Design and Analysis of a Real-time Data Monitoring Prototype for the LWA Radio Telescope

Sushrutha Vigraham

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical Engineering

February 1, 2011
Blacksburg, Virginia

Keywords: *FPGA, data streaming, PCIe, data-centric computing*

Design and Analysis of a Real-time Data Monitoring Prototype for the LWA Radio Telescope

Sushrutha Vigraham

ABSTRACT

Increasing computing power has been helping researchers understand many complex scientific problems. Scientific computing helps to model and visualize complex processes such as molecular modelling, medical imaging, astrophysics and space exploration by processing large set of data streams collected through sensors or cameras. This produces a massive amount of data which consume a large amount of processing and storage resources. Monitoring the data streams and filtering unwanted information will enable efficient use of the available resources. This thesis proposes a data-centric system that can monitor high-speed data streams in real-time. The proposed system provides a flexible environment where users can plug-in application-specific data monitoring algorithms. The Long Wavelength Array telescope (LWA) is an astronomical apparatus that works with high speed data streams, and the proposed data-centric platform is developed to evaluate FPGAs to implement data monitoring algorithms in LWA. The throughput of the data-centric system has been modelled and it is observed that the developed data-centric system can deliver a maximum throughput of 164 MB/s.

To Amma and Appa

Acknowledgments

I profusely thank my primary advisor, Dr. Cameron Patterson, for providing me the opportunity to work on a very interesting project and offering invaluable guidance throughout my thesis. I thank Dr. Ellingson, for inducting me to the LWA project. I am grateful to Dr. Paul Plassmann for consenting to be a member of my thesis committee. This research was supported in part via contract N00014-07-C-0147 from The Office of Naval Research.

I am highly indebted to Christopher Wolfe, without whom this thesis would not have materialized as well as it has now. I thank him for imparting his profound knowledge and extraordinary advice. I am very thankful to Gautham Chavali for reviewing my thesis and nudging me to complete my thesis well and on time. His patience and moral support has strengthened me immensely during the tough times.

I would like to thank Piyush Garyali and Athira Chandrasekharan for guiding me through the initial phase of my Masters. I am grateful to Guruprasad Subbarayan and Tannous Frangieh for their timely advice. Special thanks to my colleagues, Karthick Lakshmanan, Ramya Priyadarshini, Sureshwar Rajagopalan, Mrudula Karve, Rohit Asthana, Umang Parek and Karl Pereira for creating an enjoyable work environment.

My heartfelt thanks to my good friend, Saparya Krishnamoorthy, for her immense support and the memorable time she created, at Virginia Tech.

I am very grateful to Pavithra Sekhar for being a wonderful roommate and offering constant support all through my thesis. I would like to thank Abirami Srinivasan, Karthik Ganesan, Srinivasa Raghavan Santhanam and Praveen Kumar for all the fun times together.

Last but the best, I am blessed to have two great friends, my mom and my dad, who have been my role models and the reason behind my every success. I am highly indebted to them for their unconditional love, support and encouragement because of whom, I am what I am now.

Sushrutha Vigraham

Blacksburg

Feb 1, 2011.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time data streaming plays a crucial role in various data-intensive applications. Scientific activities such as astronomical observations using telescopes, intensive simulations in biological research, and weather prediction in meteorology require real-time data analysis. On a commercial scale, vehicle navigation systems providing directions based on the vehicle location and security surveillance using various sensors also have a need for real-time processing. In all these applications, the processed data streams are to be sent to appropriate output devices such as display screens in navigation systems or storage units for further processing as in astronomical observations. Recent advancements in high performance computing devices such as FPGAs and GPUs, and high-speed communication interfaces such as Peripheral Component Interconnect Express (PCIe) and 10 Gigabit Ethernet(10GbE), have led to the design of high performance data streaming systems.

In this thesis, a data-centric system is developed to stream and monitor data in real-time. Data-centric systems acquire, analyze and store data streams originated from inputs such as antennas [1]. The data-centric system is developed as a prototype for a data monitoring system in the Long Wavelength Array (LWA) telescope. LWA is an astronomical instrument being built in New Mexico for radio astronomy [2]. The proposed data-centric system will help in monitoring the celestial emissions captured by LWA antennas may help in mitigating radio frequency interference (RFI). In this thesis, data-centric system and data monitoring system are synonymous.

1.1 Motivation

LWA station captures celestial radio emissions through its antennas. Data acquired by the stations are routed to the storage units. Capturing unusable data (such as data with a high amount of RFI) might result in wasteful consumption of storage capacity and processing power utilized for recording and processing of unwanted data. The proposed data monitoring system provides a framework that can be used to detect RFI in the radio signals captured by an LWA station, and discard unusable data. The advantages of the new data monitoring system for LWA can be summarized as follows:

- *Efficient utilization of storage capacity:* Every LWA station has dedicated data storage units. Due to long periods of observation, the storage disks frequently run out of space and need to be replaced with empty disks. A large amount of storage resources and increased LWA station maintenance are required to replace the disks. Monitoring data helps in detecting unusable data which can be discarded. Reducing the amount of unusable data before storage will enable efficient use of the available storage space and minimize the costs associated with extra storage and LWA station maintenance.

- *Efficient utilization of processing time and power:* Data recorded in the storage disks of an LWA station are processed at a central unit, usually a computer cluster. These computers are high performance shared computing resources that consume a large amount of power. Filtering data at the LWA stations avoids processing of unnecessary data and hence saves the processing time and power along with the costs involved in utilizing the high performance resources.

The prototype data monitoring system developed in this thesis is used to evaluate FPGAs for monitoring data in an LWA station. The developed data-centric system might also be used for data monitoring and analysis in real-time streaming applications such as computer surveillance systems to monitor Internet traffic.

1.2 Contributions

A generic FPGA-based data-centric system that can be integrated with any PC-based streaming application is developed in this thesis. The data-centric system is developed as a prototype FPGA-based data monitoring system for LWA in order to evaluate FPGAs for implementing data monitoring algorithms in LWA. The data-centric system is not tied to a specific application but provides a framework to which data monitoring algorithms can be plugged-in. Using Xilinx System Generator (XSG), a model-based tool for developing algorithms on FPGA, DSP functions such as a 8-point Fast Fourier Transform (FFT) block and a Finite Impulse Response (FIR) block of order 21 are developed. The FFT and FIR blocks developed using XSG are used to test the data-centric system and demonstrate its plug-in capability, but are not intended as the goal data monitoring algorithms. A synthetic data generator developed in this thesis generates random data and mimics an input data source which transmits data-to-be-monitored to the the data-centric system via a 10GbE link.

In the current design of an LWA station, the data generated by the Digital Processor (DP) system is routed to the Monitoring and Control System Data Recorder (MCS-DR) system which records data in storage disks known as Data Recorder Storage Units (DRSUs). The FPGA component of the developed data monitoring system is inserted into the already existing MCS-DR Personal Computer (MCS-DR PC). Figure 1.1 shows the LWA station current design and Figure 1.2 shows the modified design with an FPGA for data monitoring.

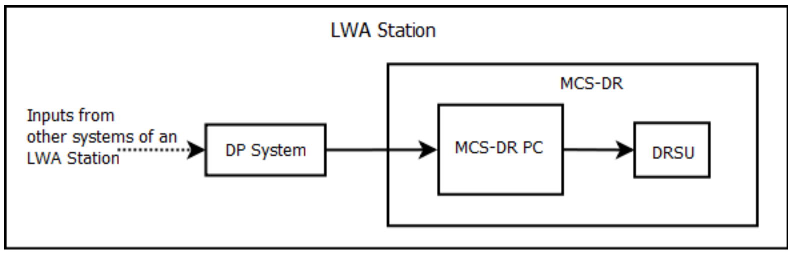


Figure 1.1: Current LWA station set-up.

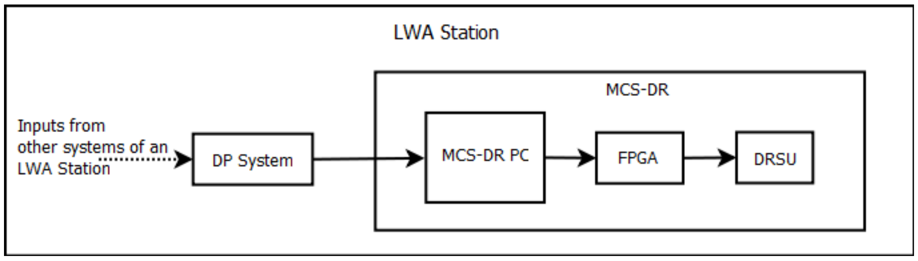


Figure 1.2: Modified LWA station set-up with FPGA based data monitoring capability.

This thesis makes the following contributions:

1. A generic data-centric system that can stream and monitor data in real-time is developed in this thesis. The data-centric system is used to evaluate FPGAs as a possible data monitoring device in LWA.

2. A plug-in capability has been provided for the data-centric system which allows its users to develop monitoring algorithms in their preferred development environment and integrate them with the developed data-centric system.

3. A synthetic data generator is developed to mimic an input data source which transmits data-to-be-monitored to the the data-centric system via a 10GbE link. The synthetic data generator is basically a 10GbE transmitter that transmits random data to the data centric system.

4. Establishing communication links:

   a) A 10GbE receiver is configured on the data-centric system. In this thesis, the 10GbE receiver is used to receive data from the synthetic data generator (10GbE transmitter) and establishes a one-way communication link between the synthetic data generator and the data-centric system.

   b) A PCIe link between the FPGA and a PC.

1.3 Thesis Organization

This thesis is organized into six chapters. Chapter 2 briefly describes the related components of the LWA architecture and discusses the features of the FPGA and the selection criteria for the XUPV5 board used in this thesis. An overview of the PCIe and 10GbE interfaces is also presented in Chapter 2. Chapter 3 provides the information required to develop algorithms on FPGAs. Chapter 4 describes the implementation of the proposed data monitoring system in detail. An analysis of the data monitoring system and the factors affecting its throughput is provided in Chapter 5. Chapter 6 presents conclusions and discusses future work.

Chapter 2

Background

This chapter provides the background information about the LWA station architecture and the communication interfaces used in the data monitoring system and the selection criteria for the XUPV5 board chosen in this thesis. Section 2.2 gives a brief description of the systems of an LWA station relevant to this thesis. Section 2.3 lists the available high performance digital devices capable of implementing data monitoring algorithms, and Section 2.4 lists the features of FPGAs. The selection criteria for the XUPV5 board used in this thesis is mentioned in Section 2.5. An overview of the communication interfaces used between various systems of the data monitoring system is discussed in Section 2.6.

2.1 The Long Wavelength Array (LWA)

Astronomy, the study of outer space, has made a significant contribution in expanding human knowledge about the universe. The branch of astronomy that studies radio emission from celestial bodies is called radio astronomy. Astronomical instruments called radio telescopes are used in radio astronomy. Radio telescopes consist of a radio receiver and an antenna system used to detect radio-frequency radiation emitted by celestial radio sources such as radio galaxies. LWA is a large radio telescope being constructed in New Mexico which will expand the knowledge of the energetics and properties of many cosmic objects and events, the solar system, and the ionosphere [2]. LWA will consist of 53 stations distributed over a region 400 km in diameter, with each station consisting of an array of 256 dipole antennas and will operate in the frequency range of 10 to 88 MHz. Figure 2.1 shows an aerial view of the first LWA station, LWA-1 and Figure 2.2 gives a close-up view of the crossed-dipole antenna.

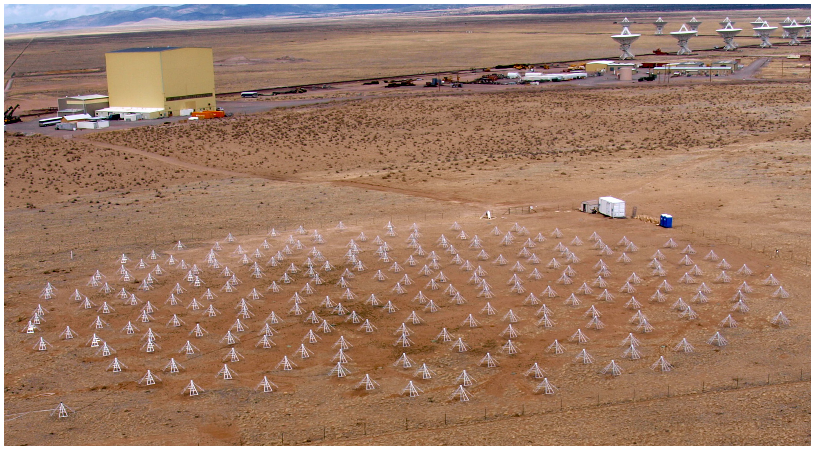


Figure 2.1: Aerial view of LWA-1, the first LWA station set up in New Mexico, USA.

Outputs from the antennas are amplified and filtered to the frequency range of interest. The filtered signals are digitized and transmitted to a the DP system. The outputs of the DP system are recorded to storage disks. The signal flow in a LWA station is summarized in Figure 2.3.

2.2 LWA Station Architecture

A LWA station is divided into subsystems which handle various functions of the station. A detailed description of the LWA station architecture and its subsystems is specified in [3]. A

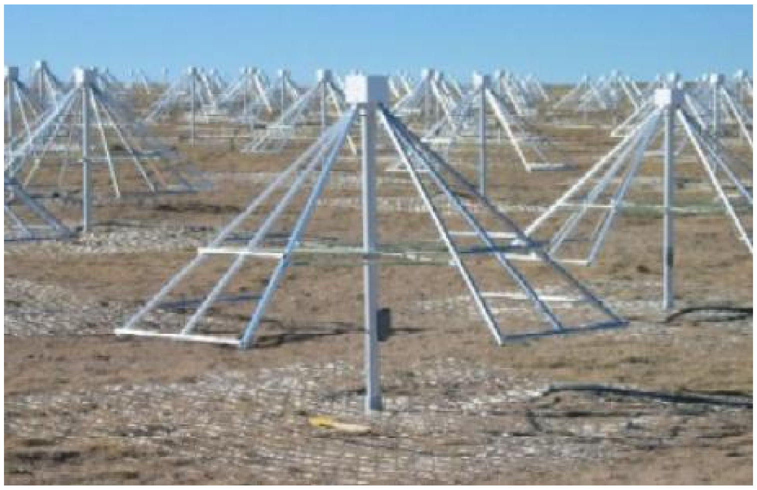


Figure 2.2: Close-up view of an LWA-1 crossed-dipole antenna.

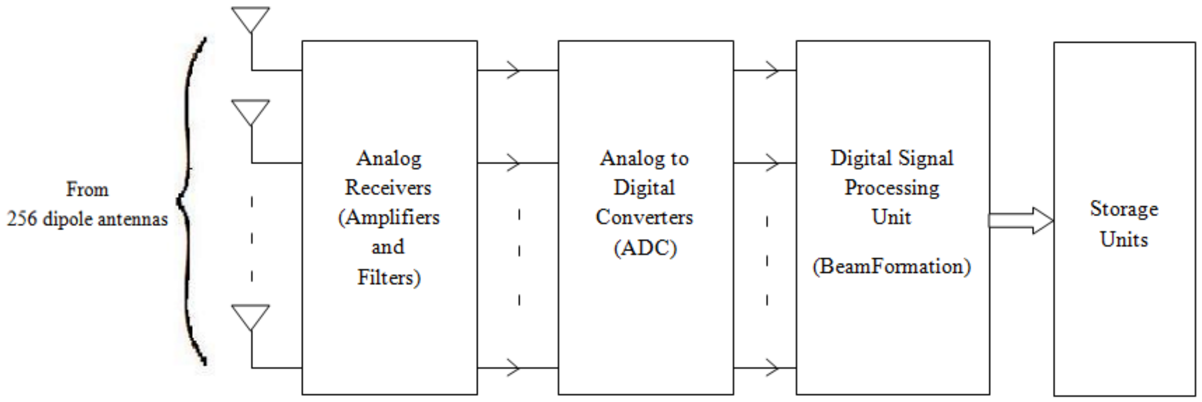


Figure 2.3: Signal flow in a LWA station.

brief description of the relevant subsystems is given below.

2.2.1 Analog Signal Processor (ASP)

As shown in Figure 2.4, the LWA antennas capture radio signals and transmit them to the ASP (Analog Signal Processor) which amplifies the raw radio signals and filters them to the frequency range of interest to enable further processing of the signal.

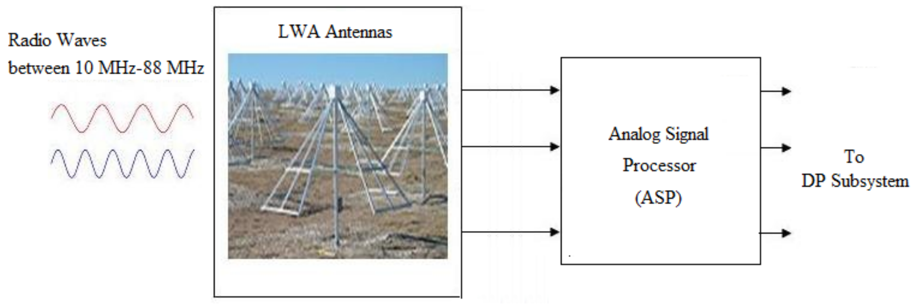


Figure 2.4: Analog Signal Processing: LWA antennas to ASP.

2.2.2 Digital Processor (DP)

The outputs of the ASP are digitized by the Digitizer (DIG) subsystem (set of analog to digital converters) and distributed to the Beam Forming Units (BFUs), the Narrow-Band Transient Buffer (TBN) and the Wide-band Transient Buffer (TBW) of the DP system. Each BFU forms a beam in the desired pointing direction and transmits it to the corresponding Digital Receiver (DRX). The TBN and TBW systems record the outputs of the DIG for later recovery and analysis. Figure 2.5 shows a block diagram of the DP system in an LWA station.

The DRXs route the corresponding BFU outputs to MCS-DR or alternative backends such as RFI and survey monitoring backends or spectrometers. The TBN and TBW systems help in the analysis and diagnosis of the LWA station in which they are installed and will also help

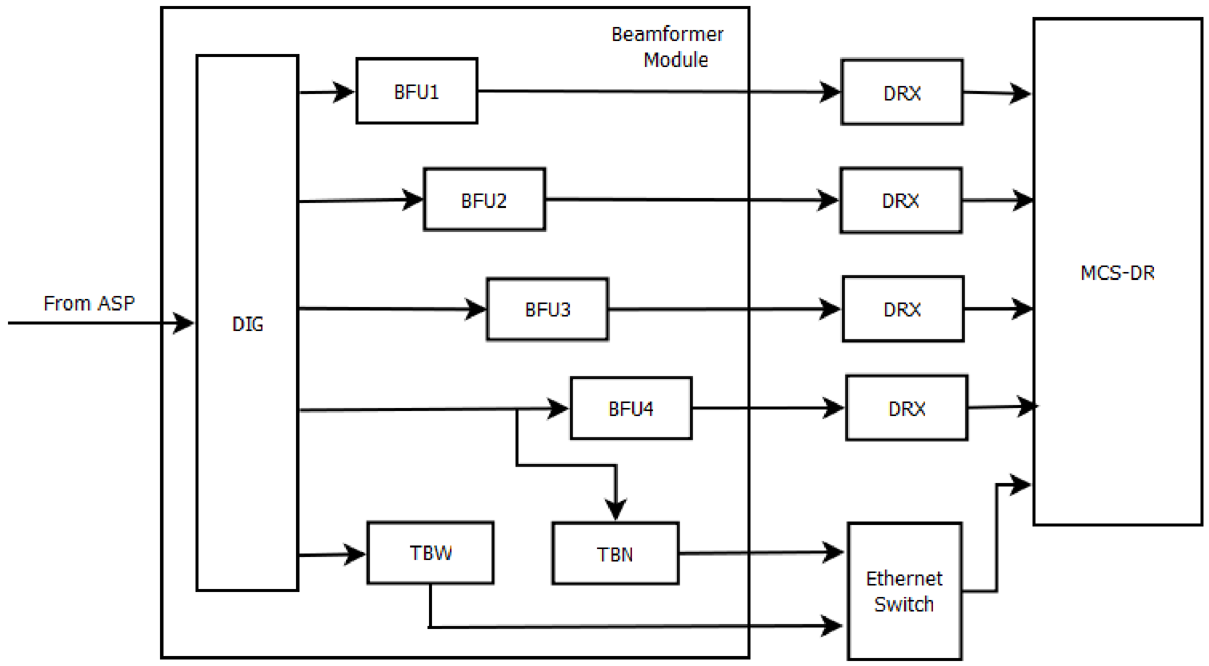


Figure 2.5: Digital Signal Processing: Digital Processor (DP).

in all-sky imaging and monitoring. The system developed in this thesis provides a framework for a backend system on which data from DRXs and TBW/TBN can be monitored. The developed framework is a part of the MCS-DR which is discussed in the next section.

2.2.3 Monitor and Control System Data Recorder (MCS-DR)

Each MCS-DR PC receives the output of the DP system and records it to a DRSU. As shown in Figure 2.6, MCS-DR consists of five identical PCs with each connected to its respective DRSU. A DRSU is a hard disk array which can record up to 5 TB of data. The PC component of the MCS-DR receives inputs from the DP system via a 10GbE link and sends outputs to the DRSUs via an eSATA link. In the current design, all data acquired by an MCS-DR PC is streamed directly to its associated DRSU. More information about MCS-DR can be found in [4].

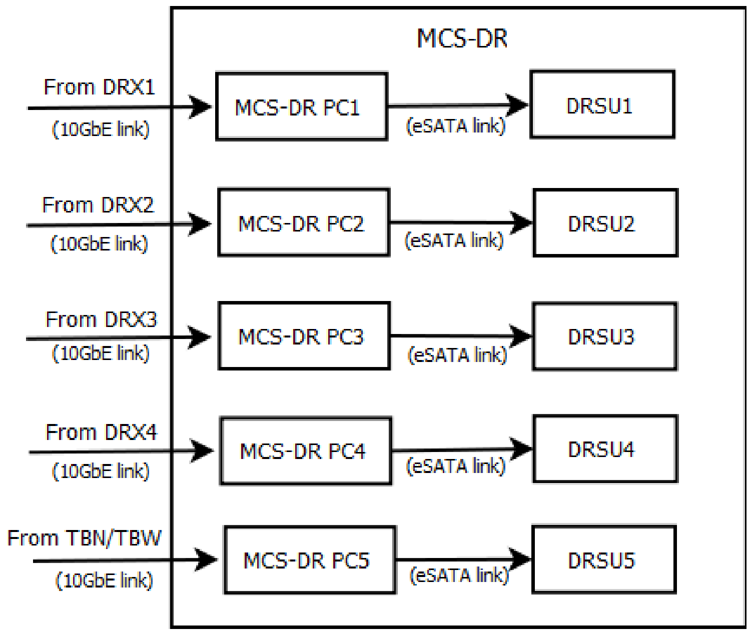


Figure 2.6: Monitoring and Control System–Data Recorder (MCS-DR).

