on the TE0712 [16] module. The block diagram of the PCB functionality and the physical board are shown in figure 7b and 8b.

Fig. 6: PCB block diagram

IV. CONCLUSION

We presented a functional, open PXie ecosystem consisting of a system controller and FMC carrier boards. The implementation of a simple DMA/PIO engine on both root-complex and endpoint allows fast data transactions to be performed over PCIe. Keeping user development in mind, we have reduced the software complexity of sending and receiving data to single functions through the API abstraction and reduced hardware complexity by using SoM solutions. In addition, users can utilise the remaining FPGA fabric to develop their own FMC applications on the peripheral module. Implementation of PCIe opens up the ecosystem to high-bandwidth applications. For future work, the current PCIe design can leverage on other existing open-source PCIe solutions such as RIFFA[17] or JetStream[18], which has the potential of increasing the overall system performance. All materials presented can be freely accessed by contacting the authors.

Fig. 7: Hardware implementation of system controller with PicoZed(a) and peripheral module with TE0712 (b)

REFERENCES

- [1] T. Fountain, A. McCarthy, and F. Peng, "Pci express: An overview of pci express, cabled pci express and pxi express," in *10th ICALEPCS Int. Conf. on Accelerator & Large Expt. Physics Control Systems*, 2005.
- [2] "Pxie modules," <http://www.ni.com>, 2017.
- [3] *AXI Central Direct Memory Access v4.1*, Xilinx, 4 2017.
- [4] *AXI Memory Mapped to PCI Express (PCIe) Gen2 v2.6*, Xilinx, 6 2015.
- [5] *PCI Express Endpoint-DMA Initiator Subsystem*, Xilinx, 11 2013.
- [6] *PetaLinux Getting Started user guide*, Xilinx, 11 2013.
- [7] *Zynq-7000 AP SoC Technical Reference Manual*, Xilinx, 09 2016.
- [8] "Xilinx cdma engine," <https://github.com/Xilinx/embeddedsw>, 2016.
- [9] A. Rubini and J. Corbet, *Linux device drivers*. "O'Reilly Media, Inc.", 2001.
- [10] "Zc706," <https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html>, 2017.
- [11] "Avnet," <http://zedboard.org/product/picozed>, 2017.
- [12] "Zynq mini-itx," <http://zedboard.org/product/mini-itx-board>, 2017.
- [13] "Xio3130 pcie switch," <http://www.ti.com>, 2017.
- [14] "Pxie-1062Q PXI Chassis," National Instruments, Inc. [Online]. Available: <http://www.ni.com/en-nz/support/model.pxie-1062q.html>
- [15] "24-Lane, 6-Port PCI Express Gen 2 Switch," Broadcom Corporation. [Online]. Available: <https://www.broadcom.com/products/pci-switches-bridges/pci-switches/pex8624>
- [16] "Te0712," <https://shop.trenz-electronic.de>, 2017.
- [17] M. Jacobsen, D. Richmond, M. Hogains, and R. Kastner, "Riffa 2.1: A reusable integration framework for fpga accelerators," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 4, p. 22, 2015.
- [18] M. Vesper, D. Koch, K. Vipin, and S. A. Fahmy, "Jetstream: an open-source high-performance pci express 3 streaming library for fpga-to-host and fpga-to-fpga communication," in *Field Programmable Logic and Applications (FPL)*, 2016 26th International Conference on. IEEE, 2016, pp. 1-9.

Appendix E

2017 11th International Conference on Sensing Technology (ICST)

Construction of a PXIe Platform for Instrumentation Development

Andrew Ang, Matthew Bourne, Sergei Obruchkov and Robin Dykstra
School of Engineering and Computer Science
Victoria University of Wellington
New Zealand

Abstract—PXIe is a modular instrumentation standard used in building electronic test equipment and modular laboratory instruments. Built on the PCIe bus standard, it provides high-speed transfer rates suitable for a wide range of applications. Instrumentation applications such as magnetic resonance imaging (MRI), generally has a high entry barrier due to high speed requirements and complexity. PXIe provides a good backplane option as the PCIe bus provides significant data bandwidth. However, the platform is inherently complex, usually contains proprietary IP and the solutions provided by vendors are often prohibitively expensive for some projects. To overcome these barriers, an open source PXIe platform has been developed. This consists of a PCIe I/O IP block for peripheral boards together with an associated device driver and application programmers interface (API) for the PCIe root complex hosted by the system controller. The design is capable of performing system controller initiated DMA transfers over PCIe in both directions. To further simplify system development, much use has been made of Xilinx based System on Module (SoM) units available from several vendors. Our example solution consists of (a) a PXIe system controller housing a Zynq based SoM mounted on a carrier board and running Linux and (b) a peripheral board based on an Artix SoM mounted on a carrier board and also hosting an FPGA Mezzanine Card (FMC) connector for further expansion.