In this thesis, the developed data monitoring system consists of a PC and a FPGA. When using the data monitoring system with LWA, the PC component of the data monitoring system refers to MCS-DR PC. The FPGA is an inserted between the PC and the DRSUs as shown in Figure 2.7.

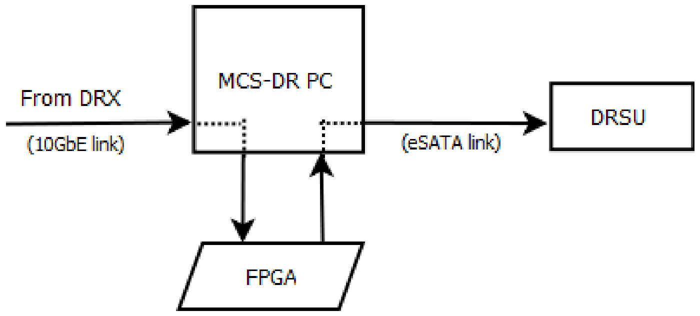


Figure 2.7: MCS-DR PC with an FPGA.

2.3 High-Performance Digital Devices

Some of the digital devices capable of implementing data monitoring algorithms include General Purpose Processors (GPPs), Digital Signal Processing (DSP) processors, Graphics Processing Units (GPUs), FPGAs, and ASICs. GPPs are easy-to-use, re-programmable platforms that can perform a wide range of operations. With advances in clock technology and multi-core architectures, PCs are able to function at high clock speeds (up to ∼3.3GHz) and are being used for high-performance computing. GPUs consist of a number of parallel processing units which distribute the workload in a data-parallel manner. The performance delivered by a GPU varies with the type and size of applications [5] [6]. DSP processors can be viewed as GPPs with a fixed number of DSP hardware blocks to perform common DSP operations (such as FFT, IIR filtering, and Viterbi decoding) efficiently. FPGAs are parallel processing devices with flexible hardware whose features are discussed in Section 2.4. Berkeley Design Technology Inc. (BDTI) Communication benchmark (OFDM) results show that FPGAs yield a higher performance per cost than DSP processors for highly parallel algorithms [7]. ASICs are high performance devices which are optimized in area, power and performance but are tied to a specific design and cannot be modified. ASICs have a rigid nature and associated with high costs. As this thesis aims at evaluating FPGAs for data monitoring, even though all the above devices are capable of implementing data monitoring algorithms, FPGAs have been selected. FPGAs possess the following features which makes them suitable for the proposed system:

2.4 Features of FPGAs

The architecture of FPGAs and the development process is discussed in Chapter 3. The features of FPGAs are summarized as follows:

- FPGAs have flexible hardware which makes them scalable and best suited for high-performance, parallel applications.

- FPGAs provide massive, low-level parallelism. Such parallelism makes FPGAs well-suited to real-time data streaming applications.
- Modern FPGAs come with dedicated DSP cores which ease the design process and deliver high performance for DSP applications.
- Model-based tools such as XSG provide simpler and faster development of designs on FPGAs.
- Designs developed on FPGAs can be easily ported to other FPGAs.
- Modern FPGAs support various communication interfaces such as PCIe and Gigabit Ethernet which allows FPGAs to be integrated with other systems.

FPGA advantages come at the cost of design time. In fixed architecture devices (such as the GPPs, GPUs and DSP processors), the user writes software on tested stable hardware whereas in FPGAs, the user implements designs on the user-configured hardware. Designing on FPGAs is a time-consuming process and requires specialized skills to design and test the user-configured hardware. Model-based tools reduce the development time on FPGAs but presently have a limited set of library functions. Developments in FPGA design tools and library functions should improve productivity [8]. More about programming using FPGAs is discussed in Chapter 3.

2.5 XUPV5 Development Board

FPGAs are generally not stand-alone devices, but rather are integrated with devices such as display controllers, off-chip memories, switches, and LEDs placed on a printed circuit board. The capabilities of these FPGA boards vary from vendor to vendor. In this thesis, the Xilinx XUPV5 development board with a Virtex-5 FPGA has been selected for the data monitoring system. The features of Virtex-5 FPGAs can be found at [9]. Figure 2.8 shows a picture of the XUPV5 board.

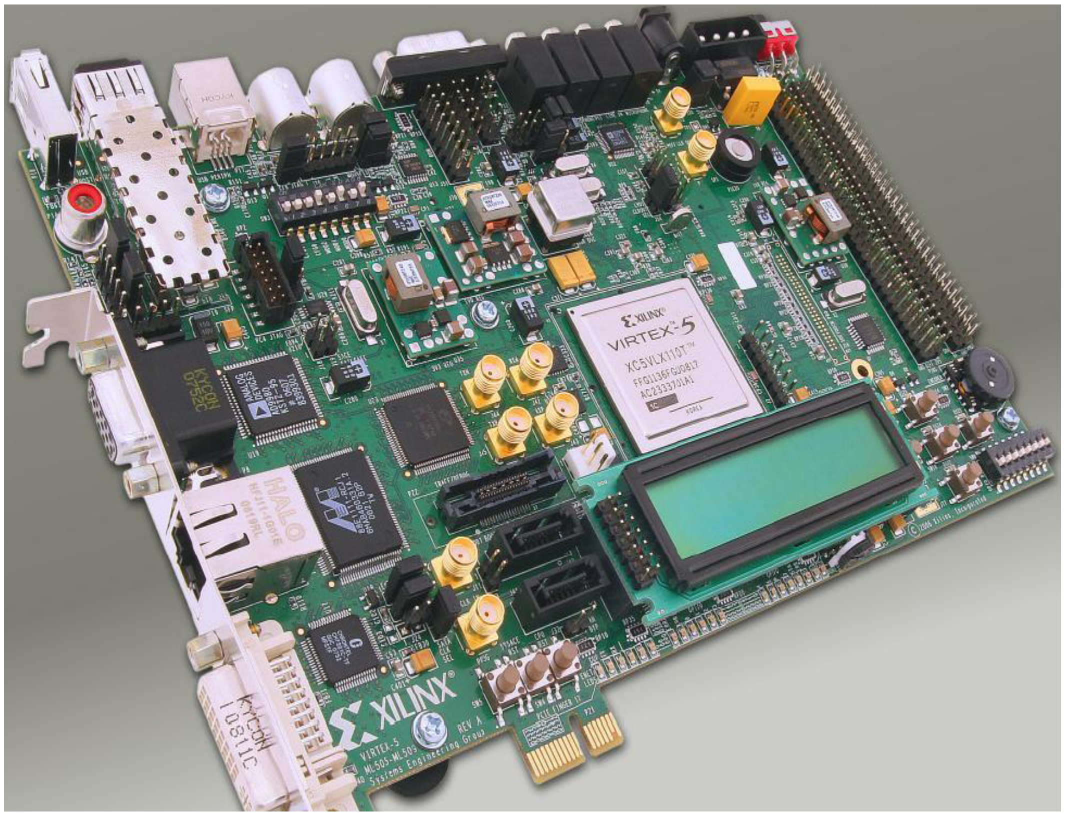


Figure 2.8: XUPV5 board with a Virtex-5 FPGA.

The selection criteria for the XUPV5 board is summarized as follows:

- *Compatible interface with the PC:* The Virtex-5 has an integrated PCIe block (compliant with the PCI Express Base Specification 1.1) which supports x1, x4, or x8 lanes per block [9]. The XUPV5 board is connected to a PC which controls and regulates data to and from the XUPV5 board. In LWA, PC refers to the MCS-DR PC to which the XUPV5 board can be plugged in through a PCIe slot.

- *Data Rate:* The DRXs of the DP system output data at a maximum rate of 80 MiB/s whereas the TBN/TBW can output data at a maximum rate of 112 MiB/s.

The throughput of the XUPV5 board varies with the data monitoring algorithms. Theoretically, the maximum throughput of the XUPV5 board is 250 MB/s per lane which is the theoretical maximum throughput achievable with a PCIe interface. Experimentally, as discussed in Chapter 5, the XUPV5 board can deliver a maximum throughput of 160 MB/s through its single lane PCIe interface which is higher than the required data rate of the DP system (maximum of 112 MiB/s through TBN/TBW). The dependence of the throughput of the XUPV5 board with respect to the parameters of the data monitoring system is discussed in Chapter 5.

- *Off-the-shelf device:* The XUPV5 board has been selected over other development boards as it was readily available and consists of a Xilinx Virtex-5 FPGA which has the features suited to the monitoring system.
- *Ease of installation and algorithm development:* The XUPV5 is compact and can be easily integrated with other systems. It can communicate with other devices through a PCIe slot or a 1GbE port. Using model-based tools such as XSG, development of data monitoring algorithms can be made simpler and faster.
- *Cost per board:* The cost of one XUPV5 board is around $2000 for commercial customers and $750 for academic customers [10].
- *Portability:* The monitoring system targeting the Virtex-5 FPGA can be ported to other FPGA families without too much effort.

2.6 Communication Interfaces

As shown in Figure 2.9, the PC component of the data monitoring system receives data via a 10GbE link. The FPGA receives and transmits data from and to the PC via a PCIe link.

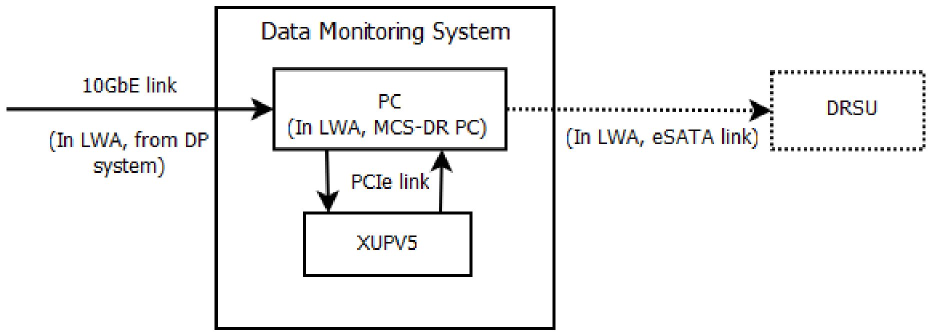


Figure 2.9: Communication interfaces between the components of the data monitoring system.

2.6.1 10 Gigabit Ethernet Interface

10GbE is an Ethernet interface (IEEE 802.3ae) which operates at a nominal rate of 10 Gigabits per second. Ethernet, formally referred to as IEEE 802.3, is a standard communication protocol that uses packet-switched technology (data is transferred through the interface in packets). The systems communicating using a Ethernet standard need to have an Ethernet network interface card installed on them and are usually physically connected to each other by a cable known as Ethernet cable. The Ethernet card implements the electronic circuitry required to communicate using an Ethernet standard and acts as an interface between the Ethernet network and the system on which the Ethernet card is installed. The Ethernet card provides has a unique ID, known as the Media Access Control (MAC) address, that acts as an address to the system.

In this thesis, the proposed platform uses a Myricom 10G-PCIE-8B-C Ethernet card. The Ethernet card communicates with the PC of the data monitoring system (in LWA, MCS-DR) through a PCIe interface (see Section 2.6.2) and is connected to the synthetic data generator (in LWA, DP system) through a CX4 cable.

Networking Software: The Ethernet card provides the hardware necessary for data transfer. Apart from this, the computer on which the Ethernet card is installed should have software that controls and communicates with the hardware in order to establish a connection to perform a data transfer. The software is typically written in high-level languages such as C, C++, or Python that support socket programming. Socket programming refers to the use of a set of functions for sending and receiving data over a network.

2.6.2 PCIe Interface

PCIe is a standard for computer expansion cards and is used to connect personal computers to peripheral devices attached to them. PCIe makes use of high speed serial link technology and suits the requirements of advanced processors and I/O technology. PCIe offers advantages over PCI, the predecessor of PCIe, in terms of increased speed, bandwidth and scalability. Unlike PCI, PCIe provides a point-to-point full duplex link dedicated to each device. This means that every device has a link of its own (dedicated) that can transmit and receive at the same time (full duplex), without interfering with other devices (point-to-point). These links are called lanes. Each lane transports data between endpoints of a link in packets of 8 bits. The number of lanes between two PCIe devices can be 1, 2, 4, 8, 12, 16 or 32. As the number of lanes increase, the data load is distributed among all the lanes of equal speed resulting in higher data transfer rates. Low speed devices require fewer lanes than the high speed devices.

Every communication interface follows a protocol in order to transfer data. The PCIe protocol is structured in three layers: physical, data link, and transaction layers. The PCIe lanes, which determine the physical connection between two PCIe devices, constitute the physical layer. Each lane consists of two pairs of differential signals, one each for transmitting and receiving data. The current technology PCIe lanes can transfer data at the rate of 250 MB/s per lane in either direction. A PCIe slot in which a peripheral is plugged into (interface) is shown in Figure 2.10.

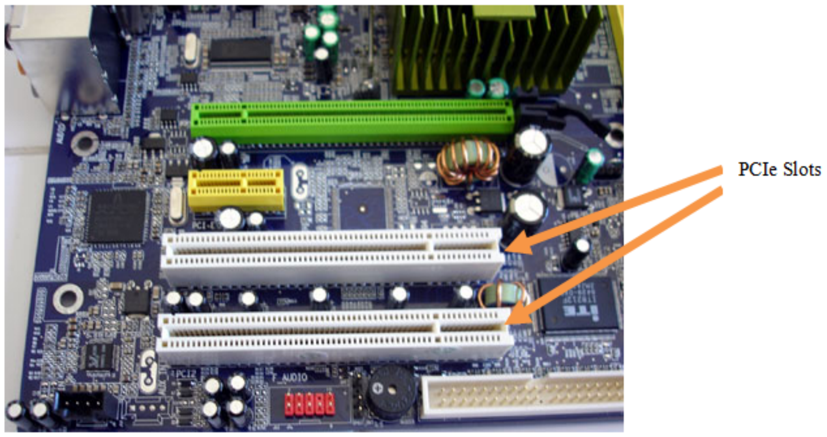


Figure 2.10: PCIe slot on the motherboard of a computer.

The data link layer makes sure that the data packets are reliably transferred across the PCIe link. The transaction layer receives the read/write requests and creates responses to the request. The requests received may correspond to PCIe configuration or data transfer between the host computer and the connected device. A PCIe device responds either with data, or data acknowledgement, or both.

PCIe compatible devices are plugged into the PCIe slot of a personal computer. Data transfer between the host computer and a PCIe device takes place as shown in Figure 2.11. The computer, during its boot up, identifies and initializes all the PCIe devices by allocating system resources such as memory and I/O space. This makes the device available for data transfer. Data can then be transferred to and from the PCIe device using software programs written in high-level languages. These applications communicate to the PCIe device through device drivers (low-level software programs). Device drivers act as translators between the hardware and the high-level PCIe software programs. The PCIe software program configures the PCIe for the type and size of data transfer, initiates a transfer and performs data transfer by providing/accepting data to/from buffers allocated by the OS during initialization.

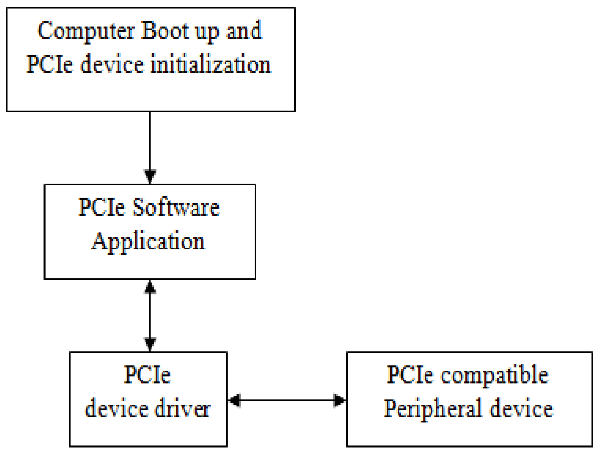

Computer Boot up and PCIe device initialization
PCIe Software Application
PCIe device driver
PCIe compatible Peripheral device

Figure 2.11: Data transfer between host computer and a PCIe device.

Chapter 3

FPGA Development

This chapter provides information necessary to develop algorithms on Xilinx FPGAs. Section 3.1 gives an overview of the architecture of a Xilinx FPGA and Section 3.2 discusses the steps involved in the development of algorithms on FPGAs,

3.1 Field Programmable Gate Arrays

FPGAs are re-configurable digital devices available in the form of integrated circuits (ICs) that can be programmed to implement digital functions. FPGAs stand in between generic microprocessors and application-specific ICs (ASICs) in terms of performance and cost. In microprocessors, designs adapt to the existing hardware where as in FPGAs and ASICs, underlying hardware adapts to the design. This hardware adaptability produces high-speed models of a design and plays an important role in applications which demand high performance. Though FPGAs and ASICs seem similar with respect to hardware adaptability, they differ in terms of architecture and re-configurability. FPGAs are user-configurable devices capable of modelling multiple designs as opposed to ASICs which are customized devices, manufactured for a specific design. This means that, on FPGAs, users can modify, update and test designs any number of times at a lower cost unlike rigid and expensive ASICs, fabricated for a specific user design by a manufacturer. Due to their reconfigurable nature, FPGAs are used in rapid prototyping of ASICs and in performance critical, high-speed applications where dynamic changes in the designs may be necessary. FPGAs are widely used in the fields of digital signal processing, software-defined radios, image and speech processing, and radio astronomy.

3.1.1 FPGA Features

FPGAs have a grid-like structure with logic elements that can be connected together to realize a digital circuit as shown in Figure 3.1. The basic elements of a FPGA include Configurable Logic Blocks (CLBs), Input Output Blocks (IOBs), clock management resources and routing channels [11]. Modern FPGAs also have on-chip memories and dedicated resources for DSP functions. Xilinx is among the leading FPGA manufacturers and the features of Xilinx FPGA are briefly described below.

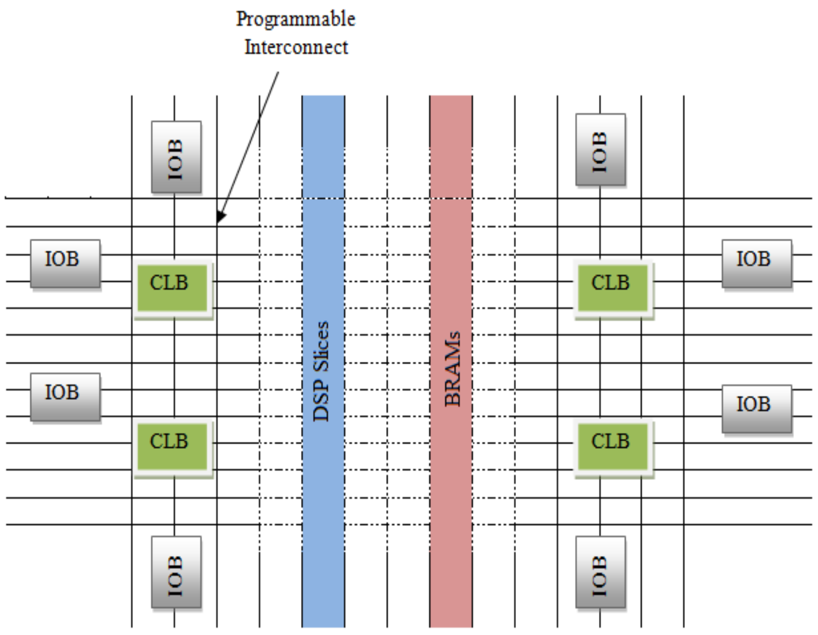


Figure 3.1: Block diagram of a Xilinx FPGA.

Configurable Logic Blocks (CLBs): The basic logic elements of FPGAs are called CLBs. A CLB consists of memory elements (generally SRAMs) which function as Look Up Tables (LUTs), selection circuitry (multiplexers), flip-flops, arithmetic gates. LUTs accepts binary inputs (usually 4 or 6) and can implement a Boolean function stored in the form of a truth table. Each CLB is capable of implementing combinatorial logic, shift registers or RAM. By connecting together various CLBs, it is possible to realize designs with complex functionality.

Routing and I/O resources: Connections between CLBs are made through programmable switch-like routing resources. The grid-like structure of a FPGA is formed by the interconnects as shown in Figure 3.1. The routing resources can be programmed to connect various CLBs in such a way as to implement the desired functionality. Apart from CLBs interconnections, the routing resources also connect to the IOBs (Input Output Blocks). The IOBs support a variety of interface standards providing a flexible interface between the FPGA package pins and the configurable logic.

Memories and other resources: Most of the designs implemented on the FPGA interact with external devices such as a host computer or other processor chips. To support such designs, FPGAs have on-chip Block RAMs that can be used to store and buffer data. Modern FPGAs come with hard and soft IP (Intellectual Property) cores that can be integrated with a user design to expedite the design process. Hard IP cores are dedicated chunks of hard-wired logic present in an FPGA whereas soft IP cores refer to optimized logic that can be used to program a FPGA for the required functionality. Embedded processors, DSP slices, and gigabit transceivers are examples of hard IP cores available on modern FPGAs. A variety of soft IP cores are provided by FPGA vendors for implementation of various communication protocols (bus interfaces), FIFOs, DSP and math functions. Inside the FPGA, the dedicated resources (such as on-chip memories, hard IPs) are placed at fixed locations as shown in Figure 3.1.

This section discussed the basics of FPGA and how digital circuits are realized on it. From an FPGA user's perspective, it is important to know how the logic elements of FPGA are programmed and the skills needed to work with FPGAs. Programming FPGAs involves a series of steps as discussed in Section 3.2.

3.2 FPGA Design Flow

The FPGA design process involves a series of steps: design entry, synthesis, design implementation and device programming, as shown in Figure 3.2. Design verification is performed at various stages of the design flow.

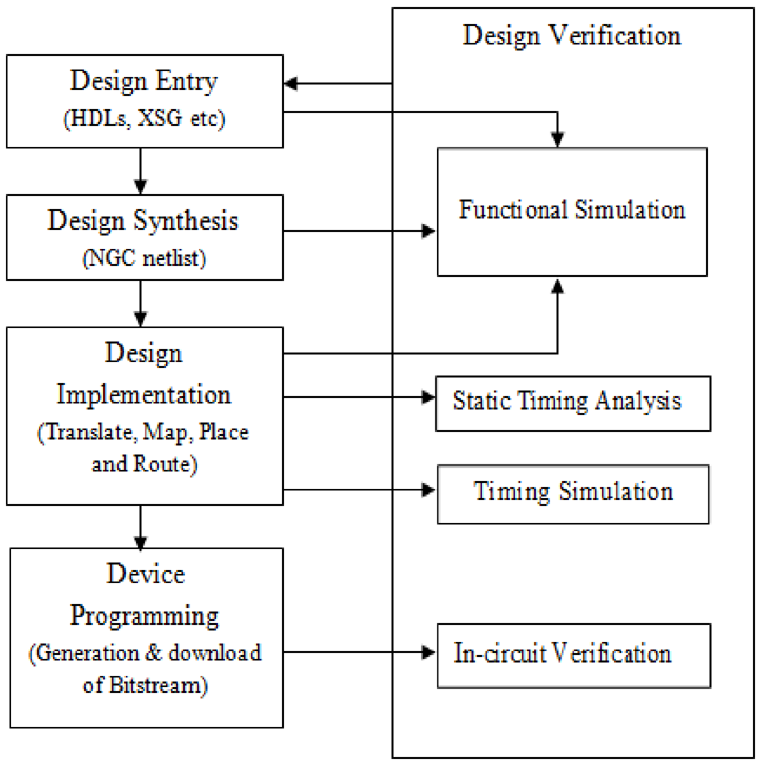


Figure 3.2: FPGA design flow.

FPGA vendors offer tools that integrate all these processes making it easier for the designer.

Xilinx provides an integrated FPGA development tool known as the Integrated Software Development (ISE) Design Suite to automate the design process [12]. A brief description of the design processes using Xilinx ISE is provided in the following sections.

3.2.1 Design Entry

Designs that are to be implemented on the FPGA are generally captured in Hardware Description Languages (HDLs) such as Verilog or VHDL. HDLs insulate designers from the details of hardware implementation. Some of the EDA tools allow design description in high-level software languages such as C and C++ which are eventually converted to HDL designs. Designs can also be captured using model-based tools such as the Xilinx System Generator (XSG) which convert schematic models to HDL descriptions.

XSG allows users with little or no HDL background to work with FPGAs. XSG uses MATLAB's Simulink tool to model designs by connecting hardware blocks together. Hardware blocks are IP cores or pieces of tested logic supplied by Xilinx. XSG relieves the designer from low-level algorithmic complexity and helps to implement designs. The XSG library has a set of DSP hardware blocks that can perform complex functions such as FFT, FIR filter design, or Viterbi decoding. XSG uses the Xilinx ISE design suite to automate HDL code generation which can then be integrated with other designs or used as a stand-alone design. Figure 3.3 gives design flow using XSG.

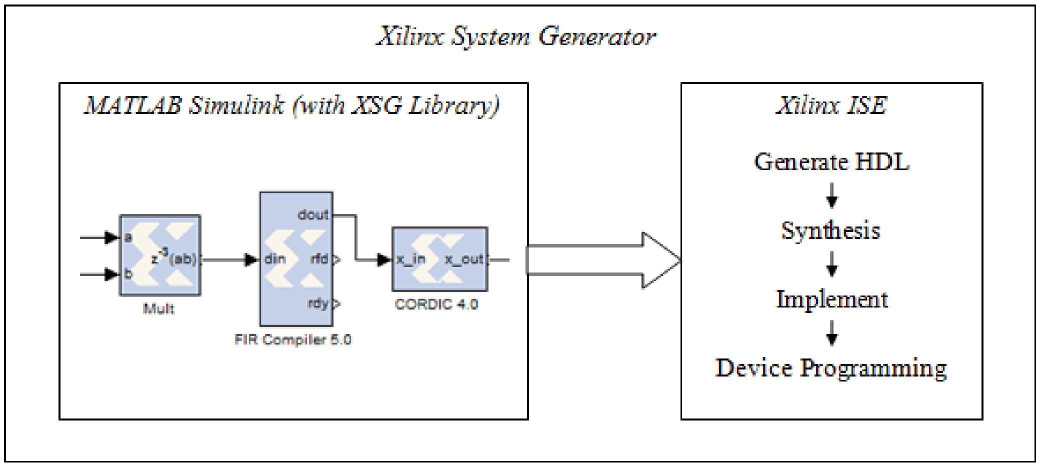


Figure 3.3: Design flow using Xilinx System Generator.

3.2.2 Design Synthesis

Synthesis tools take in HDL designs as inputs, check the code syntax, and create a Register-Transfer Level (RTL) description of the design. The RTL description is an optimized version of a design described in terms of basic logic gates. The XST (Xilinx Synthesis Technology) tool combines the RTL description with Xilinx-specific optimization and creates a file known as Native Generic Circuit (NGC) netlist. The NGC file contains the logical design data in terms of LUTs along with constraints specific to Xilinx FPGAs. The NGC file is passed on to the next step of design process, design implementation.

3.2.3 Design Implementation

Design implementation occurs in three steps: Translate, Map, and Place and Route. NGDBuild, MAP and PAR are Xilinx ISE programs that perform these design implementation steps. Figure 3.4 lists the three steps of design implementation along with the files generated at each step.

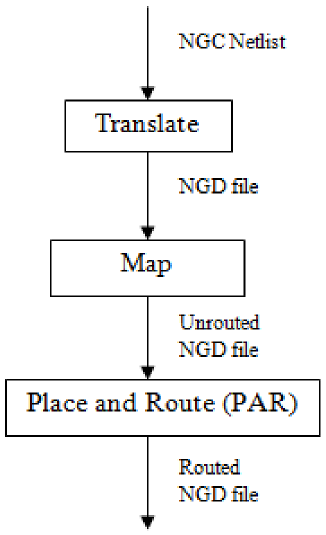


Figure 3.4: FPGA design implementation.

Translate: In the Translate phase, the netlists generated by the synthesis tool are combined together with a User Constraints File (UCF) to form a single logic design file. UCF contains information about the physical elements (such as pins) related to the packaging of the targeted FPGA obtained from the FPGA vendor. The translate tool assigns the input/output ports of a design to the physical elements of the FPGA. NGDBuild combines all the NGC files provided by the synthesis tool along the UCF provided by the user, to a single file called Native Generic Database (NGD) file.

Map: Mapping refers to fitting the design to the underlying FPGA architecture. The Map process divides the whole design into small logic elements and maps them to the logic blocks (such as CLBs, IOBs, hard IP cores) of the target FPGA. Xilinx's MAP program maps the design defined by NGD file to the target FPGA and produces a Native Circuit Description (NCD) file which is a physical representation of the design on the FPGA.

Place and Route: PAR places the logic functions of a design into the logic blocks of the target FPGA (similar to the MAP program) and also connects the logic blocks together. The MAP program maps the design into the available resources on the target device whereas PAR is an iterative process which places the logic onto the FPGA and routes connections until all constraints (such as timing and power) are satisfied. If the PAR tool fails to route the design or meet all constraints, an error is generated. In this case, design should be modified accordingly and the whole design process is repeated. The result of the PAR tool is an optimized (with respect to area, performance and power) and completely routed NCD file which meets all the constraints.

3.2.4 Device Programming

The NCD file generated by the PAR tool has all the necessary placement and routing information for the implementation of the design on the selected target FPGA. This should be converted to a format that can be used to program the FPGA. The BITGEN program converts the routed NCD file to a bitstream (a file with a ".bit" extension). Using a programming cable, the bitstream can be downloaded from the host computer to the FPGA to configure it for the design. The iMPACT tool allows the selection of a configuration mode and aids the configuration download process.

3.2.5 Design Verification

Design verification refers to testing of the design for functional correctness and performance requirements. The verification process can be broadly categorized into three steps: simulation, static timing analysis, and in-circuit verification. Simulation of a design is performed by simulators such as ModelSim. Functional simulation checks the logical correctness of the design before it is implemented on a device. It is performed on HDL designs (pre-synthesis) or NCD files (post-translate) and helps in correcting the design at an earlier stage. Simulators also perform timing simulation. Timing simulation is performed after the design is implemented in order to verify the design speed under worst-case conditions.

Static timing analysis performs quick timing checks of a design after the MAP or the PAR process. It lists the delays derived from the design logic and routing to help evaluate timing performance. Once the design is tested for functional and timing correctness, the bitstreams can be downloaded on to the FPGA and verified for in-circuit operation.

Chapter 4

System Implementation

In this thesis, a prototype for the FPGA-based data monitoring system for LWA is developed. This chapter discusses the implementation aspects of the prototype data monitoring system. The details of the hardware and the communication interfaces which constitute the proposed system are discussed in Section 4.1. The implementation of the system, which involves establishing communication links (through the Ethernet and the PCIe interfaces) and configuring the PC and FPGA for data transfer is described in Section 4.2.

4.1 Hardware Setup

As shown in Figure 4.1, the hardware used to implement the prototype data monitoring system includes:

1. A host computer with a quad-core 2.66 GHz Core i7-920 CPU,

2. A Myricom 10G-PCIE-8B-C Ethernet Card and a 10GbaseCX4 Ethernet cable,

3. XUPV5 development board with a programming cable.

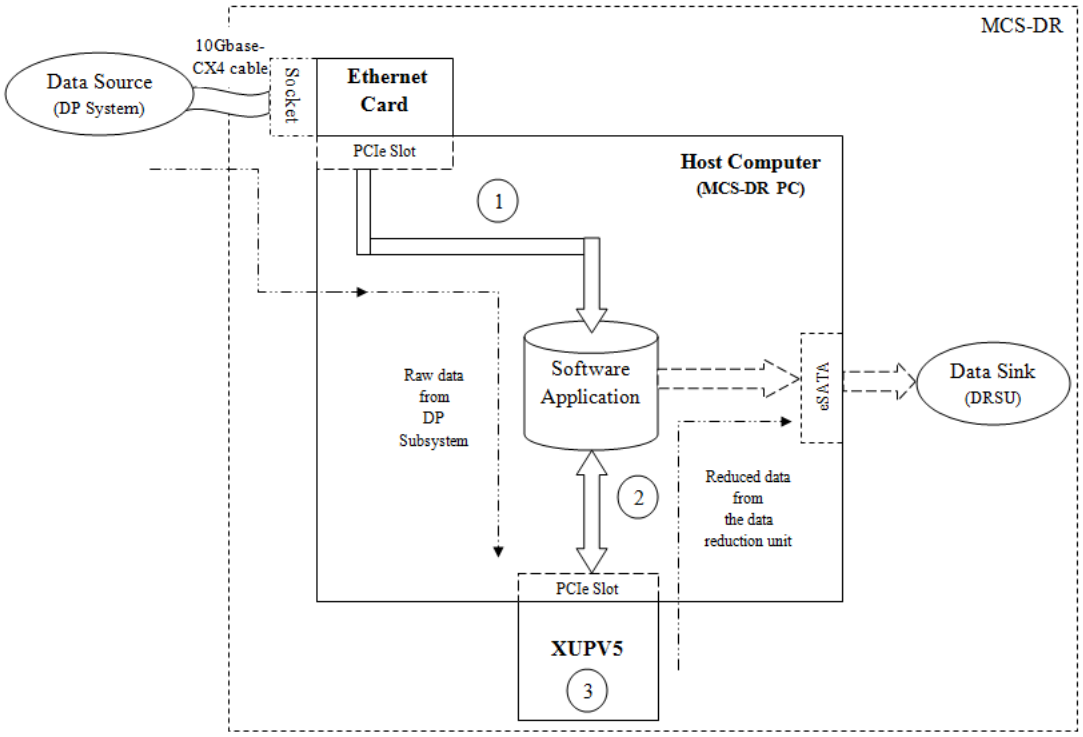


Figure 4.1: Components of the data monitoring system.

The monitoring system is inserted between the data source and the data sink as shown in Figure 4.1. Data from the data source (DP subsystem) is routed to the Ethernet card via a 10GbaseCX4 cable. The Ethernet card, inserted into the host computer (MCS-DR PC) via a PCIe slot, transfers the output of the data source to the software process running on the host computer [4]. The software process accepts data from the Ethernet card (DP subsystem) and routes it to the XUPV5. The XUPV5 board is connected to the host computer through another PCIe slot and monitors the data obtained from the data source. The output data from the XUPV5 board is then transferred back to the software process which routes it to the data sink (DRSU). Thus, the host computer acts as a data manager and routes data between the appropriate units.

MCS-DR system is an already existing system in an LWA station which routes the data obtained from the DP system directly to the DRSU [4]. In this thesis, an XUPV5 board is attached to the MCS-DR in order to evaluate FPGAs to implement data monitoring algorithms in an LWA station.

In order to test the prototype monitoring system, another computer with a quad-core 2.66 GHz Core i7-920 CPU is used to generate synthetic DP subsystem output. The synthetic data generator computer is connected to the host computer (Ethernet card) using a 10GbaseCX4 cable. Both the computers have the Ubuntu 8.10 64-bit Linux operating system installed on them. The next section discusses in detail the implementation of the data monitoring system.

4.2 Data Monitoring System

As shown in Figure 4.1, implementation of the FPGA-based data monitoring system consists of three main parts.

(1) Communication through the 10GbE interface,

(2) Communication through the PCIe interface,

(3) Data monitoring framework in the XUPV5 board.

Communication through the 10GbE interface is controlled by the software process running on the host computer. Communication through the PCIe interface requires the installation of the PCIe driver on the host computer and the necessary PCIe core logic on the FPGA. The software process running on the host computer initiates a DMA transfer, and the hardware process (on the XUPV5 board) responds to the commands from the host computer. Data monitoring is performed by the FPGA and is entirely a hardware process. Sections 4.2.1, 4.2.2 and 4.2.3 discuss the data monitoring system implementation in detail. Section 4.3 gives an overview of the data transfer process in the data monitoring system in terms of software and hardware processes.

4.2.1 Communication Through the 10GbE Interface

An Ethernet socket is configured by the software process on the PC using the socket programming functions defined in the ANSI C library. The `socket()` and `bind()` functions are used to create a Internet Protocol version 4 (IPv4) UDP receive socket on the PC [13]. The `recvfrom()` function puts the receive socket into a listen mode where the socket waits for data. When the socket detects incoming data directed towards it, the receive buffer is updated with the received data. The receive buffer is then used to update the transfer to the data monitoring unit (FPGA). In this thesis, the required device driver for the Ethernet card has been developed and provided by Myricom. The next step discusses in detail the data transfer process through the PCIe link.

4.2.2 Communication Through the PCIe Interface

The necessary code required for the communication between the PC component of the data monitoring system and the XUPV5 board is provided by Xilinx's xapp1052 application [14]. In this thesis, the code provided by Xilinx, which includes the software driver, the PCIe software program and the Bus Master DMA engine (BMD), are collectively referred to as the PCIe framework.

Communication between the host computer and the XUPV5 board is controlled by the PC component of the data monitoring system and is performed by the BMD on the XUPV5 board. This section discusses the PCIe framework (provided by Xilinx) required to establish the connection and transfer data between the host computer and the XUPV5 through the PCIe link. As shown in Figure 4.2, the PCIe framework includes three main components:

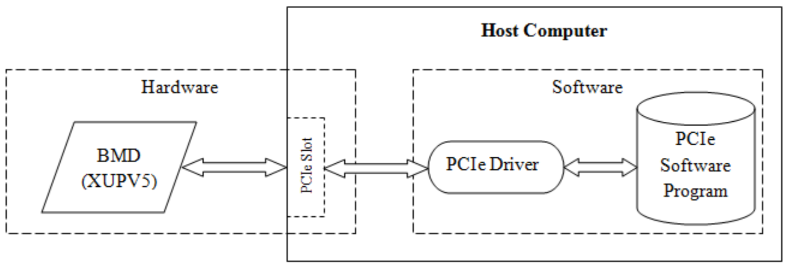


Figure 4.2: Components of the PCIe framework.

a. PCIe driver installation: The PCIe driver for the XUPV5 board is obtained from Xilinx through the xapp1052 application [14]. The driver is a low-level software program (written in C) installed on the host computer that links the higher level PCIe software program on the computer to the data monitoring framework on the XUPV5 board. The driver contains various routines called by the PCIe software program which are used to communicate with the hardware via the PCI Express link. After the host computer detects the XUPV5 device, the PCIe driver is installed for the XUPV5 board as per the instructions mentioned in [14]. The driver first gets the base address of the hardware and maps the hardware bus memory to the system (host computer) memory. The operating system then allocates memory resources (data buffers) and loads the modules needed for PCIe link configuration and data transfer. After registering the device, the driver is loaded after proper hardware initialization and buffer allocation. The installed driver resides in the kernel memory of the host computer. The XUPV5 board is now ready to transfer data to and from the host computer.

b. PCIe software program: The PCIe software program is a C program provided through the xapp1052 application which invokes routines in the driver to perform the necessary data movements. The PCIe software program reads the PCIe device configuration space and updates descriptor registers in order to check the status of the PCIe device and initiate data transfers. The configuration space is the common memory space between the host computer and the XUPV5 board that has information about the capabilities of the PCIe link such as link speed, link width, and link control. The descriptor registers are used to set up a transfer and to initialize the BMD design with user-specified information regarding data transfer such as the payload, size and count of data packets, and read/write enable from/to the XUPV5 board [14]. The PCIe software program also manages and updates the write-to-XUPV5 and the read-from-XUPV5 endpoint data buffers.

The series of steps involved in making a data transfer to/from the XUPV5 board is shown in Figure 4.3. The PCIe software program first reads the PCIe device configuration space and gets the capabilities of the PCIe link. The PCIe software program then checks if the device (XUPV5) is ready to transfer data and updates various descriptor registers to set up a transfer. The write-to-XUPV5 data buffer is updated by data obtained from the DP subsystem via the Ethernet port. The XUPV5 board receives raw data and sends back the monitored data to the host computer at the same time. Hence, a two-way data transfer is initiated by updating the control register to enable reads and writes from/to the XUPV5 board. The control register is a descriptor register used to trigger the BMD to start a DMA transfer and indicate the status of a transfer. The contents of the descriptor registers are transferred to the BMD through programmable input/output (PIO) transfers which update the control and status registers of the BMD. On receiving instructions from the host computer (PCIe driver) to start the data transfer process, the BMD on the XUPV5 board initiates a DMA request to the host computer. The BMD then takes control of the PCIe bus and performs a DMA transfer to move data between the endpoint buffers and the system (host computer) memory. The completion of a DMA transfer is obtained by reading the contents of the control register using a PIO call to the FPGA.

c. Bus Master DMA engine (BMD) design: The BMD is a Verilog (HDL) design running on the Virtex-5 FPGA to implement the PCIe protocol, and controls the data transfer between the host computer and the data monitoring framework [14]. A PCIe link can transfer data at a theoretical maximum of 8000 MB/s (32 lanes) compared to the 132 MB/s theoretical maximum throughput of the PCI link. Thus, the evolution of PCIe has led to improved I/O bandwidth and enabled the use of external devices (such as the XUPV5) with various

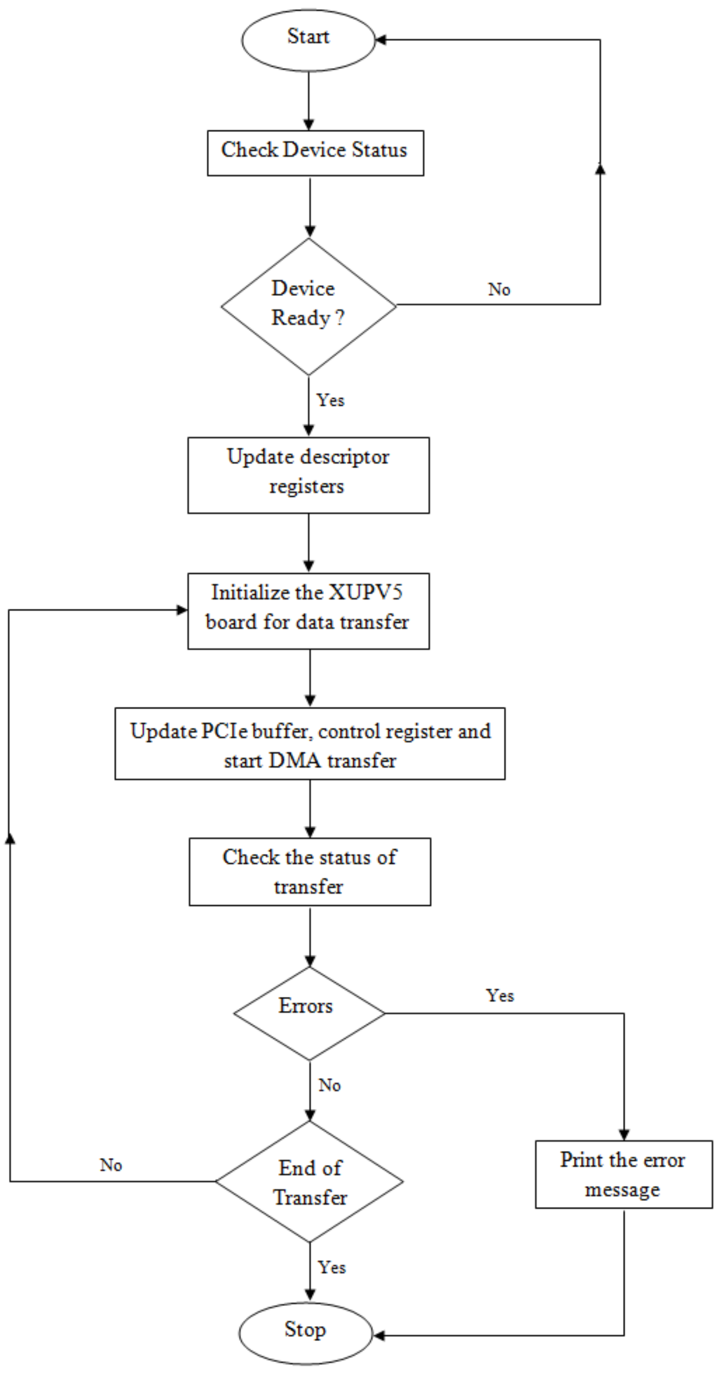

Start
Check Device Status
Device Ready ?
No
Yes
Update descriptor registers
Initialize the XUPV5 board for data transfer
Update PCIe buffer, control register and start DMA transfer
Check the status of transfer
Errors
Yes
No
End of Transfer
No
Yes
Print the error message
Stop

Figure 4.3: Flow chart for the PCIe software program.

capabilities along with PCs in high-speed data streaming applications.