I. INTRODUCTION

Developed in 2005 [1], Peripheral eXtensions for Instrumentation express (PXIe) adapts the serial computer expansion bus standard PCIe into the PXI form factor. An example chassis is shown in figure 1. The adoption of PCIe increased the PXI chassis backplane bandwidth from 132 MB/s to 6 GB/s while maintaining compatibility with older PXI hardware [2]. The standard was introduced for measurement and automation applications that require high transfer rates. This makes general instrumentation development very attractive. However, it is built on proprietary intellectual property (IP). Instrumentation developers have no choice but to purchase off the shelf solutions or, if developing their own hardware, still have to pay for IP. The release of high performance FPGA devices from vendors such as Xilinx and the availability of open-source Linux operating systems has made it convenient to develop an open PXIe platform. One area that would benefit significantly from PXIe is the development of Magnetic Resonance Imaging (MRI) systems which are modular in nature due to the need for a range of signal types. This includes RF, for sample coil excitation and receive, and high

power audio frequency drive for gradient coils that perturb the magnetic field while obtaining an image.

An example of a PXIe system for MRI is shown in figure 2. It is composed of several modules: system controller board, system timing board and a few peripheral modules. These are housed in a PXIe chassis as shown in figure 1. The system controller board provides a user interface through an operating system (OS), allowing control with any modules within the system. Synchronisation between multiple devices is carried out by the system timing board using a star trigger and system reference clock. Peripheral modules provide specific functionality for a wide range of applications such as oscilloscopes, data capture, waveform generators and spectrum analysers [3].

Fig. 1: PXIe chassis [4]

Fig. 2: Example PXIe system for MRI application

II. CORE ARCHITECTURE DEVELOPMENT

Development of the platform consisted of developing the hardware and software components for both system controller and peripheral modules. On each peripheral module, FPGA fabric is used to implement a direct memory access and programmed input/output (DMA/PIO) engine which interfaces with the root complex of the host system. In our case the host is based on the Xilinx Zynq series system-on-chip (SoC). The management of data transfer between the host and an endpoint is provided by a device driver for the host operating system, which in this case is Linux.

Fig. 3: Stack model of PXIe architecture

Finally an API layer was developed to abstract the systems calls between user and kernel space, allowing a simple method of I/O transaction between the host and the endpoint.

A. Peripheral PCIe CDMA Subsystem

To perform data transfers over a PCIe link, a DMA/PIO engine was designed and incorporated into the programmable logic of the endpoint. The design uses multiple IP blocks to build the overall endpoint system. This consists of a central DMA engine (CDMA)[5] and an *AXI memory mapped to PCIe* block [6]. Both blocks are connected to an AXI connect which interfaces with a memory management controller. This allows data to be directly stored onto external memory which can be fetched from the host through DMA.

1) *AXI memory mapped to PCIe*: This block performs the PCIe packet decoding. It contains both a slave and master AXI port so that the host system can send and receive data over PCIe. Two base address registers (BAR) are configured which informs the PCIe endpoint about address mapping (PCIe:BAR) to the host system's I/O space, and reading/writing (AXI:BAR) procedures such as scatter-gather (SG).

2) *AXI CDMA*: This block controls DMA between a memory-mapped source address and a memory-mapped destination address using the AXI4 protocol. This performs

memory mapped DMA with a given source and destination address. The inclusion of SG provides off-loading CPU management tasks to hardware automation when requesting for non-segmented memory in the operating system. This further decreases processor utilisation on the root complex side of the PCIe link. To facilitate dynamic translation SG DMA data transfers, a translation BRAM is added to the endpoint engine to temporarily store PCIe address translation vectors.

3) *System controller root-complex engine*: Within the PXIe system, the system controller board houses the Zynq based root complex. It allows generation of PCIe transactions on behalf of the processor within the Zynq device. Using PetaLinux tools from Xilinx [7], a Linux kernel was developed for the system controller board which incorporates the provided AXI-PCIe bridge driver. What is required is the connection of the *AXI Memory Mapped to PCIe* block to the ARM processors of the Zynq device. In this case, the *AXI Memory Mapped to PCIe* block is configured to be a root complex, thus allowing generation of PCIe traffic.