The BMD design is developed over the Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs [15] which implements the PCIe protocol on the XUPV5. The Virtex-5 FPGA Integrated Endpoint block is a hard IP core embedded in the Virtex-5 FPGA that provides the full functionality of the transaction layer, the data link layer, the physical layer, and the configuration space as per the PCI Express Base 1.1 Specification [16]. The Endpoint Block connects to the PCIe Fabric through the transceivers [17] embedded in the Virtex-5 FPGA [15] as shown in Figure 4.4. The embedded Integrated Endpoint Block can be accessed through the LogiCORE IP Endpoint Block Plus v1.13 for PCI Express generated by the Xilinx's CoreGen tool [18].

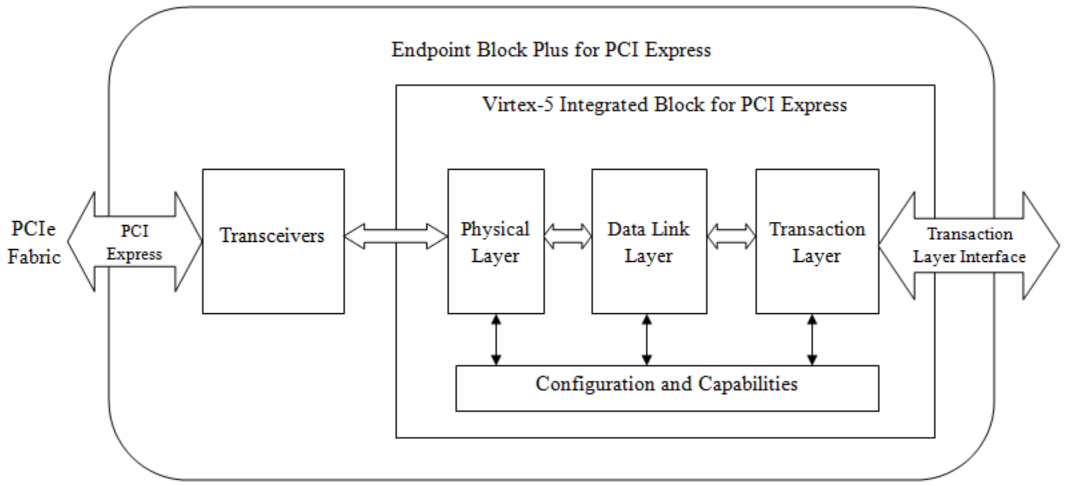

Endpoint Block Plus for PCI Express
Virtex-5 Integrated Block for PCI Express
PCIe Fabric
PCI Express
Transceivers
Physical Layer
Data Link Layer
Transaction Layer
Transaction Layer Interface
Configuration and Capabilities

Figure 4.4: Block diagram of the Integrated Endpoint Block Plus in Virtex-5 FPGA.

The Endpoint Block Plus is Verilog code provided by Xilinx which acts as a wrapper to the Integrated Endpoint Block as shown in Figure 4.4. The Endpoint Block Plus wrapper connects the transaction layer of the Integrated Endpoint Block to the rest of the BMD design which receives and transmits data at the transaction layer in the form of transaction layer packets (TLPs). The BMD control engines connect to the bridge platform and eventually to the data monitoring plug-in module as shown in Figure 4.5.

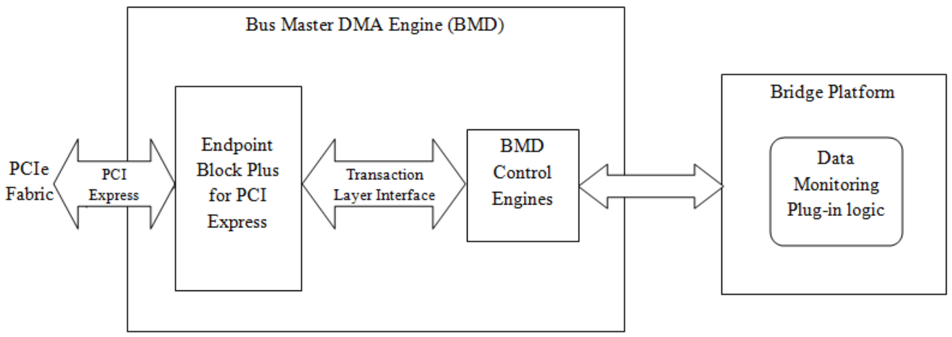

Bus Master DMA Engine (BMD)
PCIe Fabric
PCI Express
Endpoint Block Plus for PCI Express
Transaction Layer Interface
BMD Control Engines
Bridge Platform
Data Monitoring Plug-in logic

Figure 4.5: High-level view of the BMD architecture.

Apart from the Integrated Endpoint core, the BMD design contains control engines for the receive and transmit data path along with various registers and memory interfaces to store and retrieve data as shown in Figure 4.6. The BMD consists of target logic, control and status registers, and initiator logic as shown in Figure 4.6. The target logic captures the Memory Write (MWr) and Memory Read (MRd) TLPs sent to the endpoint via PIO which are used to monitor and control the DMA hardware.

The target logic updates the status and control registers with the contents of the descriptor registers discussed in the Section 4.2.2. The control and status registers contain operational information for the DMA controller about the link capabilities, start and stop of transfer, size and the count of TLPs, and status of the transfer. The initiator logic contains transmit and the receive engines, which generate MWr or MRd TLPs based on the upstream (endpoint to system memory) or the downstream (system memory to endpoint) transfer. A MWr TLP is generated by the transmit engine (TX engine) and consists of the DMA hardware address, TLP size, TLP count followed by data to be transferred to the system memory. A MRd TLP is a read request which has a similar pattern as the MWr TLP except for data. Data is received by the receive engine (RX engine) in response to a MRd request TLP. The read and write DMA control and status registers specify the address, size, payload content, and number of TLPs to be received/sent, to construct MRd and MWr TLPs. The data (raw data from the DP subsystem) received by the RX engine is transferred to the bridge platform, and the monitored data to be sent is provided by the bridge platform to the TX engine. In this thesis, a 64-bit transmit and receive PCIe links are used which can transfer 64 bits of data per PCIe clock.

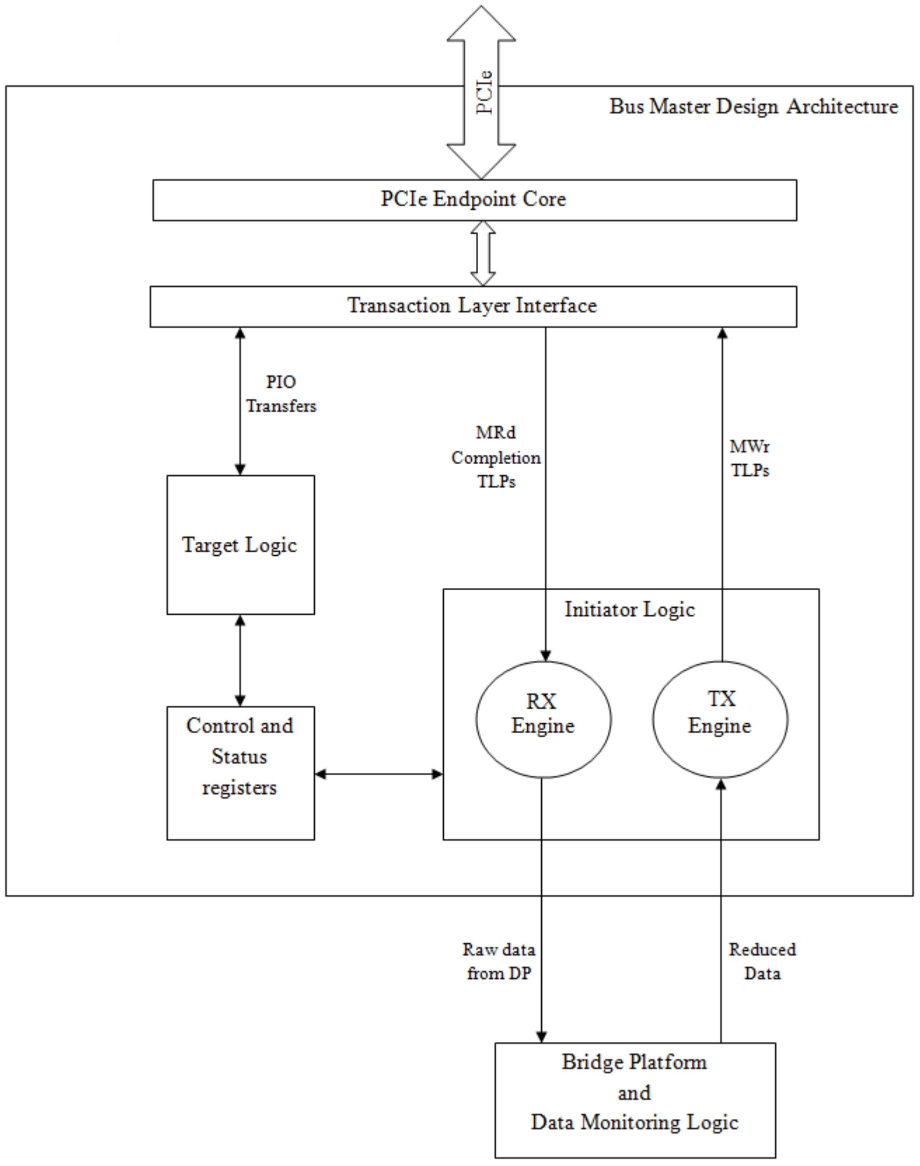
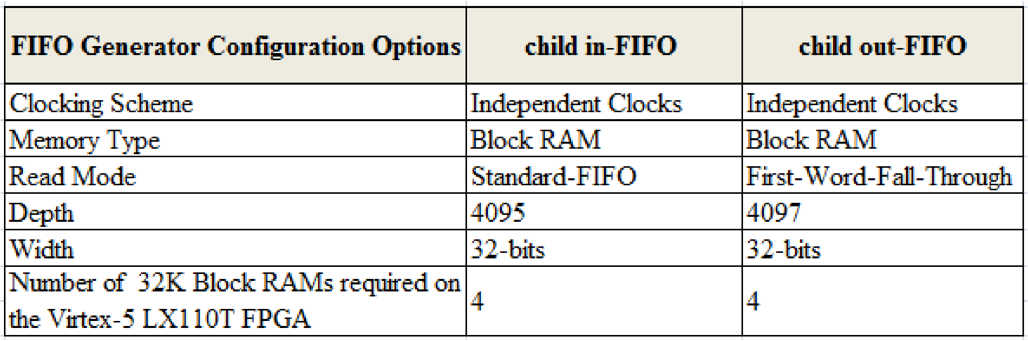


Figure 4.6: BMD architecture.

4.2.3 Data Monitoring Framework in the XUPV5 Board

Data monitoring is performed by the plug-in user module which is built over the bridge platform. The BMD provides incoming TLPs to the bridge platform. The bridge platform extracts data from the TLPs and routes it to the data monitoring plug-in logic. The user-defined plug-in processes the incoming data from the bridge platform and routes it back to the bridge platform. The bridge platform provides the monitored data to the BMD which forms outgoing TLPs and eventually transfers them to the host computer. The bridge platform and the data monitoring plug-in modules are described below:

a. Bridge Platform: The bridge platform, written in Verilog, performs data buffering using FIFOs and provides a plug-in capability for data monitoring logic through the FIFO interface and the data monitoring wrapper logic. The FIFO interface logic is provided in Appendix A.1 and the data monitoring wrapper code is provided in Appendix A.2. The bridge platform uses two FIFOs, one for buffering data received through the RX engine of the BMD (in-FIFO), and one for buffering data to be transmitted to the TX engine of the BMD (out-FIFO). The FIFOs are generated by Xilinx's CoreGen tool [19]. Both the generated FIFOs use independent clock Block RAM implementations. The in-FIFO uses a standard FIFO read mode where as the out-FIFO uses a first-word-fall-through read mode which are available in the FIFO generator core. Each of the parent FIFOs (in-FIFO and the out-FIFO) is constructed from two FIFOs, called child-FIFOs in this thesis, which are half the depth of the parent FIFOs with a data width of 32 bits each. The FIFO generator core configuration options are listed in Table 4.1. FIFOs of different capacity can be used with the design with modification made to the in-FIFO and out-FIFO instantiations and inserting the NGC file generated for the new FIFOs.



FIFO Generator Configuration Options	child in-FIFO	child out-FIFO
Clocking Scheme	Independent Clocks	Independent Clocks
Memory Type	Block RAM	Block RAM
Read Mode	Standard-FIFO	First-Word-Fall-Through
Depth	4095	4097
Width	32-bits	32-bits
Number of 32K Block RAMs required on the Virtex-5 LX110T FPGA	4	4

Table 4.1: Configuration options chosen for the FIFO generator core.

The transmit and receive PCIe links used in this thesis are 64-bits wide. The TX and the RX engines used operate on 32-bits data (if the other 32-bits of the PCIe link are used for headers) or 64-bit data. The TX and the RX engine of the PCIe framework are modified to assert appropriate enable signals for selecting one child-FIFO (for 32-bits of data in one PCIe clock) or both the child-FIFOs (for 64-bits of data in one PCIe clock) in every clock where data is available. Integrating two 32-bit data width child-FIFOs to form the in-FIFO and the out-FIFO will provide a 32-bit/64-bit input interface of the in-FIFO and the out-FIFO to the RX and TX engines respectively, and a 64-bit interface to the data monitoring module.

The write to the in-FIFO and the read from the out-FIFO are synchronized to the BMD clock with a 32-bit/64-bit data interface, while the read from the in-FIFO and the write to the out-FIFO are synchronized to the clock of the data monitoring logic with a 64-bit data interface. The FIFO interface logic routes data between the data monitoring logic and the FIFOs by asserting appropriate enable signals to the FIFOs and the monitoring logic. The data monitoring algorithm varies based on the user's requirement. Hence, in order to provide a flexible data monitoring platform, a data monitoring wrapper is developed to provide the plug-in capability for the user-defined data monitoring logic. The data monitoring wrapper instantiates the user-defined data monitoring module and connects the data monitoring module to the FIFO interface which in turn connects to the BMD. Thus, the bridge platform acts as an interface between the BMD and the data monitoring logic.

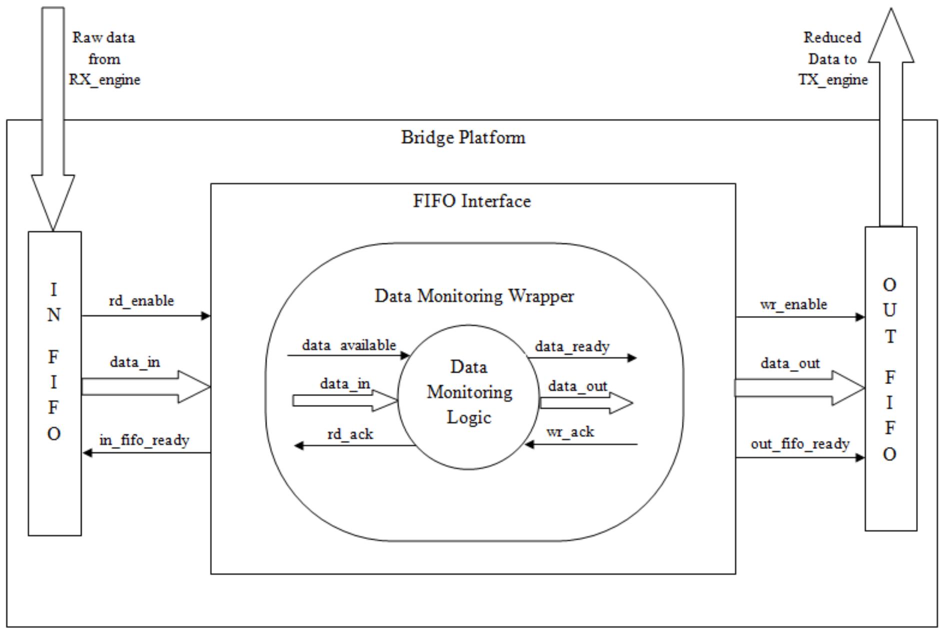


Figure 4.7: Bridge Platform.

b. Data Monitoring Plug-in: The data monitoring logic is developed by the users of the data monitoring system based on the requirements of the application for which the system is used. The data monitoring logic should be specified as an HDL module or as a NGC netlist. They can either be developed in HDLs or using model-based tools such as XSG which can generate HDL code or an NGC netlist for the user-defined XSG models. The input/output ports of the user-developed, top-level data monitoring module should match the ports of the data monitoring wrapper. A top-level Verilog module and a top-level XSG model are provided to users which can be used to match the data monitoring module to its instantiation in the data monitoring wrapper. Signals inform the data monitoring logic when the FIFOs are ready. The data monitoring logic then processes the incoming data from the in-FIFO and provides monitored data to the out-FIFO. Apart from the data processing, the data monitoring logic should also respond back to the FIFO interface by asserting acknowledge signals for the reads from the in-FIFO and the writes to the out-FIFO respectively.

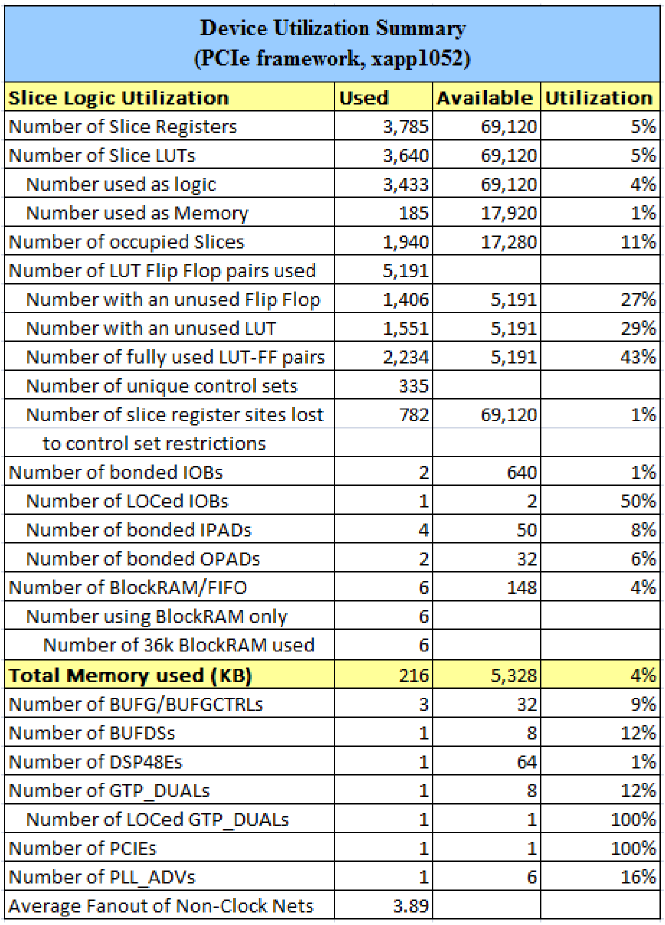
To demonstrate the plug-in feature, DSP functions such as FIR filter and FFT have been developed using XSG. XSG uses a model-based development environment and generates a netlist using Xilinx ISE tools. The netlist is used along with the data streaming framework developed in Verilog in order to generate a bitstream for the entire design. The Virtex-5 FPGA is then programmed with the generated bitstream to perform the functionality specified by the data monitoring plug-in on the input data. An 8-point FFT and a FIR filter of order 21 are developed using XSG. These blocks are developed and plugged into the design in order to test the plug-in capability of the data monitoring system and do not represent real data monitoring algorithms. Data monitoring logic should be developed by the user of the of the data monitoring system.

4.2.4 Modifications made to the PCIe framework

In this thesis, FIFO enable signals are added to the TX and RX control engines of the PCIe framework provided by Xilinx. Also, the state machine of the TX engine is modified in order to have blocked write to FPGA triggered by the amount of data available in the out-FIFO. The PCIe software application and the BMD are modified in order to add descriptor registers that communiate the status of the FIFOs (data count and buffer overflow). The PCIe driver is also modified by adding an extra set of PCIe buffers in order to enable swapping of PCIe buffers. While one DMA transfer is in progress using one set of read and write buffers, the software process operates on the other set of buffers by updating the write-to-FPGA buffer for the next transfer and extracting data from the read-from-FPGA buffer obtained from previous transfer.

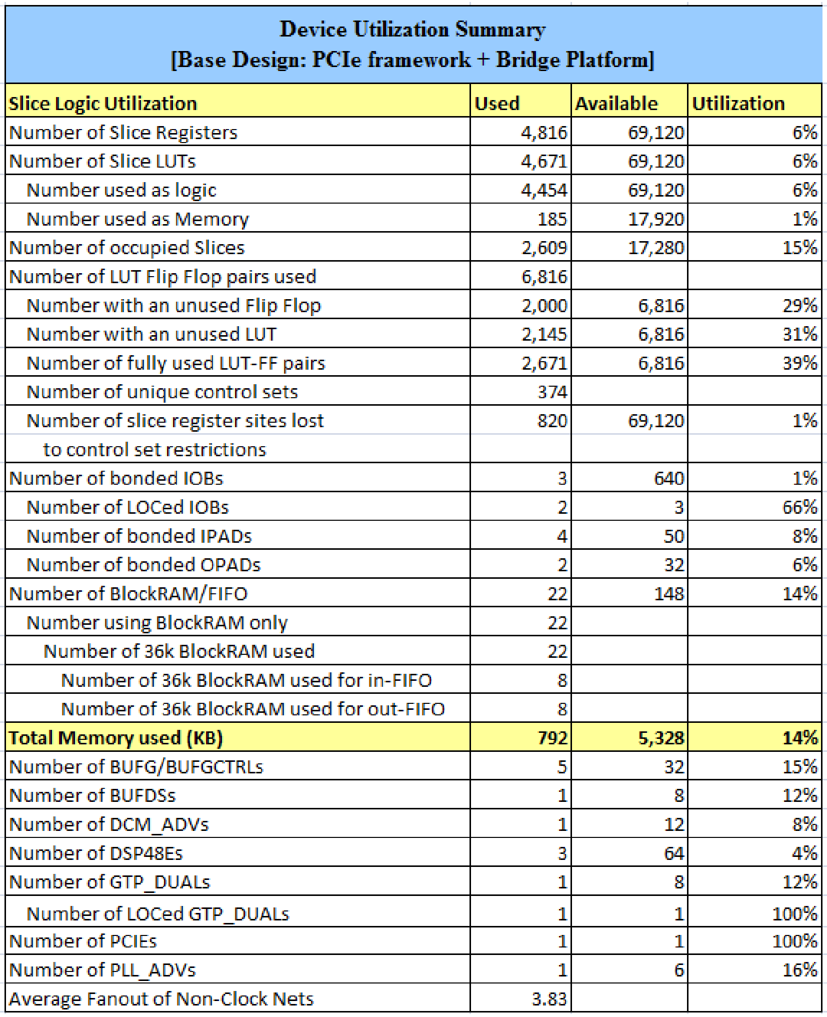
4.2.5 FPGA Resource Utilization

Table 4.2 gives the resources consumed by the PCIe framework on the FPGA (default design, xapp1052, provided by Xilinx). Table 4.3 gives the resource utilization for the base design developed in this thesis which includes the PCIe framework and the bridge platform along with the FIFOs used and does not include any data monitoring plug-in logic. The in-FIFO which uses a standard read mode has a depth of 8190 and a width of 32-bits whereas the out-FIFO with a first-word-fall-through read mode has a depth of 8194 and width of 32-bits (see Section 4.2.3). The FIFO generator core configuration options are listed in Table 4.1. More information about FIFOs can be found in [19].



Device Utilization Summary (PCIe framework, xapp1052)			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	3,785	69,120	5%
Number of Slice LUTs	3,640	69,120	5%
Number used as logic	3,433	69,120	4%
Number used as Memory	185	17,920	1%
Number of occupied Slices	1,940	17,280	11%
Number of LUT Flip Flop pairs used	5,191		
Number with an unused Flip Flop	1,406	5,191	27%
Number with an unused LUT	1,551	5,191	29%
Number of fully used LUT-FF pairs	2,234	5,191	43%
Number of unique control sets	335		
Number of slice register sites lost to control set restrictions	782	69,120	1%
Number of bonded IOBs	2	640	1%
Number of LOCed IOBs	1	2	50%
Number of bonded IPADs	4	50	8%
Number of bonded OPADs	2	32	6%
Number of BlockRAM/FIFO	6	148	4%
Number using BlockRAM only	6		
Number of 36k BlockRAM used	6		
Total Memory used (KB)	216	5,328	4%
Number of BUFG/BUFGCTRLs	3	32	9%
Number of BUFDSs	1	8	12%
Number of DSP48Es	1	64	1%
Number of GTP_DUALs	1	8	12%
Number of LOCed GTP_DUALs	1	1	100%
Number of PCIEs	1	1	100%
Number of PLL_ADVs	1	6	16%
Average Fanout of Non-Clock Nets	3.89		

Table 4.2: FPGA resource utilization for the PCIe framework provided by Xilinx.



Device Utilization Summary [Base Design: PCIe framework + Bridge Platform]			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	4,816	69,120	6%
Number of Slice LUTs	4,671	69,120	6%
Number used as logic	4,454	69,120	6%
Number used as Memory	185	17,920	1%
Number of occupied Slices	2,609	17,280	15%
Number of LUT Flip Flop pairs used	6,816		
Number with an unused Flip Flop	2,000	6,816	29%
Number with an unused LUT	2,145	6,816	31%
Number of fully used LUT-FF pairs	2,671	6,816	39%
Number of unique control sets	374		
Number of slice register sites lost to control set restrictions	820	69,120	1%
Number of bonded IOBs	3	640	1%
Number of LOCed IOBs	2	3	66%
Number of bonded IPADs	4	50	8%
Number of bonded OPADs	2	32	6%
Number of BlockRAM/FIFO	22	148	14%
Number using BlockRAM only	22		
Number of 36k BlockRAM used	22		
Number of 36k BlockRAM used for in-FIFO	8		
Number of 36k BlockRAM used for out-FIFO	8		
Total Memory used (KB)	**792**	**5,328**	**14%**
Number of BUFG/BUFGCTRLs	5	32	15%
Number of BUFDSs	1	8	12%
Number of DCM_ADVs	1	12	8%
Number of DSP48Es	3	64	4%
Number of GTP_DUALs	1	8	12%
Number of LOCed GTP_DUALs	1	1	100%
Number of PCIEs	1	1	100%
Number of PLL_ADVs	1	6	16%
Average Fanout of Non-Clock Nets	3.83		

Table 4.3: FPGA resource utilization for the base design (PCIe framework and Bridge Platform with FIFOs).

4.3 Data Transfer Process in the Data Monitoring System

Section 4.2 discusses the implementation of the data monitoring system. This section summarizes the data transfer between the XUPV5 board and the PC component of the monitoring system in terms of the software process (executed on the PC) and the hardware process (executed on the FPGA). The software and hardware processes are described next.

4.3.1 Software Process

The data streaming and monitoring process is controlled by a software process running on the PC. The software process updates various software buffers and routes data to and from external devices (such as the DP system and the DRSUs) attached to the PC. A flow chart for the software process is shown in Figure 4.8 and the data transfer loop is provided in Appendix A.3.

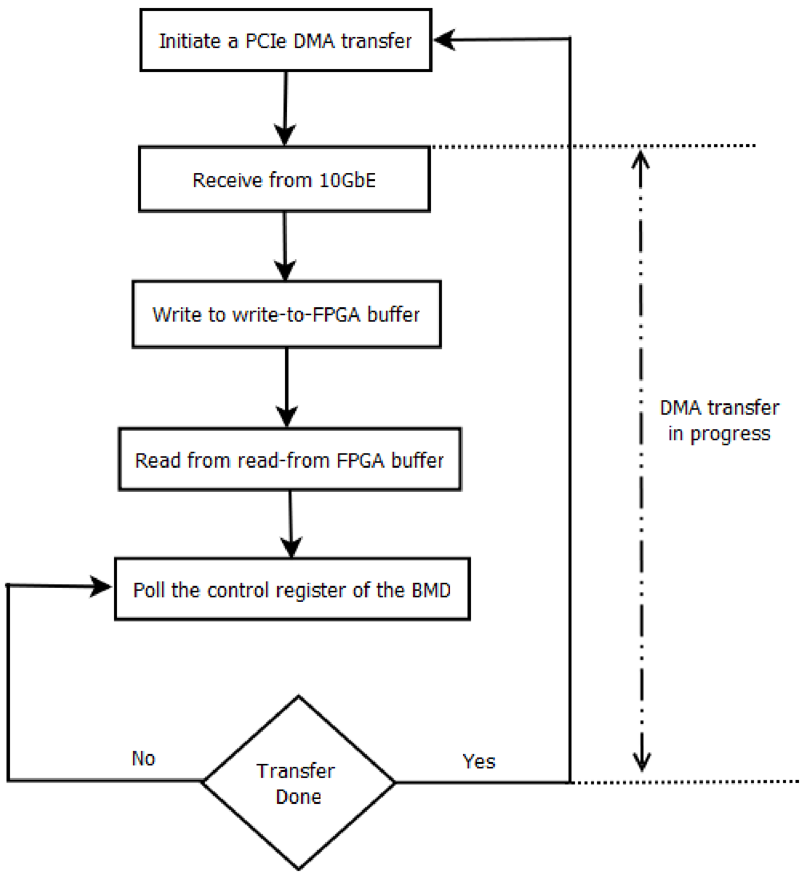

Initiate a PCIe DMA transfer
Receive from 10GbE
Write to write-to-FPGA buffer
Read from read-from FPGA buffer
Poll the control register of the BMD
Transfer Done
No
Yes
DMA transfer in progress

Figure 4.8: Flow Chart for the Software Process.

The software process first initiates a PCIe transfer to the FPGA. The FPGA then takes control of the bus and performs a DMA transfer of data between the FPGA and the PC. While the DMA transfer is in progress, the software process prepares the next transfer by updating:

(a) The receive-from-DP software buffer with the data obtained via the 10GbE link,

(b) The write-to-PCIe kernel buffer with the contents of the receive buffer that will be sent in the next transfer, and

(c) The transmit-to-DRSU buffer with the contents of the read-from-PCIe kernel buffer which was obtained from the previous transfer.

After updating the buffers, the process waits for the DMA transfer to end after which a new transfer is initiated. The status of the DMA transfer is obtained by polling the control register on the FPGA. Using interrupts instead of polling is a more efficient way of determining the end of the data transfer between the PC and the FPGA. Attempts to make use of interrupts to determine the end of transfer were unsuccessful. Exploring ways of implementing an interrupt-driven transfer is left to future work.

The software process is sequential and is not multi-threaded. The write-to-FPGA and read-from-FPGA buffers are updated once every transfer through blocking function calls. As there is only one FPGA per PC, the software process does not benefit form a multi-threaded implementation and a single-thread implementation using blocking calls works efficiently. For the receive from 10GbE function, the data received per transfer is routed to the FPGA. Moreover, the rate at which the data is received into the recv-from-10GbE buffer should be lower than the rate at which the data transfer happens in the FPGA (see Section 5.8.2 of Chapter 5). Thus, the software process done not benefit from multi-threading and hence the sequential single threaded implementation is sufficient.

4.3.2 Hardware Process

The hardware process (DMA transfer) occurs in parallel to the software process as shown in Figure 4.8. The hardware process has the PCIe framework, the bridge platform and the data monitoring plug-in module implemented on the FPGA executing in parallel. The data monitoring plug-in is connected to the PCIe framework through a data monitoring wrapper which isolates the data monitoring module from the PCIe protocol implementation.

The PCIe framework consists of the the xapp1052 application provided by Xilinx [14]. The xapp1052 application encompasses the PCIe core, which connects to the physical layer of the PCIe link, and provides a transaction layer interface through the TX and RX engines. The TX and RX engines regulate the flow of data to and from the FPGA via the PCIe link. In this thesis, a bridge platform is developed which connects to the TX and RX engines. The bridge platform buffers the data obtained from the RX engine and the data generated by the data monitoring plug-in which will be sent out of the FPGA through the TX engine. After a transfer is initiated on the FPGA, the data transfer process in the FPGA splits into two parallel processes each of which is controlled by the TX engine and the RX engine state machines.

TX and RX Engines: Section 4.2.2 discusses the role of TX and RX engines in the PCIe framework. Figure 4.9 describes the TX engine state machine. The TX engine sends the MRd REQs (MRd Requests) when in the MRd REQ state and the MWr REQs (MWr Requests) when in the MWr REQ state. The MRd REQ requests a TLP from the PC. The RX engine receives MRd CPLDs (MRd Completions with Data) as responses to the MRd REQs sent by the TX engine. The data obtained from the MRd CPLDs are transferred to the data monitoring module. Data generated by the data monitoring module is sent out of the FPGA to the PC through MWr REQ TLPs. A MWr REQ TLP is available when the data monitoring module has generated data equivalent to the size of one TLP. The MWr REQ state has a higher priority than the MRd REQ state. The state machine enters the MWr REQ state if there is a TLP available to be sent out of the FPGA, otherwise it enters the MRd REQ state.

This process is repeated until all the MRd and MWr REQs, specified by the transfer size from the software process, are sent out to the PC. Apart from the DMA transfer, the TX and RX engines also process PIO requests made by the PC. The RX engine receives the PIO read and write requests made by the PC. The data sent to the FPGA through a PIO write is used to update appropriate registers in the FPGA such as the control registers and the TLP size and count registers. In response to a PIO read request, the TX engine sends out data requested by the PC.

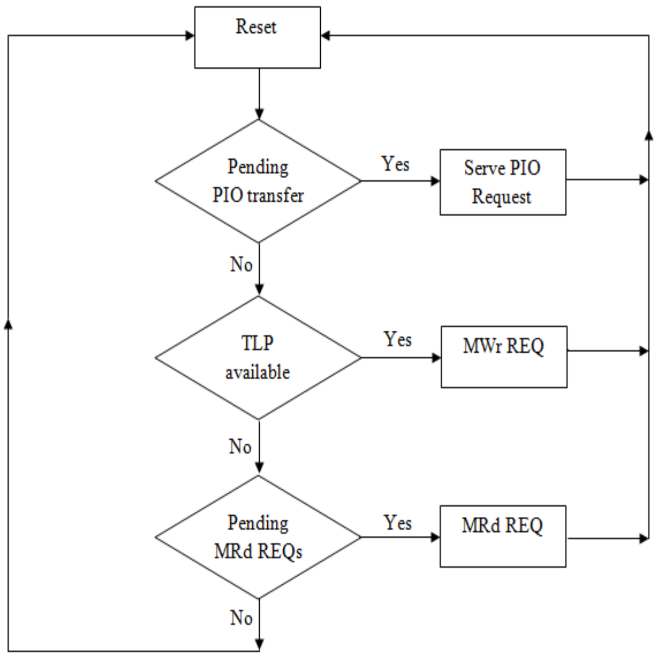


Figure 4.9: TX engine state machine.

The TX and RX engines are connected to the FIFO interface of the bridge platform developed in this thesis. The RX engine loads the received data into the in-FIFO whereas the TX engine fetches data to be transmitted from the out-FIFO. The FIFO interface is connected to the data monitoring wrapper which acts as a user interface between the PCIe framework and the data monitoring module. The data monitoring module obtains DP system raw data from the in-FIFO, processes it, and sends the monitored data to the out-FIFO which is eventually sent out to the PC by the TX engine.

4.4 Synthetic Data Generator

In order to test the prototype data monitoring system, a synthetic data generator has been developed. The data generator is basically a Ethernet transmitter that sends data to the prototype system through a 10GbE link. Similar to the configuration process discussed in Section 4.2.1, the `socket()` function creates a Ethernet transmit socket. The `sendto()` function transmits data to the receive socket identified by the IP address of the receive socket. In this thesis, the synthetic data generator is used merely as an input data source that transmits data via a 10GbE link and does not follow the format of the output of the DP system of the LWA station. In this thesis, the synthetic data generator outputs random data at a rate of 175 MB/s with a packet size of 4096 bytes. Controlling the output data rate of the synthetic data generator is performed by using a software timer [20]. A data rate of 175 MB/s is chosen in order to not limit the throughput of the data monitoring system (see Section 5.4) and a packet size of 4096 bytes is selected as the DP system of the LWA outputs data in packets of size 4096 bytes.

Chapter 5

Results and Analysis

This chapter discusses the factors affecting the throughput of the data monitoring system. Section 5.1 discusses the data transfer process in the FPGA and Section 5.2 lists the parameters of the data monitoring system. The throughput of the data monitoring system has been modelled based on the system parameters and the throughput equation is derived in Sections 5.3 and 5.4. Section 5.9 gives the measured values of throughput with respect to the system parameters and compares the measured throughput of the data monitoring system with the modelled values obtained from the equation derived in Section 5.4.

5.1 Analysis of the Data Transfer Process in the FPGA

The data transfer process within the FPGA is divided into three phases depending on the state of the TX engine.

Phase 1: After the transfer process is initiated, the TX engine starts sending MRd REQs. After a certain delay, the MRd CPLD response is received by the RX engine. The first part of the transfer process occurs until the arrival of the first MRd CPLD. The data monitoring module is inactive in this phase due to the lack of sufficient data in the in-FIFO. The out-FIFO is empty in this phase and hence no MWr REQs will be sent by the FPGA. Thus the first phase of transfer is the period of time for which only the MRd REQs are sent to the PC.

Phase 2: When the in-FIFO has sufficient data from the MRd CPLDs, the data monitoring module becomes active and starts filling the out-FIFO with processed data. The TX engine starts sending the MWr REQ state when the out-FIFO has data equivalent to the size of one TLP. As the MRd REQ state has a lower priority than the MWr REQ state, MRd REQs are sent to the PC when MWr REQ is not possible, i.e. when the out-FIFO does not have sufficient amount of data. In the second phase both the MRd and MWr REQ states are active.

Phase 3: The third phase is the period of transfer after all the MRd REQs are sent to the PC and only the MWr REQs are pending. This phase corresponds to the period of transfer where the MWr REQs are either waiting for data (in the out-FIFO) from the monitoring module or being sent one after the other. The end of the last MWr REQ marks the end of a transfer.

5.2 System Parameters That Affect the DMA Transfer in the FPGA

The amount of time spent at each phase of the data transfer process depends on the following system parameters measured in terms of PCIe clock cycles.

5.2.1 PCIe Transfer Parameters

The following PCIe parameters are set by the user based on the required transfer size:

- S_w is the number of 32-bit words in one TLP sent from the FPGA to the PC, and must be an even number.

- N_w is the total number of TLPs sent from the FPGA to the PC in one transfer.
- S_r is the number of 32-bit words in a TLP sent from the PC to the FPGA, and must be an even number.
- N_r is the total number of TLPs sent from the PC to the FPGA in one transfer.

Let B_w be the total number of bytes transferred from the FPGA to the PC in one transfer, and B_r be the total number of bytes sent from the PC to the FPGA in one transfer. We observe that

$$B_w = 4 \cdot S_w \cdot N_w$$

$$B_r = 4 \cdot S_r \cdot N_r$$

In this thesis, all experiments were performed by fixing S_w or S_r to 32, the maximum value supported by the selected hardware. As the value of S_w or S_r decreases, the overheads associated with every MWr REQ and MRd CPLDs increases which decreases the throughput of the entire system.

5.2.2 PCIe Framework Parameters

The following parameters are specific to the hardware (XUPV5 board) and the xapp1052 base design used in this thesis. P_f refers to the frequency of the clock used by the PCIe core and the PCIe framework, which is set to 62.5 MHz in the current design. P_T is the time period of the clock used by the PCIe core and the PCIe framework,

$$P_f = 62.5 \text{ MHz}$$

$$P_T = 1/P_f = 16 \text{ ns}$$

A 64-bit PCIe transmit link and a 64-bit PCIe receive link are used in this design which means that 64 bits of data can be transferred in one PCIe clock cycle. The number of clock cycles required to complete one MWr REQ is a function of S_w. The sequence of events to complete one MWr REQ is as follows:

(i) While in the reset state, check if a MWr REQ can be sent.

(ii) If a MWr REQ can be sent, transmit the first 64-bits of the MWr REQ header.

(iii) Transfer the remaining bits of the header and the first 32-bit word.

(iv) Transfer (S_w-2) 32-bit words at the rate of 64 bits per PCIe clock.

(v) Transfer the last 32-bit word and the 32-bit trailer and go back to the reset state.

Thus, the total number of clock cycles required for one MWr REQ is:

$$C_{mwr-req} = (S_w/2) + 2$$

The MRd REQ is independent of the transfer size and occurs in the following sequence:

(i) While in reset state, check if a MRd REQ can be sent.

(ii) If a MRd REQ can be sent, send the first 64-bits of the MRd REQ.

(iii) Transmit the remaining bits of the MRd REQ and go back to reset state.

Thus, the total number of clock cycles required for one MRd REQ is

$$C_{mrd-req} = 2$$

Irrespective of the size of the TLP, the MRd CPLDs are sent to the PC in batches of 64 bytes of data with a 32-bit header and a 32-bit trailer. The number of clock cycles to receive 64 bytes of data from the PC is

$$C_{mrd-cpld-64} = 9$$

After a transfer is initiated, the FPGA starts sending MRd REQs to the PC. Let C_{delay} refer to the number of clock cycles elapsed between the transmission of the first MRd REQ and the reception of the first MRd CPLD represented. It is observed that C_{delay} is a constant:

$$C_{delay} = 140$$

5.2.3 Bridge Platform Parameters

$BW_{in-inFIFO}$ represents the input bandwidth of the in-FIFO and is defined as the number of bytes loaded into the in-FIFO in one PCIe clock cycle. $BW_{out-inFIFO}$ represents the output bandwidth of the in-FIFO and is defined as the number of bytes unloaded from the in-FIFO by the data monitoring module in one PCIe clock cycle. $BW_{in-outFIFO}$ represents the input bandwidth of the out-FIFO and is defined as the number of bytes loaded by the data monitoring module into the out-FIFO in one PCIe clock cycle. $BW_{out-outFIFO}$ represents the output bandwidth of the out-FIFO and is defined as the number of bytes unloaded by the TX engine (MWr REQ state) from the out-FIFO in one PCIe clock cycle.

5.2.4 Data Monitoring Module Parameters

The data monitoring plug-in is a user-defined module which can be interfaced with the bridge platform through the data monitoring wrapper. The data monitoring wrapper provides a 64-bit input and a 64-bit output data interface to the data monitoring module. Data monitoring modules are processing blocks operating on data streams whose behaviour can be modelled using the following parameters [20]:

(i) *Load (l):* The number of clock cycles required to load the inputs into the data monitoring module. Due to the 64-bit interface to the data monitoring module, the number of bytes that can be loaded in one cycle is 8. The minimum value of l corresponds to the number of clock cycles required to load 8 bytes. Hence $l \geq 1$ since it takes at least one clock cycle to load 8 bytes.

(ii) *Process (p):* The number of clock cycles required to process the inputs to produce outputs, where $p \geq 0$.

(iii) *Unload (u):* The number of clock cycles required to unload data from the data monitoring module. Due to the 64-bit data interface to the out-FIFO, the number of bytes that can be unloaded in one clock cycle is 8. The minimum value for u corresponds to the number of clock cycles required to unload 8 bytes of data. Hence, $u \geq 1$ since it takes at least one clock cycle to unload data from the data monitoring module.

(iv) *Wait (z):* The number of clock cycles required for the data monitoring module to load the next set of inputs after loading the previous set of inputs. For parallel processing algorithms, z will be the negative of the sum of p and u since there will be no wait between loading of the inputs into the data monitoring module. Hence, $z \geq 1$ for serial algorithms, and $z = -(p + u)$ for parallel algorithms.

(v) *Bytes in* (B_{inDR}): The number of bytes consumed by the data monitoring module in one load process. Due to the 64-bit data interface between the in-FIFO and the data monitoring module, B_{inDR} is always a multiple of 8. Hence, $B_{inDR} = 8n$ where n is an integer greater than or equal to 1.

(vi) *Bytes out* (B_{outDR}): The number of bytes produced by the data monitoring module for one unload process. Due to the 64-bit data interface between the out-FIFO and the data monitoring module, B_{outDR} is always a multiple of 8. Hence, $B_{outDR} = 8m$ where m is an integer greater than or equal to 1.

For a sequential processing block, the above described processes occur one after the other as shown in Figure 5.1. For parallel algorithms, all the processes overlap as shown in Figure 5.2.