B. Device Driver Design

Xilinx has provided an open source driver for its CDMA engine [8]. It provides the base interface between the PCIe and CDMA engine, which is then wrapped into a custom PCI style character driver.

1) *Initialising access to a device*: A character (char) device can be accessed in the file system of the operating system. It is instantiated through the terminal command *insmod*, and is then managed as part of a virtual file system which is traditionally mounted at */dev* in Unix systems [9]. A device is represented as an open file in the Linux kernel space which is associated with two structures: *inode* and *pci_driver*. An *open* function is defined within the driver to create the two data structures. *struct pci_driver* contains a *id_table* field that tells the kernel what PCIe devices it supports and callbacks to properly initialise and remove a device. A pointer of the *pci_driver* is then registered to the PCI core using *pci_register_driver*.

2) *Direct memory access (DMA)*: Setup of the CDMA engine occurs when creating an instance of the device on the host system. An instance of the CDMA engine is created as a field of the *struct xpcie_dev*. DMA read/writes occur by setting the *source* and *destination* registers on the endpoint CDMA block. It is then followed by writing the number of bytes to the *bytes to transfer* register.

C. Application Programming Interface (API)

The API provides an abstraction layer between the kernel and user space. It consists of clearly defined function calls which use system calls to request a service from the Linux kernel. These functions include establishing a connection, writing and reading data to a particular PCIe endpoint. An example use case is provided in listing 1.

1) *Opening and closing a device*: The API removes the complexity of initialising and closing a device through the *open_dev* and *close_dev* functions. It requires the pathname to the device which can be found in the */dev* folder of the OS after instantiating the char driver.

```

/* open device */
fd = open("/dev/xcpcie_1");

/* write to DDR memory on Zynq endpoint with buffer
   containing data */
write_data(fd, (unsigned char *)&w_data, 0x00100000);

/* read from DDR memory on Zynq endpoint */
read_data(fd, (unsigned char *)&r_data, 0x00100000);

/* get previous transfer speed */
printf("Last transfer ran at %d MB/s\n", ioctl(fd,
    TRANS_SPEED, 0));

/* close device */
close("/dev/xcpcie_1");

```

Listing 1: API use case example of a loopback function

2) *Reading and writing*: Read and write is abstracted in the API through the *read_data* and *write_data* functions. The use of system calls *copy_to_user* and *copy_from_user* are used to maintain the security of the kernel from unintentional user access. This is then followed by reading or writing to the device using *read* or *write* system calls. These abstracted functions in the API require three arguments. These are a pointer (handle) to the device associated *file* structure, a user space buffer to store received or to be written data, and an address on the endpoint location.

3) *I/O control (ioctl)*: Also provided in the API is access to *ioctl* commands to the PCIe endpoint. These commands configure different parameters of transfers and provide statistics of transfers that has previously occurred. A summary of the available *ioctl* commands are shown in table I.

TABLE I: API reference

Command	Description
<code>open_dev(const char *path)</code>	Opens and initialise device
<code>close_dev(const char *path)</code>	Closes device and returns allocated memory
<code>read_data(int dev, unsigned char *read_buf, int addr)</code>	Reads contents from device at address <i>addr</i> then stores it in <i>read_buf</i>
<code>write_data(int dev, unsigned char *data, int addr)</code>	Writes data to device at address <i>addr</i>
<code>ioctl(int fd, TRANS_TIME, 0)</code>	Get previous transfer time
<code>ioctl(int fd, TRANS_SPEED, 0)</code>	Get previous transfer speed
<code>ioctl(int fd, SET_DMA, 0)</code>	Set transfers to DMA mode
<code>ioctl(int fd, SET_PIO, 0)</code>	Set transfers to PIO mode

D. Test setup

Development of the engine for the endpoint module and the device driver on the system controller board was done on a prototype system consisting of three development boards: ZC706 [10], PicoZed [11] and a Zynq Mini-ITX [12]. The latter is configured to be the root-complex while the ZC706 and PicoZed were acting as PCIe endpoints. A Texas Instrument XIO3130 4-port, 4-lane PCIe switch [13] was used to connect the host system to the PCIe endpoints. The setup is shown in figure 4. Testing of the DMA/PIO engine and driver yielded transfers through a single PCIe lane upwards of 100 MB/s.

Fig. 4: Prototyping setup

III. PXIE HARDWARE IMPLEMENTATION

The next stage of development of the platform consists of developing custom PXIE system controller and peripheral boards. Component distributors such as Avnet have developed modular system-on-module (SoM) solutions that contain a Xilinx FPGA or Zynq device coupled with other components such as memory, communications interfaces as well as a connector to interface to host carrier boards. By utilising these SoMs, the complexity of the board designs are significantly reduced.

Fig. 5: PicoZed(a)[9] and TE0712 Artix Module (b)[10]

A. System controller board

The Xilinx Zynq is a system-on-chip (SoC) which integrates software and hardware programmability into one package. This unique integration provides an elegant solution in developing a system controller board. This is the bridge that interfaces the user with the PXIE system. The user can develop

their own programs in the user space and use the API to interface with any peripheral module within the system. The fabric of the Zynq device contains the AXI PCIe block that enables data transfer through PCIe. A custom Eurocard sized PXIe system PCB was developed based on the Avnet Picozed Zynq module. The block diagram of the PCB functionality as well as a render of the physical board are shown in figures 7a and 8a.

B. FMC carrier peripheral module

In contrast to the Zynq, the Artix only has programmable logic. The Artix FPGA device is used for the peripheral board as it does not need to have an on-board OS. The Artix fabric contains the necessary PCIe peripheral units to implement PCIe connectivity and also the logic required to interface with FMC modules that would be connected to the carrier card. Another Eurocard sized PXIe system PCB was developed based on the TE0712 [14] module from Trenz Electronics. The block diagram of the PCB functionality and a render of the physical board are shown in figures 7b and 8b.

Fig. 6: PCB block diagram

IV. CONCLUSION

We presented a functional, open PXIe platform consisting of a system controller and FMC carrier boards. The implementation of the supporting DMA/PIO engine on both root-complex and endpoint allows fast data transactions to be performed over PCIe. Keeping user development in mind, we have reduced the software complexity of sending and receiving data to single functions through the API abstraction and reduced hardware complexity by using SoM solutions. Both modules presently support 4 lanes at PCIe generation 1 specification (1 GB/s),

(a)

(b)

Fig. 7: Hardware implementation of system controller (a) and peripheral module (b)

but the desire is to work towards generation 3 (3.94 GB/s) which is now used by several vendors. All material presented in this paper can be freely accessed by contacting the authors.

REFERENCES

- [1] T. Fountain, A. McCarthy, and F. Peng, "Pci express: An overview of pci express, cabled pci express and pxi express," in *10th ICALEPCS Int. Conf. on Accelerator & Large Expt. Physics Control Systems*, 2005.
- [2] G. Caesar and M. Wetzel, "Pxi express: extending backplanes to 6 gbyte/s while maintaining backwards compatibility," in *Autotestcon, 2005. IEEE*. IEEE, 2005, pp. 231–234.
- [3] "Pxi modules," <http://www.ni.com/en-nz/shop/select/pxi-modules-category>, 2017.
- [4] "Pxi chassis," <http://www.ni.com/en-nz/support/model.pxi-1062q.html>, 2017.
- [5] *AXI Central Direct Memory Access v4.1*, Xilinx, 4 2017.
- [6] *AXI Memory Mapped to PCI Express (PCIe) Gen2 v2.6*, Xilinx, 6 2015.
- [7] *PetaLinux Getting Started user guide*, Xilinx, 11 2013.
- [8] "Xilinx cdma engine," <https://github.com/Xilinx/embeddedsw>, 2016.
- [9] A. Rubini and J. Corbet, *Linux device drivers*. " O'Reilly Media, Inc.", 2001.
- [10] "Zc706," <https://www.xilinx.com/products/boards-and-kits/ek-z7-zc706-g.html>, 2017.
- [11] "Avnet," <http://zedboard.org/product/picozed>, 2017.
- [12] "Zynq mini-itx," <http://zedboard.org/product/mini-itx-board>, 2017.
- [13] "Xio3130 pcie switch," <http://www.ti.com/product/XIO3130>, 2017.
- [14] "Te0712," <https://shop.trenz-electronic.de/en/TE0712-02-200-2C-Micromodule-Artix-7-XC7A200T-2C-4x5cm-standard-footprint-com.-temp.-range?c=341>, 2017.