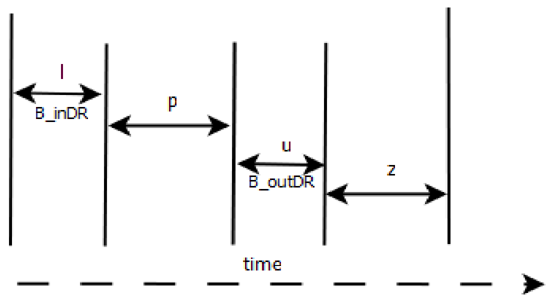


Figure 5.1: Sequential data monitoring algorithms.

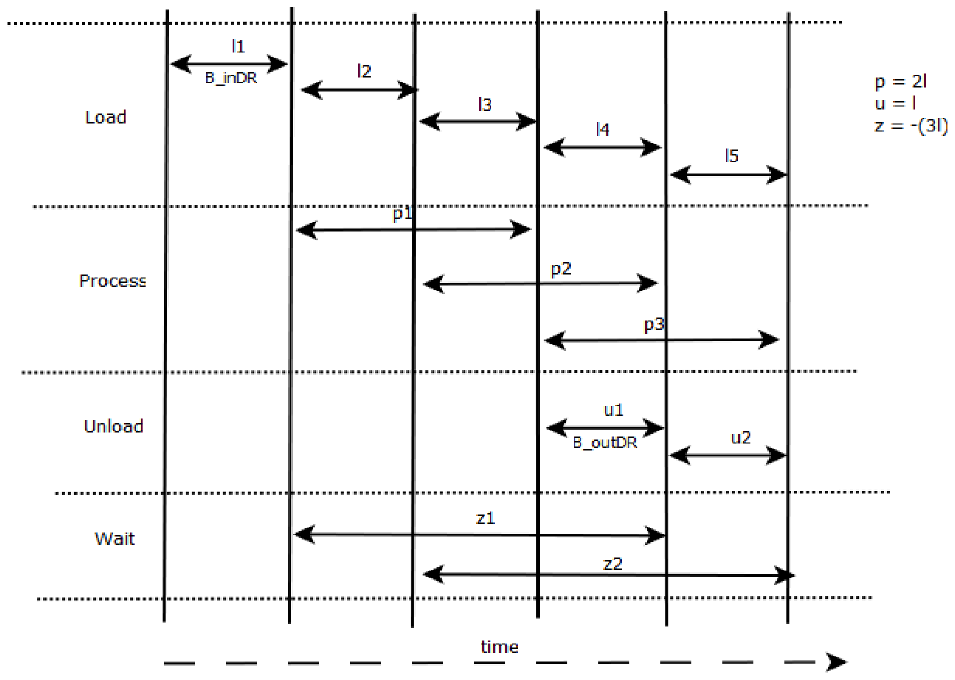


Figure 5.2: Parallel data monitoring algorithms.

The data monitoring module is separated from the PCIe framework by asynchronous FIFOs, and hence can function at a clock frequency different from the one used by the PCIe framework. Let DR_f be the frequency of the data monitoring clock. Clock Factor, *CF*, is defined as the ratio of the frequency of the clock of the data monitoring module and the frequency of the PCIe clock. Thus, $CF = DR_f/P_f$.

The above definitions of l, p, u, z represent the load, unload, process and wait stages of the data monitoring module in terms of data monitoring module clock. Let L, P, U, Z represent the parameters l, p, u, z respectively in the PCIe clock domain.

$$X = \frac{x}{CF}$$

where X = L, P, U, Z for x = l, p, u, z respectively.

The period (C_{DR}) of the data monitoring module is defined as the number of clock cycles required to load and process B_{inDR} bytes by the data monitoring module in order to produce B_{outDR} bytes of data.

$$C_{DR} = L + U + P + Z$$

The input bandwidth (BW_{inDR}) of the data monitoring module is defined as the number of bytes loaded into the data monitoring module in one period. The output bandwidth (BW_{outDR}) of the data monitoring module is the number of bytes produced by the data monitoring module in one period:

$$BW_{inDR} = \frac{B_{inDR}}{C_{DR}}$$
$$BW_{outDR} = \frac{B_{outDR}}{C_{DR}}$$

The in-FIFO and out-FIFO connect to the data monitoring module through a 64-bit interface. This limits the maximum input and output bandwidth of the data monitoring module to 8 bytes per PCIe clock. Also, the BW_{in_inFIFO} can be less than BW_{inDR}, which will result in a lower BW_{inDR} limited by $BW_{in-inFIFO}$. Thus, the effective $BW_{inDR}, BW_{inDR-eff}$ is defined as

$$BW_{inDR-eff} = Minimum(BW_{inDR}, BW_{in-inFIFO})$$

If $BW_{in-inFIFO} < BW_{inDR}$, the number of cycles required to load data increases as the data monitoring module waits for the data in the in-FIFO. As a result, the period of the data monitoring module increases due to the extended load cycle. Thus, the effective period of the data monitoring module is

$$C_{DR-eff} = \frac{B_{inDR}}{BW_{inDR-eff}}$$

The effective output bandwidth of the data monitoring module is defined as

$$B_{outDR-eff} = \frac{B_{outDR}}{C_{DR-eff}}$$

5.3 Time Required for One DMA Transfer in the FPGA

The following intermediate variables are defined to model the three phases of the data transfer process described in Section 5.1:

- C_{wait}: the number of PCIe clock cycles elapsed between two MWr REQs,
- $N_{r_{wait}}$: the number of MRd REQs that can be sent from the FPGA to PC in C_{wait} PCIe cycles,
- N_{r1}, N_{r2}: the number of MRd REQs sent from the FPGA to the PC in phase 1 and phase 2 respectively,
- N_{w2}, N_{w3}: the number of MWr REQs sent from the FPGA to the PC in phase 2 and phase 3 respectively,

- $C_{ph1}, C_{ph2}, C_{ph3}$: the total number of PCIe clock cycles elapsed in phase 1, phase 2 and phase 3 respectively,
- C_{HW}: the total number of PCIe clock cycles required for the FPGA to complete one transfer,
- T_{HW}: the total time in seconds taken by the FPGA to complete a transfer.

As discussed in Section 5.1, the data transfer in the FPGA occurs in three phases. The time elapsed in each phase is calculated as follows:

(i) *Phase 1:* The number of clock cycles consumed in phase 1 will be the sum of the number of clocks elapsed between the start of the transfer and the start of the first MWr REQ. Let F be the number of clock cycles consumed by the data monitoring module in order to produce the number of bytes equal to the size of one MWr TLP (S_w words of data).

$$\begin{aligned} F &= \frac{\text{Number of bytes required for one MWr TLP}}{\text{Number of bytes produced by the data monitoring module in PCIe clock cycle}} \\ &= \frac{4 \cdot S_w}{BW_{outDR-eff}} \end{aligned}$$

C_{ph1} is the sum of the initial constant delay, C_{delay}, the number of PCIe clock cycles required to load and process the first set of inputs and F. Thus,

$$C_{ph1} = C_{delay} + L + P + F \tag{5.1}$$

$$N_{r1} = \frac{\text{Number of PCIe clock cycles elapsed in phase 1}}{\text{Number of PCIe clock cycles required per MRd REQ}} = \frac{C_{ph1}}{C_{mrd-req}}$$

(ii) *Phase 2:* C_{ph2} is the total number of PCIe clock cycles required to finish sending all the MRd REQs to the PC. The total number of MRd REQs left from phase 1 that are to be sent out in phase 2,

$$N_{r2} = \begin{cases} N_r - N_{r1} & \text{for } N_r > N_{r1} \\ 0 & \text{for } N_r \leq N_{r1} \end{cases} \tag{5.2}$$

F PCIe clock cycles are required to produce one outgoing TLP. The difference between F and $C_{mwr-req}$ will determine the amount of time for which the MWr REQ state is stalled due to insufficient data in the out-FIFO. Thus,

$$C_{wait} = F - C_{mwr-req}$$

$$N_{r_{wait}} = \frac{C_{wait}}{C_{mrd-req}}$$

$N_{r_{wait}}$ can also be defined as the number of MRd REQs sent for every MWr REQ. The total number of MRd REQs sent in phase 2 is same as the number of number MRd REQs sent from FPGA to the PC for N_{w2} MWr REQs. Thus,

$$N_{r2} = N_{r_{wait}} \cdot N_{w2} \tag{5.3}$$

From Equations 5.2 and 5.3,

$$N_{w2} = \frac{N_r - N_{r1}}{N_{r_{wait}}}$$

C_{ph2} is same as the number of PCIe clock cycles required by the data monitoring module to produce data required for N_{w2} MWr REQs. Hence,

$$C_{ph2} = F \cdot N_{w2} \tag{5.4}$$

(iii) *Phase 3:* C_{ph3} is the number of PCIe clock cycles required to complete all the remaining MWr REQs,

$$C_{ph3} = F \cdot N_{w3} + C_{mwr-req} \tag{5.5}$$

where $N_{w3} = N_w - N_{w2}$.

From Equations 5.1, 5.4 and 5.5,

$$C_{HW} = C_{ph1} + C_{ph2} + C_{ph3} \tag{5.6}$$

Thus,

$$\begin{aligned} T_{HW} &= C_{HW} \cdot P_T \\ &= \left[L + P + 4 \cdot S_w \cdot \left[\frac{N_w + 1}{BW_{outDR-eff}} \right] + \left(C_{delay} + C_{mwr-req} \right) \right] \cdot P_T \end{aligned} \tag{5.7}$$

From the equation, the total time per transfer in the FPGA is a function of the constant PCIe framework parameters, $C_{mrd-req}$ and $C_{mwr-req}$, the output transfer size, B_w and the data monitoring parameters, L, P, and $BW_{outDR-eff}$.

5.4 Throughput of the Data Monitoring System

The total time per transfer is defined as the time elapsed from the start of one transfer until the start of the next transfer. Figure 5.3 shows the data transfer cycle in the data monitoring system. The start of a transfer is marked by the initiate transfer signal sent by the PC to the FPGA, and the end of the transfer is marked by the transfer done flag of the control register in the FPGA. During the DMA transfer, the software process prepares for the next transfer after which the control register in the FPGA is polled continuously. When the transfer done status is asserted, the PC sends a initiate transfer signal to the FPGA to start a new transfer.

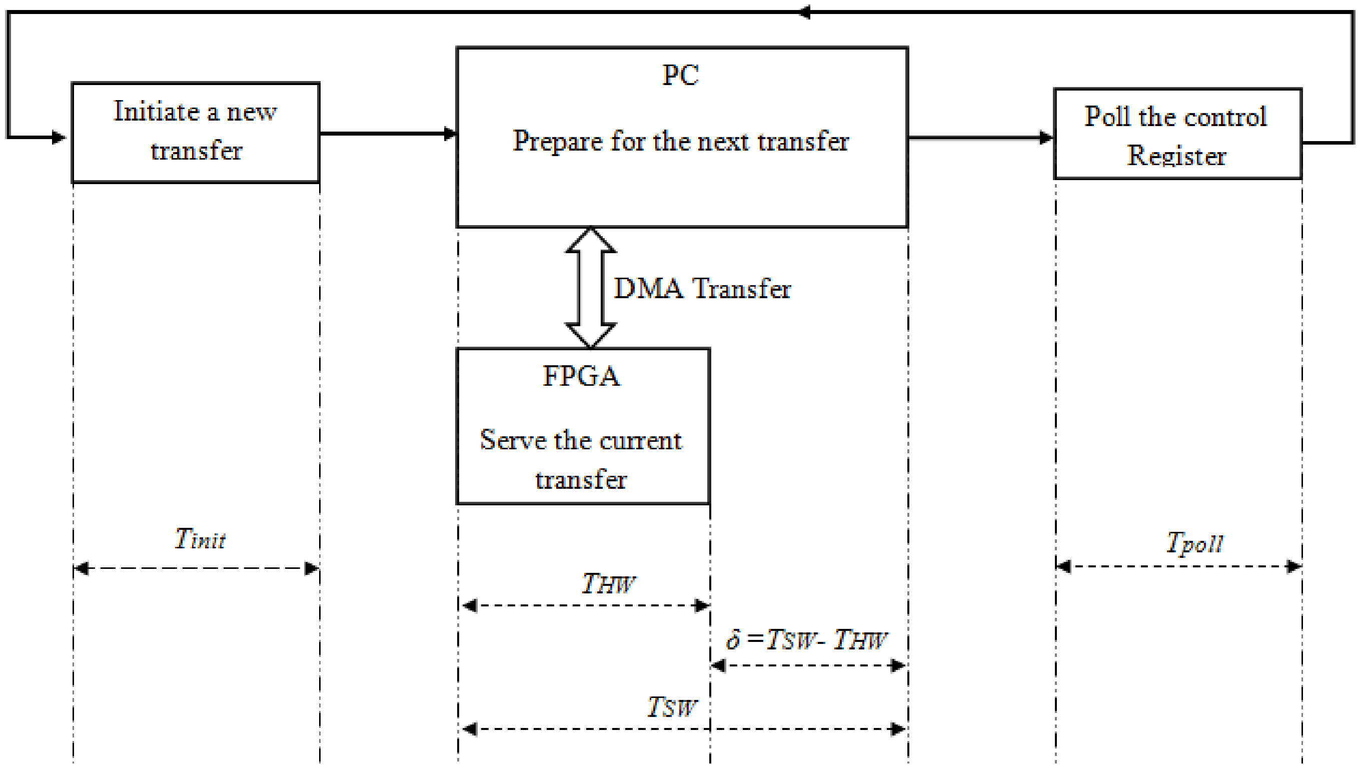

Initiate a new transfer
PC
Prepare for the next transfer
Poll the control Register
DMA Transfer
FPGA
Serve the current transfer
Tinit
THW
δ =TSW- THW
TSW
Tpoll

Figure 5.3: Data transfer cycle in the data monitoring system.

Each individual step of the data transfer process is described as follows:

(a) Initiate a PCIe transfer by:

(i) Resetting the hardware process for the new transfer by writing to the reset register in the FPGA.

(ii) Updating the hardware addresses of the read and write data buffers used for the DMA transfer. In the current design, the PCIe driver uses two read and two write data buffers for DMA transfers. This allows the software process to operate on one set of read/write buffers while the other set is being used for the ongoing DMA transfer. At the start of every transfer, the buffers are swapped and the corresponding hardware addresses are updated on the FPGA. Two PIO writes to the FPGA are made in order to update the hardware addresses of the read and write data buffers.

(iii) Clearing the transfer done flag of the control register by writing to the control register in the FPGA (one PIO write to FPGA), which will trigger the start of a new transfer in the FPGA.

Let T_{init} be the amount of time taken to initiate a transfer and T_{pio-wr} be the time required to complete one PIO write to the FPGA. T_{pio-wr} depends on the PCIe driver and the PC on which the driver is installed. For the hardware set up used in this thesis,

$$T_{pio-wr} = 160 \text{ ns}$$

$$T_{init} = 5 \cdot T_{pio-wr} = 800 \text{ ns}$$

(b) Prepare for the next transfer: While the DMA transfer is in progress, the software process prepares for the next transfer after which the control register is polled. From Section 4.3.1, the total time, T_{SW}, required by the PC to prepare for the next transfer will be the sum of the individual buffer updates as shown in Equation 5.8.

$$T_{SW} = T_{recv} + T_{write} + T_{read}, \tag{5.8}$$

where T_{recv} is the time required to receive data (sufficient for one DMA transfer between the PC and the FPGA of the data monitoring system) from the DP system through the 10GbE link, T_{write} is the time required to write data to the PCIe kernel buffer, and T_{read} is the time required to read data from the PCIe kernel buffer.

Let δ be the excess time consumed by the software process over the hardware process before the control register is polled.

$$\delta = \begin{cases} T_{SW} - T_{HW} & \text{for } T_{SW} > T_{HW} \\ 0 & \text{for } T_{SW} \leq T_{HW} \end{cases}$$

(c) Poll the control register: The contents of the control register of the FPGA are read via PIO transfers. When no DMA transfer is in progress, a PIO read consumes one clock cycle in the FPGA. While the DMA transfer is in progress, the PIO read is blocked if there is an ongoing MWr REQ or a MRd REQ after which the PIO read request is served. Every poll can thus delay the transfer by a clock cycle. As the software process consumes a significant amount of time, the number of times polled per transfer is less and hence the delay due to polling is ignored.

Let T_{poll} be the time taken for the last poll of the control register which corresponds to the time spent after the transfer is done. $T_{poll} = T_{pio-read}$, where $T_{pio-read}$ is the time taken by the software process to finish one PIO read from FPGA. For the current hardware set up, $T_{pio-read} = 2000$ ns. Hence,

$$T_{poll} = T_{pio-read} = 2000 \text{ ns}$$

The total time required to complete a single transfer, T_{total}, can be modelled as

$$\begin{aligned} T_{total} &= T_{init} + T_{HW} + \delta + T_{poll} \\ &= D + \delta + \\ &\quad \left[L + P + 4 \cdot S_w \cdot \left\lceil \frac{N_w + 1}{BW_{outDR-eff}} \right\rceil + \left(C_{delay} + C_{mwr-req} \right) \right] \cdot P_T \end{aligned} \tag{5.9}$$

where $D = T_{init} + T_{poll}$, which is constant for the hardware set-up. Thus, the total time taken for one transfer is a function of the output transfer size, B_w, the output bandwidth of the data monitoring module, $BW_{outDR-eff}$, and time required exclusively for the software process, T_{SW}.

The input throughput of the data monitoring system, $Thrpt_{in}$, is defined as the number of bytes transferred from the DP system to the data monitoring system per unit time:

$$Thrpt_{in} = \frac{B_r}{T_{total}}$$

The output throughput of the data monitoring system, $Thrpt_{out}$, is defined as the number of bytes transferred from the data monitoring system to the DRSUs in unit time:

$$Thrpt_{out} = \frac{B_w}{T_{total}} \tag{5.10}$$

The effective throughput of the data monitoring system, $Thrpt_{DR}$, is

$$Thrpt_{DR} = \text{Minimum}\ (Thrpt_{in}, Thrpt_{out}) \tag{5.11}$$

5.5 Maximum Achievable PCIe Input Bandwidth

$BW_{in-inFIFO}$ is the rate at which data arrives at the in-FIFO, i.e. the rate at which the MRd CPLDs are received by the RX engine. In order to achieve the maximum $BW_{in-inFIFO}$, the PC should always have pending MRd CPLDs to send back to the FPGA in such a way that there is no delay between MRd CPLDs because of lack of MRd REQs. Hence, the MRd REQs should be sent as quickly as possible so that the PC always has pending MRd CPLDs to send back to the FPGA. As mentioned previously, a 64-bit PCIe link has been used in this design. Hence, theoretically, the maximum $BW_{in-inFIFO}$ is 8 bytes per PCIe clock. The hardware used in this thesis uses a PCI Express 1.0 bus (PCIe physical link) which limits the maximum theoretical $BW_{in-inFIFO}$ to 4 bytes per PCIE clock (250 MB/s). Experimentally, it is observed that the maximum possible $BW_{in-inFIFO}$ is lower than the theoretical limit. This decrease in the maximum $BW_{in-inFIFO}$ is due to the latency associated with each transfer due to stalling of the transfer by the PC. Section 5.9.1 discusses more about the nature of the latencies and its effect on the input bandwidth of the data monitoring system. The experimental maximum of $BW_{in-inFIFO}$ is measured for transfer sizes varying from 4096 bytes to 4194304 bytes and is approximately:

$$BW_{in-inFIFO_{max}} = 2.63 \text{ bytes per PCIe clock} \tag{5.12}$$

Thus, the maximum input throughput of the PCIe link, $Thrpt_{in-max_{PCIe}}$,

$$\begin{aligned} Thrpt_{in-max_{PCIe}} &= \frac{\text{Number of bytes transferred to the FPGA}}{\text{time elapsed}} \\ &= \frac{BW_{in-inFIFO_{max}}}{P_T} \qquad (5.13) \\ &= 164.37 \text{ MB/s} \qquad (5.14) \end{aligned}$$

5.6 Maximum Achievable PCIe Output Bandwidth

The number of bytes generated by the data monitoring module in one period is always less than or equal to number of bytes loaded into the data monitoring module in one period. From the definitions of $BW_{in-outFIFO}$, $BW_{outDR-eff}$ and $BW_{inDR-eff}$,

$$BW_{in-outFIFO} \leq BW_{in-inFIFO}$$

$BW_{out-outFIFO}$ is the rate at which the TX engine (MWr REQ state) unloads data from the out-FIFO. The TX engine enters the MWr REQ state only if there is sufficient data in the out-FIFO. Thus, $BW_{out-outFIFO}$ also depends on $BW_{in-outFIFO}$:

$$BW_{out-outFIFO} \leq BW_{in-outFIFO}$$

and

$$BW_{out-outFIFO} \leq BW_{in-inFIFO}$$

with

$$BW_{out-outFIFO_{max}} = 2.63 \text{ bytes per PCIe clock} = 164.37 \text{ MB/s}$$

Therefore, the maximum output throughput of the PCIe link, $Thrpt_{out-max_{PCIe}}$, is

$$\begin{aligned} Thrpt_{out-max_{PCIe}} &= \frac{\text{Number of bytes transferred to the FPGA}}{\text{time elapsed}} \\ &= \frac{BW_{out-outFIFO_{max}}}{P_T} \\ &= 164.37 \text{ MB/s} \end{aligned}$$

5.7 Constant PCIe Input Bandwidth

The input bandwidth of the PCIe link is constant irrespective of the data monitoring module parameters and the PCIe transfer parameters. In order to derive the constant $BW_{in-inFIFO}$, $BW_{out-outFIFO}$ is first determined based on the following two conditions:

(i) The out-FIFO can be unloaded only when there is data in it. Thus, $BW_{out-outFIFO}$ is always less than or equal to the rate at which the out-FIFO is loaded $BW_{in-outFIFO}$.

$$BW_{out-outFIFO} \leq BW_{in-outFIFO}$$

(ii) The out-FIFO is unloaded by the TX engine and hence $BW_{out-outFIFO}$ also depends on the rate at which the TX engine can unload data ($BW_{out-outFIFO_{TX}}$). $BW_{out-outFIFO}$ is always less than or equal to the rate at which the TX engine can unload the data.

$$BW_{out-outFIFO} \leq BW_{out-outFIFO_{TX}}$$

Thus,

$$BW_{out-outFIFO} = \text{Minimum } (BW_{in-outFIFO}, BW_{out-outFIFO_{TX}}) \tag{5.15}$$

Assuming that the out-FIFO always has sufficient data, the maximum rate at which the TX engine can unload data,

$$\begin{aligned} BW_{out-outFIFO_{TX}} &= \frac{\text{Number of bytes unloaded from the out-FIFO for one MWr REQ}}{\text{Number of PCIe clock cycles for one MWr REQ}} \\ &= \frac{4 \cdot S_w}{C_{mwr-req}} \\ &= \frac{8}{1+\frac{4}{S_w}} \end{aligned} \tag{5.16}$$

If $BW_{out-outFIFO} = BW_{out-outFIFO_{TX}}$, the MWr REQs are continuously sent to the PC without any wait period between two MWr REQs. As the MWr REQs have a higher priority over the MRd REQs, the MRd REQs are blocked until all the MWr REQs are sent which in turn block the MRd CPLDs from the PC. This will result in a decrease in the $BW_{in-inFIFO}$ below the maximum value, $BW_{in-inFIFO_{max}}$.

From the definitions of $BW_{in-outFIFO}$ and $BW_{outDR-eff}$,

$$BW_{in-outFIFO} \leq BW_{in-inFIFO} \tag{5.17}$$

From Equation 5.16 and 5.12,

$$BW_{out-outFIFO_{TX}} > BW_{in-inFIFO_{max}} \quad \text{for all } S_w > 2 \tag{5.18}$$

From Equations 5.15, 5.17 and 5.18,

$$BW_{out-outFIFO} = BW_{in-outFIFO} \leq BW_{out-outFIFO_{TX}} \quad \text{for all } S_w > 2$$

Thus, for all $S_w > 2$, there always exists a wait period between two MWr REQs corresponding to loading of out-FIFO with sufficient data. Pending MRd REQs will be sent to the PC during the wait period. Thus, the MRd REQ queue at the PC never empties during the transfer since for a data monitoring system, the $BW_{outDR-eff}$ is always less than or equal to $BW_{inDR-eff}$. Hence, the in-FIFO always operates at the constant maximum bandwidth,

$$BW_{in-inFIFO} = BW_{in-inFIFO_{max}} \text{ for all } S_w > 2.$$

5.8 Buffer Overflows

FIFOs are fixed sized memories which are used to store data and retrieve it when needed. As the FIFOs get loaded with data continuously, there is a possibility that they become full and can no longer load new data elements. Any new data element provided to the FIFO when it is full will be ignored/discarded and the buffer is said to have overflowed beyond its limit. Buffer overflows can be avoided if the rate at which the FIFO is emptied is always greater than or equal to the rate at which the FIFO is loaded.

5.8.1 PCIe Buffers on the FPGA

The PCIe framework is connected to two FIFOs: in-FIFO to buffer the incoming data from the PC, and the out-FIFO to buffer the outgoing data obtained from the data monitoring module. As discussed in Section 5.7, the TX engine always unloads data from out-FIFO at a rate faster than the $BW_{in-outFIFO}$. Hence, the out-FIFO does not overflow for all $S_w > 2$.

In case of the in-FIFO, $BW_{in-inFIFO}$ is constant whereas the $BW_{out-inFIFO}$ depends on the data monitoring module. From the definition of $BW_{inDR-eff}$ and $BW_{out-inFIFO}$,

$$BW_{out-inFIFO} = BW_{inDR-eff}.$$

If $BW_{inDR-eff} < BW_{in-inFIFO}$, the in-FIFO fills up faster than the rate at which it is unloaded. The data monitoring system operates on data streams and hence the in-FIFO will overflow for transfers exceeding a specific limit. Let B_{add} be the number of bytes retained in the in-FIFO for one PCIe clock:

$$B_{add} = \begin{cases} BW_{inDR-eff} - BW_{in-inFIFO} & \text{for } BW_{inDR-eff} < BW_{in-inFIFO} \\ 0 & \text{for } BW_{inDR-eff} \geq BW_{in-inFIFO} \end{cases} \tag{5.19}$$

Let $IN\text{-}FIFO\text{-}SIZE$ represent the total number of bytes the in-FIFO can store. C_{OF} is the number of PCIe clock cycles required to fill the in-FIFO with $IN\text{-}FIFO\text{-}SIZE$ number of bytes beyond which the in-FIFO will overflow.

$$C_{OF} = \frac{IN\text{-}FIFO\text{-}SIZE}{B_{add}} \tag{5.20}$$

Let B_{C-OF} be the number of bytes that can be sent to the FPGA (in-FIFO) from the PC in C_{OF} clock cycles i.e. the maximum number of bytes that can be sent to the FPGA without causing an overflow.

$$B_{C-OF} = C_{OF} \cdot BW_{in-inFIFO}. \tag{5.21}$$

From 5.19, 5.20 and 5.21, the total number of bytes that can be sent to the FPGA per transfer without causing an overflow of the in-FIFO, $B_{r-no-OF}$, should be less than or equal to B_{C-OF}

$$B_{r-no-OF} \leq \frac{IN\text{-}FIFO\text{-}SIZE}{1 - R}, \quad \text{where } R = \frac{BW_{inDR-eff}}{BW_{in-inFIFO}} \tag{5.22}$$

Thus buffer overflow depends on the size of the in-FIFO and the output bandwidth of the data monitoring module. The lower the output rate of the data monitoring system, the higher will be the required in-FIFO size in order to avoid buffer overflows. At any point, the number of words available in the FIFO is obtained making a PIO read request to the FPGA. Buffer overflow is detected by reading the buffer overflow flag of the control register during polling.

5.8.2 10GbE Buffer

The current system is developed as a plug-in to the already existing data streaming system used in the LWA station. The data streams received via the 10GbE link are stored in a 10GbE kernel buffer. If the data monitoring system does not unload the 10GbE buffer at a sufficient rate, the 10GbE overflows and a few packets from the DP system will be lost. In order to avoid packet drops, the time taken to finish processing the set of inputs received from the DP system should be greater than or equal to the time taken to receive the inputs.

$$\begin{aligned} T_{total} - T_{recv} &\leq T_{recv}, \\ T_{total} &\leq 2 \cdot T_{recv} \end{aligned}$$

In terms of throughput,

$$Thrpt_{DR} \geq \frac{Thrpt_{10GbE}}{2}$$

where $Thrpt_{10GbE}$ is the throughput of the 10GbE receiver of the data monitoring system. The DP system outputs data to the data monitoring system at a maximum rate of 112 MB/s with a packet size of 4096 bytes. In order to avoid packet misses,

$$Thrpt_{DR} \geq \frac{120}{2}\ \text{MB/s} \geq 60\ \text{MB/s}$$

5.9 Measurements and Analysis

This section presents the measured throughput of the data monitoring system as a function of the PCIe transfer parameters and the data monitoring module parameters. The measured values are compared with the throughput calculated from the modelled equation derived in Section 5.4. The maximum transfer size up to which no PCIe buffer overflows occur is calculated from Equation 5.22 and compared with the experimental values.

5.9.1 Throughput of the Data Monitoring System

As discussed in Section 5.7, $Thrpt_{in-max_{PCIe}}$ is constant and does not vary with the data monitoring or the PCIe parameters. The $Thrpt_{in-max_{PCIe}}$ is a function of the data monitoring parameters and can reach a maximum of 165.375 MB/s. From Section 5.5 and 5.6,

$$\begin{aligned} Thrpt_{DR} &= \text{Minimum } (Thrpt_{in}, Thrpt_{out}) \\ &= Thrpt_{out} \end{aligned} \tag{5.23}$$

From Equations 5.23 and 5.10, the overall throughput of the data monitoring system, $Thrpt_{DR}$, is a function of B_w, $BW_{outDR-eff}$ and δ. This section compares the results obtained from the modelled equation with the actual measured values for the throughput of the data monitoring system. In order to model the effect of the bridge platform parameters, all the measurements in this section were made with $\delta = 0$ i.e. the software process completes preparration for the next transfer before the hardware process completes the corresponding DMA transfer.

(a) Input throughput: As mentioned in Section 5.5, the data monitoring system delivers a maximum throughput of 164.37 MB/s. The decrease in the maximum input throughput below the theoretical limit (250 MB/s) is due to the latencies associated with the transfer of data from the PC to the FPGA. The input bandwidth is a function of the latencies due to the PCIe DMA transfer. PCIe read latency (Q) is measured as the difference between the theoretical and the measured number of PCIe clock cycles required to transfer B_r bytes from the PC to the FPGA. Figure 5.4 shows that the measured latencies increase linearly with the increase in the input transfer size (B_r) due to the PCIe delivering a constant maximum $BW_{in-inFIFO}$ of 164.37 MB/s irrespective of the transfer size.

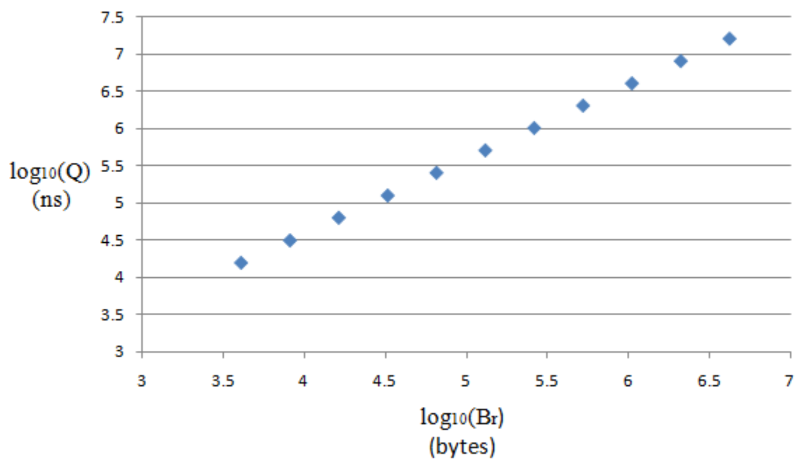


Figure 5.4: PCIe read latency versus input transfer size.

(b) Effect of B_w on $Thrpt_{DR}$: In order to measure the effect of B_w on $Thrpt_{DR}$, the data monitoring system was tested with a loopback mode. In the loopback mode, the output of the in-FIFO is directly routed to the input of the out-FIFO. The loopback mode will thus give an estimate of the maximum possible throughput for a particular transfer size, B_w. Figure 5.5 shows the plot of $Thrpt_{DR}$ versus B_w. The effect of overheads associated with initiating a transfer are higher for lower values of B_w. As B_w increases, the total time required per transfer increases due to the constant overhead decreasing. Thus, the throughput increases with increase in the transfer size, and reaches the maximum value of 164 MB/s as expected.

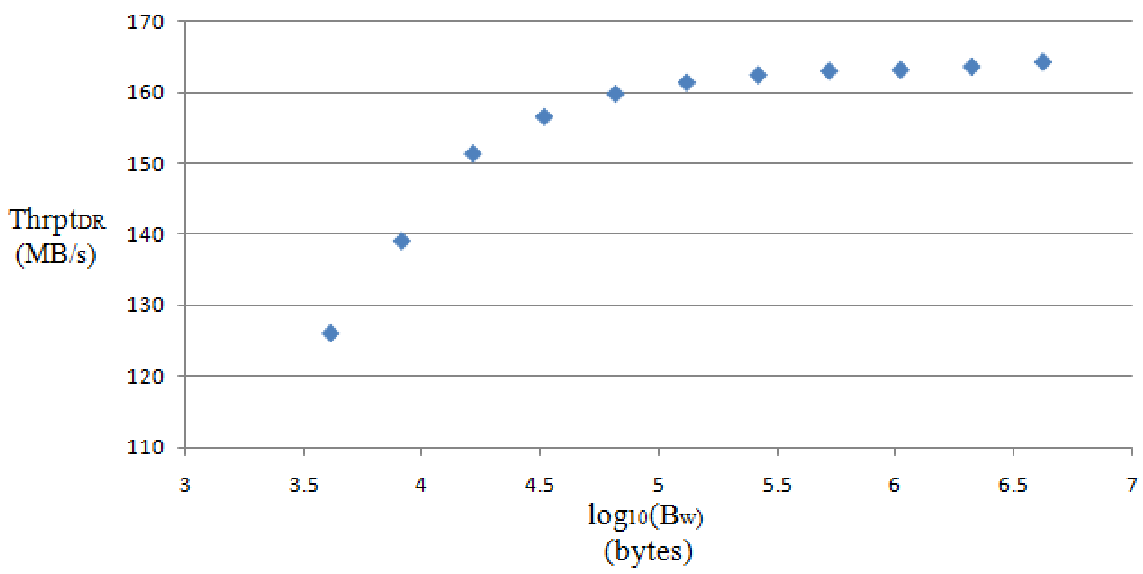
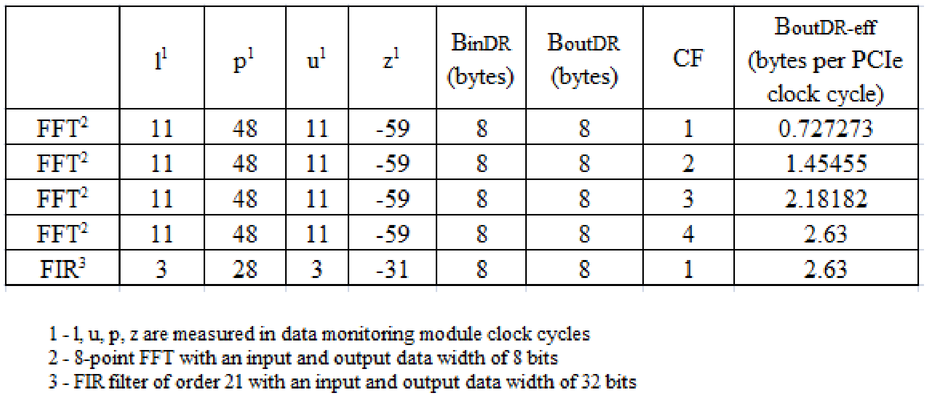


Figure 5.5: Throughput of the data monitoring system versus output transfer size.

(c) Effect of data monitoring module parameters on $Thrpt_{DR}$: As discussed in Section 5.4, $Thrpt_{DR}$ varies with $BW_{outDR-eff}$. An 8-point FFT block with a data width of 8 bits and an FIR filter of order 21 and data width 32 bits have been used as a sample data monitoring building blocks to quantify the effect of the data monitoring parameters on $Thrpt_{DR}$. Table 5.1 lists the parameters of FFT and FIR processing blocks.



	l[1]	p[1]	u[1]	z[1]	BinDR (bytes)	BoutDR (bytes)	CF	BoutDR-eff (bytes per PCIe clock cycle)
FFT[2]	11	48	11	-59	8	8	1	0.727273
FFT[2]	11	48	11	-59	8	8	2	1.45455
FFT[2]	11	48	11	-59	8	8	3	2.18182
FFT[2]	11	48	11	-59	8	8	4	2.63
FIR[3]	3	28	3	-31	8	8	1	2.63

1 - l, u, p, z are measured in data monitoring module clock cycles
2 - 8-point FFT with an input and output data width of 8 bits
3 - FIR filter of order 21 with an input and output data width of 32 bits

Table 5.1: Parameters of FFT and FIR processing blocks.

Figure 5.6 shows the effect of $BW_{outDR-eff}$ on the throughput of the system. As $BW_{outDR-eff}$ increases, $Thrpt_{DR}$ also increases and reaches the maximum for $BW_{outDR-eff}$ $= BW_{in-in-FIFO}$. From Figure 5.6, for $B_w = 4096$ bytes and $BW_{outDR-eff} = BW_{in-in-FIFO}$, $Thrpt_{DR}$ = 123 MB/s, which is approximately equal to the $Thrpt_{DR}$ for B_w = 4096 bytes, is 125 MB/s from Figure 5.5.

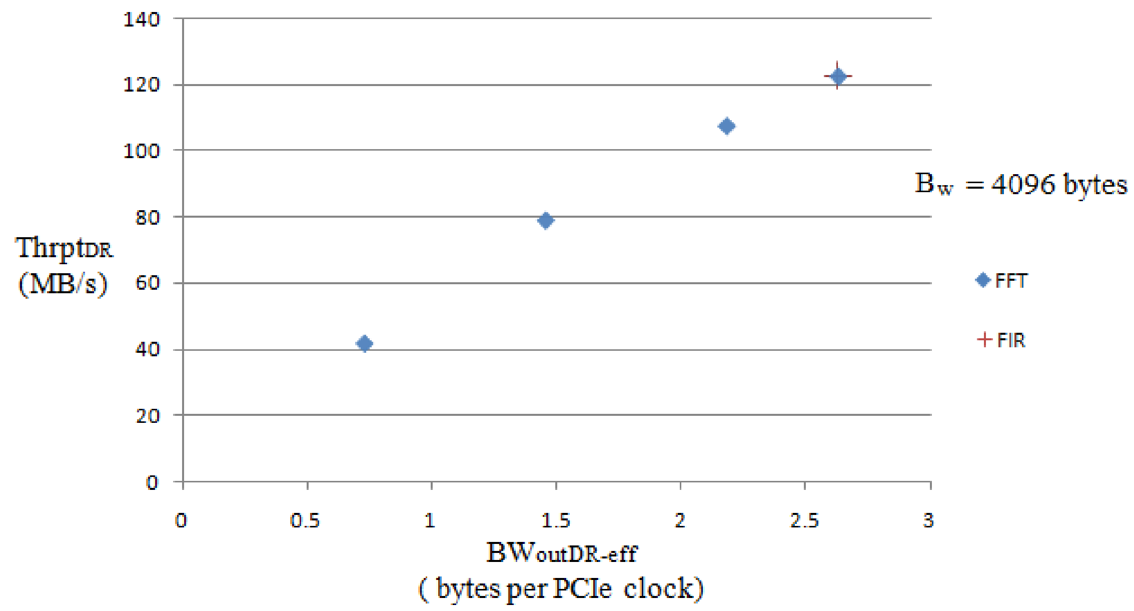


Figure 5.6: Throughput of the data monitoring system versus output bandwidth of the data monitoring module.

Figure 5.7 compares the $Thrpt_{DR}$ delivered by the FFT and FIR blocks with the same $BW_{outDR-eff}$. Thus, the measurements also show that $Thrpt_{DR}$ is controlled by the $BW_{outDR-eff}$ parameter of the data monitoring module irrespective of the nature of the algorithm and the other parameters mentioned in Section 5.2.4.

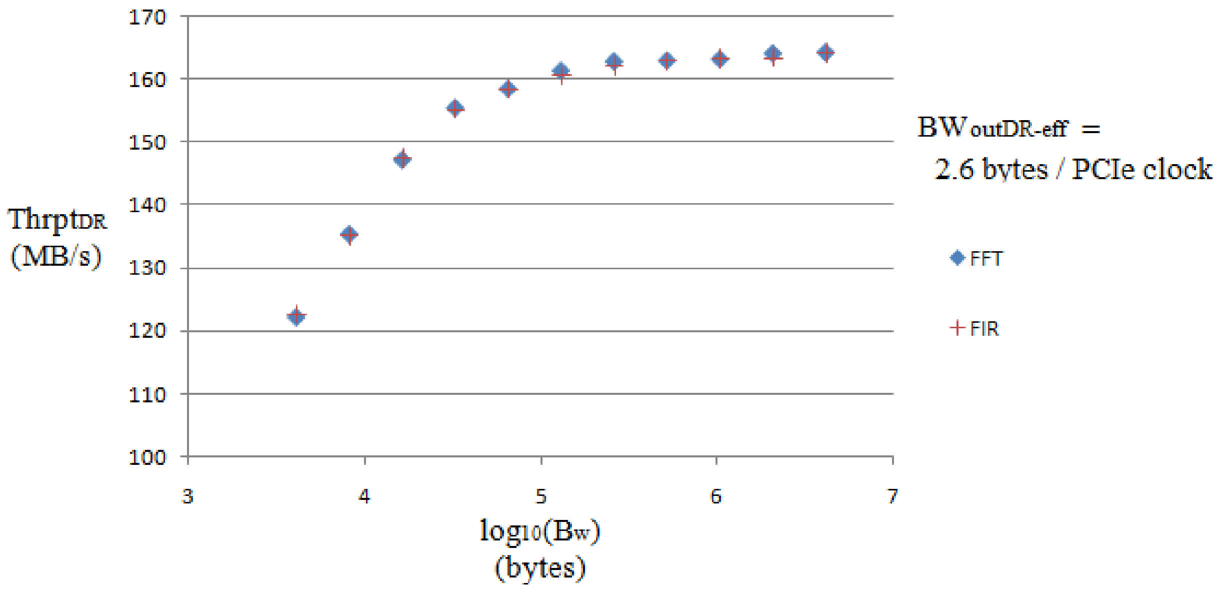


Figure 5.7: Throughput delivered by FFT and FIR blocks with same $BW_{outDR-eff}$.

(d) Combined effect of B_w and $BW_{outDR-eff}$ on $Thrpt_{DR}$: Figure 5.8 shows a plot of the combined effect of B_w and $BW_{outDR-eff}$ on $Thrpt_{DR}$. From Equation 5.10, $Thrpt_{DR}$ decreases as B_w increases and as $BW_{outDR-eff}$ increases which is also demonstrated by the measured values. Figure 5.8 also plots the modelled $Thrpt_{DR}$ obtained from Equation 5.10 for varying B_w and $BW_{outDR-eff}$ values. The modelled value closely predicts the $Thrpt_{DR}$.

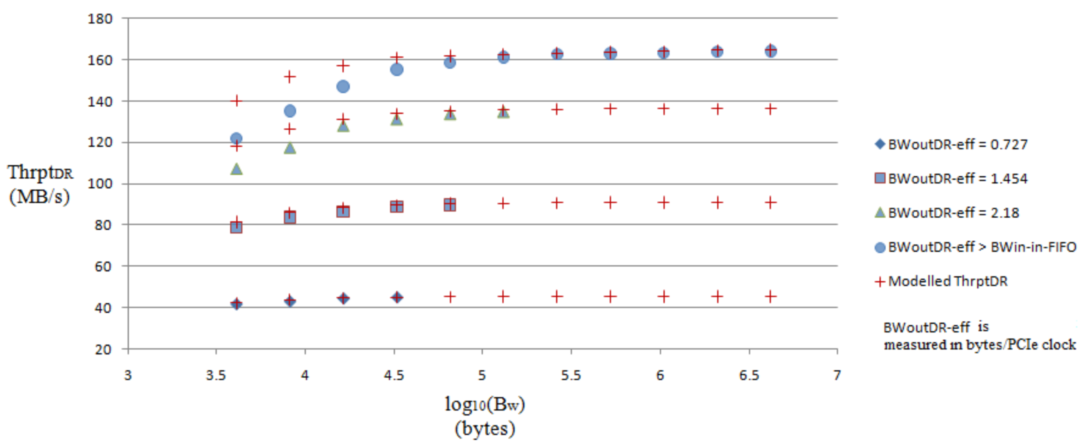
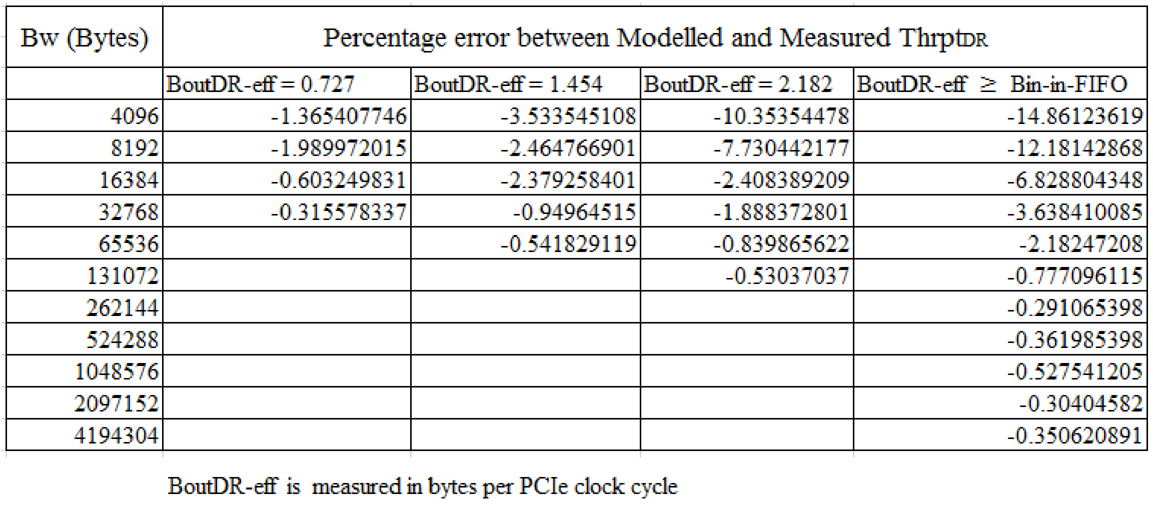


Figure 5.8: Effect of B_w and $BW_{outDR-eff}$ on the throughput of the data monitoring system.



Bw (Bytes)	Percentage error between Modelled and Measured ThrptDR			
	BoutDR-eff = 0.727	BoutDR-eff = 1.454	BoutDR-eff = 2.182	BoutDR-eff ≥ Bin-in-FIFO
4096	-1.365407746	-3.533545108	-10.35354478	-14.86123619
8192	-1.989972015	-2.464766901	-7.730442177	-12.18142868
16384	-0.603249831	-2.379258401	-2.408389209	-6.828804348
32768	-0.315578337	-0.94964515	-1.888372801	-3.638410085
65536		-0.541829119	-0.839865622	-2.18247208
131072			-0.53037037	-0.777096115
262144				-0.291065398
524288				-0.361985398
1048576				-0.527541205
2097152				-0.30404582
4194304				-0.350620891

BoutDR-eff is measured in bytes per PCIe clock cycle

Table 5.2: Percentage error between the modelled and measured $ThrptDR$.

From Table 5.2, it is observed the percentage error between the modelled and the measured value averages to 3%. The effect of overheads is higher for lower values of B_w. Due to the non-deterministic nature of the multi-core PC, the software overheads show a high variance leading to higher errors for lower values of B_w which yield higher $Thrpt_{DR}$.

5.10 Effect of δ on Throughput

δ is the excess time consumed by the software process after the DMA transfer is completed by the FPGA. For varying values of B_w, δ was calculated and the modelled $Thrpt_{DR}$ is obtained as shown in Figure 5.9. From Equation 5.10, as δ increases, the $Thrpt_{DR}$ decreases as observed in Figure 5.9.

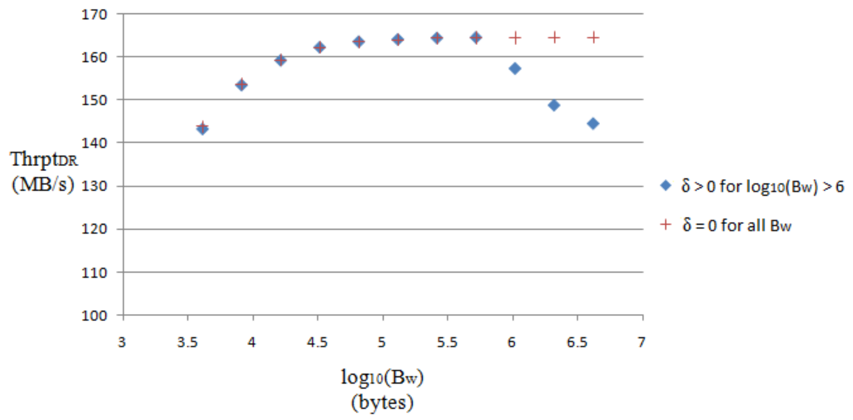


Figure 5.9: Effect of δ on the throughput of the data monitoring system.

T_{write} and T_{read} are fairly constant for a specific transfer size. T_{recv} depends on $Thrpt_{10GbE}$ which had a high variance. Moreover, if $T_{SW} <<< T_{HW}$, the software process should wait for a specific period before polling the control register which otherwise would cause a decrease in $Thrpt_{DR}$ because of polling delays. Due to the non-deterministic nature of the multi-core PC used in this thesis, the wait process in software could not be precisely modelled. Modelling software delays is an area to be explored further.

5.11 Maximum Transfer Size for No Buffer Overflows in the FPGA

The maximum transfer size in order to avoid PCIe buffer overflow has been calculated for a FIFO size of 32760 bytes depth 8190 and width 32-bits) and plotted as a function of $BW_{outDR-eff}$ as shown in Figure 5.10. The experimental values obtained by using the FFT block for varying $BW_{outDR-eff}$ are tabulated along with the modelled values in Table 5.3 which predict the B_{C-OF} with high accuracy.

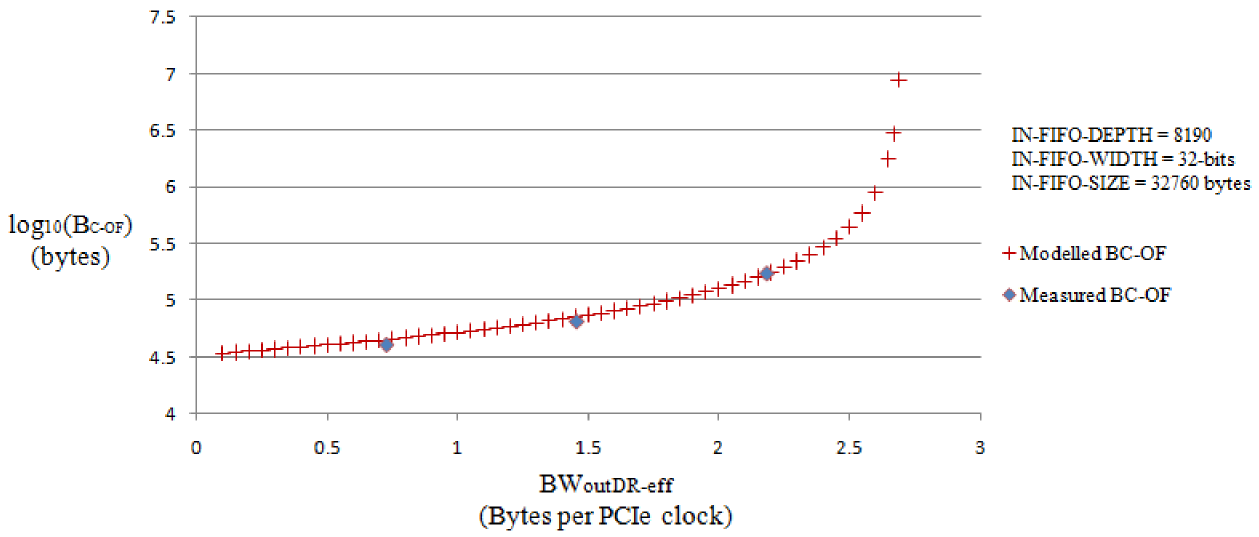
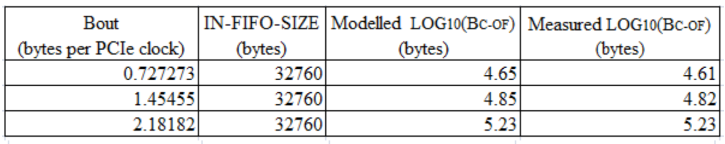


Figure 5.10: Maximum transfer size for no PCIe buffer overflows.



Bout (bytes per PCIe clock)	IN-FIFO-SIZE (bytes)	Modelled LOG10(BC-OF) (bytes)	Measured LOG10(BC-OF) (bytes)
0.727273	32760	4.65	4.61
1.45455	32760	4.85	4.82
2.18182	32760	5.23	5.23

Table 5.3: Modelled and measured B_{C-OF}.

5.12 Summary

This chapter lists the parameters of the data monitoring system. Section 5.2.4 defines the parameters of the data monitoring plug-in irrespective of the nature of the algorithm. The total time taken per transfer is derived in terms of the system parameters in Sections 5.3 and 5.4. The effect of the system parameters on the total throughput of the data monitoring system is discussed in Section 5.9. The conditions for avoiding buffer overflows is derived in Section 5.8.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The developed system is a generic data-centric system that can be used to monitor data streams in real-time. The data-centric system is developed to evaluate FPGAs for implementing data monitoring algorithms in LWA. The FPGA-based data monitoring system allows assesment of data captured by the LWA and provides the flexibility of implementing various monitoring algorithms. In this thesis, a XUPV5 board with a Virtex-5 FPGA has been selected as it was readily available and had the features necessary for the data monitoring system.

The PC component of the data monitoring system sends data to and from the XUPV5. Screened data can be routed to output devices such as storage units. In the modified design of the MCS-DR system, the DP system sends data to the MCS-DR PC which routes it to the XUPV5 board. Data from the XUPV5 board may be routed to the DRSUs. In order to not discard packets from the DP system, the entire data monitoring system should function at a rate greater than or equal to the output data rate of the DP system. The DP system can send data to the MCS-DR PC at a maximum rate of 112 MiB/s (TBN/TBW systems). In order to avoid losing data from the DP system, the data monitoring system should function at a rate of at least 112 MiB/s. This thesis aims at modelling the throughput of the data monitoring system and identify the parameters that affect its throughput.

The throughput of the data monitoring system depends on the rate at which the FPGA can process and transfer data back to the DRSUs. This thesis models the throughput of the data monitoring system as a function of the parameters specific to the PCIe communication and the data monitoring plug-in. From the modelled equations, the following observations are made:

(i) As the amount of data sent out of the FPGA per transfer increases, the maximum throughput of the data monitoring system increases rapidly up to the transfer sizes 32768 bytes and then averages to 160 MB/s beyond the transfer size of 65536 bytes.

(ii) Irrespective of the nature of the algorithm, any data monitoring module can be modelled using the number of clock cycles required to load one set of inputs, process them, unload the corresponding outputs and wait until the next set of inputs. The throughput of the data reduction module can be obtained from the number of bytes per load and unload cycle.

(iii) The overall throughput of the PCIe link is limited only if the throughput of the data monitoring module is less than the throughput that can be delivered by the PCIe link for the specified data transfer size.

(iv) The throughput of the data monitoring system decreases as the time consumed by the software processes increase beyond the time taken by the hardware process for one DMA transfer.

(v) The maximum possible throughput for the hardware used in this thesis is $\sim$160 MB/s obtained for transfer sizes greater than 65536 bytes in cases where the throughput of the data monitoring module alone is greater than 160 MB/s and the time consumed by software process is less than the time required to complete the DMA transfer.

It is observed that the latencies involved with the transfer of data to the FPGA vary linearly with the increase in the transfer size, thus delivering a constant maximum throughput of the PCIe link irrespective of the transfer size. But the overall throughput of the data monitoring system is low for smaller transfer sizes due to the software overheads associated with each transfer. DSP building blocks such as an 8-point FFT and an FIR filter of order 21 were developed in XSG and plugged into the design. The experimental values for throughput and the condition for no buffer overflows closely match the predicted values. The throughput of the data monitoring module and the excess time consumed by the software process are the major factors that affect overall throughput.

In LWA, the MCS-DR PC acts as an interface between the data monitoring unit (FPGA), and the DP and DRSU systems. The FPGA implements the PCIe protocol and the data monitoring algorithm. This thesis provides a framework on which algorithms can be plugged-in and does not develop realistic algorithms. Real data monitoring algorithms can be developed independently by the users and plugged-in to the framework developed in this thesis.

6.2 Future Work

Some of the areas for future work are:

- *Compatibility of the data monitoring system with the DRSUs:* In LWA, the MCS-DR PC receives inputs from the DP system via a 10GbE link and transmits outputs to the DRSUs via an eSATA link. The PC is configured to accept data via a 10GbE port which is sent to the XUPV5 board. Data from the XUPV5 board to the PC is discarded. Configuring the PC of the data monitoring system for an eSATA output link to the DRSUs will complete the data flow chain in the MCS-DR.

- *Using advanced development boards:* A Xilinx Virtex-5 FPGA has been used in the prototype developed in this thesis. The XUPV5 board used in this thesis uses a PCIe

1.0 bus which can theoretically function at 250 MB/s. Boards with newer version PCIe buses 2.0 can deliver higher performance. In this thesis, a single lane PCIe link has been used. Boards supporting multiple lanes yield higher throughput. Hence using advanced development boards with multiple lanes and larger capacity FPGAs with more logic, DSP and BRAM resources is a solution for applications limited by the throughput of the PCIe link.

- *Unblocked input and output PCIe transfers in FPGA:* In the current design, the data to the FPGA is continuous irrespective of the processing speed of the data monitoring module. However,the outgoing data from the FPGA depends on the throughput of the data monitoring module. The PCIe framework is blocked by the data monitoring module until it generates sufficient outputs and thus the throughput of the PCIe link is regulated by the data monitoring module. Modifying the PCIe framework on the FPGA for unblocked read and write states and determining the effect of unblocked states on throughput is an interesting area for future work.

- *Modelling software delays:* In the current design, the throughput was measured by varying the hardware parameters (on the FPGA). Throughput can also be modelled with respect to the software process. Due to the non-deterministic nature of the PC process scheduling the accuracy of the results decreases. Modelling the throughput in terms of software parameters and software latencies is another area of future work.

Appendix A

Bridge Platform Logic and Data Transfer Loop Software

A.1 FIFO Interface Logic

```
//
    ______________________________________________________________________________

//-- Filename:        manage_fifo_in_out.v
//-- Author:       Sushrutha Vigraham, Virginia Tech
//-- Last Updated: Jan 25th 2011
//--
//-- Description: Instantiates four FIFOs, two child FIFOs for in-FIFO
    and two child FIFOs for the out-FIFO receives enable signals
//--         Instatiates the bridge_fifo_plugin module in which the data
    monitroing plug-in module is inserted
//--         This module gets data from the RX engine and places it in in-
    FIFO
//--         This module gets monitored data from the data monitoring plug-
    in and placed it in out-FIFO
//--         This module asserts write enable and read enable to in-FIFo
    and out-FIFO based on the enable signals received from the RX and TX
     engines respectively
//--         Alternates between the two child FIFOs of the in and out child
     FIFOs
```

```
//
    ————————————————————————————————————————————————————————————————————————————

module manage_fifo_in_out (

        clk ,
        pcie_rst ,
        out_FIFO_rd_en ,
        read_update ,
        k ,
        fifo_in_data ,
        fifo_out0 ,
        fifo_out1 ,
        full ,
        empty ,
        total_datacount_in_infifos ,
        total_datacount_in_outfifos ,
        trn_rd
      );

//————————Input Ports————————
input clk;
input pcie_rst;
input [1:0] out_FIFO_rd_en;
input [1:0] read_update;
input k;       // Current FIFO that has to be read/written
input [63:0] fifo_in_data;
input [63:0] trn_rd;


//——————Output Ports——————
output [31:0] fifo_out0;
output [31:0] fifo_out1;
output [1:0] empty;
output [1:0] full;
output [20:0] total_datacount_in_infifos;
output [20:0] total_datacount_in_outfifos;

wire [63:0]  trn_rd1;
wire [1:0] wr_ack;
wire [1:0] valid;
wire [1:0] wr_en;
wire [31:0] data_in[1:0];
reg fifo_in_select;
wire plugin_clk;

//------------FIFO ports-------------

wire [31:0] transfer_data[1:0];
wire [20:0] fifoin_data_count [1:0];
wire [20:0] fifoout_data_count [1:0];
wire [1:0] fifoin_empty;
wire [1:0] fifoin_full;
wire [1:0] fifoout_empty;
wire [1:0] fifoout_full;

wire [63:0] din64;
wire [63:0] dout64;
wire [1:0] fifoin_rd_en;
wire [1:0] fifoout_wr_en;
wire [20:0] fifoin_other_count [1:0];
wire [20:0] fifoout_other_count [1:0];
wire [20:0] infifo_other_count;
wire [20:0] outfifo_other_count;
wire [20:0] in_fifo_count;

initial begin
  fifo_in_select = 1'b0;
end

//----------------FIFO Interface Logic----------------

assign full = fifoin_full;
assign empty = fifoout_full;

assign trn_rd1 = {trn_rd[07:00],trn_rd[15:08], trn_rd[23:16], trn_rd
    [31:24], trn_rd [39:32],trn_rd[47:40], trn_rd[55:48], trn_rd
    [63:56]}; // Data from the PC (MCS-DR PC)
assign din64 = {transfer_data[1], transfer_data[0]};     // Data from the
     data monitoring module

assign total_datacount_in_infifos = (fifoin_data_count[0] +
    fifoin_data_count[1]);
```

```
assign total_datacount_in_outfifos = (fifoout_data_count[0] +
    fifoout_data_count[1]);

assign wr_en[0] = ( (read_update == 2'b11) ? 1 : (((fifo_in_select == 0)
     && (read_update != 2'b00)) ? 1 : 0) );
assign wr_en[1] = ( (read_update == 2'b11) ? 1 : (((fifo_in_select == 1)
     && (read_update != 2'b00)) ? 1 : 0) );

assign data_in[0] = ( (read_update == 2'b11) ? ((fifo_in_select == 1'b0)
     ? (fifo_in_data[31:0]) : ((fifo_in_select == 1'b1) ? (fifo_in_data
    [63:32]) : 32'h0000)) :
            ( (fifo_in_select == 1'b0) ? ((read_update == 2'b01) ? (
                fifo_in_data[31:0]) : ((read_update == 2'b10) ? (
                fifo_in_data[63:32]) : 32'h0000)) : 32'h0000) );

assign data_in[1] = ( (read_update == 2'b11) ? ((fifo_in_select == 1'b1)
     ? (fifo_in_data[31:0]) : ((fifo_in_select == 1'b0) ? (fifo_in_data
    [63:32]) : 32'h0000)) :
            ( (fifo_in_select == 1'b1) ? ((read_update == 2'b01) ? (
                fifo_in_data[31:0]) : ((read_update == 2'b10) ? (
                fifo_in_data[63:32]) : 32'h0000)) : 32'h0000) );

always @ (posedge clk)
begin
  if(pcie_rst)
  begin
    fifo_in_select <= 0;
  end
  else
  begin
    if( (read_update == 2'b01) || (read_update == 2'b10) )
      fifo_in_select <= fifo_in_select + 1;
    else
      fifo_in_select <= fifo_in_select;
  end
end

//---------- FIFO Instances ---------------

//-----in-FIFO-child1---------------------
fifo_generator_v5_3_std in_fifo_0 (
  .rst(pcie_rst),
  .wr_clk(clk),
  .rd_clk(plugin_clk),
  .din(data_in[0]), // Bus [31 : 0]
  .wr_en(wr_en[0]),
  .rd_en(fifoin_rd_en[0]),
  .dout(transfer_data[0]), // Bus [31 : 0]
  .full(fifoin_full[0]),
  .empty(fifoin_empty[0]),
  .rd_data_count(fifoin_other_count[0]),
  .wr_data_count(fifoin_data_count[0])
  );

//-----in-FIFO-child2------------------
fifo_generator_v5_3_std in_fifo_1(
  .rst(pcie_rst),
  .wr_clk(clk),
  .rd_clk(plugin_clk),
  .din(data_in[1]),
  .wr_en(wr_en[1]),
  .rd_en(fifoin_rd_en[1]),
  .dout(transfer_data[1]),
  .full(fifoin_full[1]),
  .empty(fifoin_empty[1]),
  .rd_data_count(fifoin_other_count[1]),
  .wr_data_count(fifoin_data_count[1])
  );

//-----out-FIFO-child1------------------
fifo_generator_v5_3_fwft out_fifo_0 (
  .rst(pcie_rst),
  .wr_clk(plugin_clk),
  .rd_clk(clk),
  .din(dout64[31:0]),
  .wr_en(fifoout_wr_en[0]),
  .rd_en(out_FIFO_rd_en[0]),
  .dout(fifo_out0),
  .full(fifoout_full[0]),
  .empty(fifoout_empty[0]),
  .rd_data_count(fifoout_data_count[0]),
  .wr_data_count(fifoout_other_count[0])
  );

//-----out-FIFO-child2------------------
fifo_generator_v5_3_fwft out_fifo_1 (
  .rst(pcie_rst),
  .wr_clk(plugin_clk),
  .rd_clk(clk),
  .din(dout64[63:32]),
  .wr_en(fifoout_wr_en[1]),
  .rd_en(out_FIFO_rd_en[1]),
  .dout(fifo_out1),
  .full(fifoout_full[1]),
  .empty(fifoout_empty[1]),
  .rd_data_count(fifoout_data_count[1]),
  .wr_data_count(fifoout_other_count[1])
  );

//------------------------------------------------------------End FIFO
    Instances----------------------------------------------------

//-----Bridge FIFO (Data Monitoring Wrapper Code) Instance----

bridge_fifo_plugin bridge(
    .pcie_clk(clk),
    .plugin_clk(plugin_clk),
    .plugin_rst(pcie_rst), //plugin_rst
    .din64(din64),
    .fifoin_rd_en(fifoin_rd_en),
    .dout64(dout64),
    .fifoout_wr_en(fifoout_wr_en),
    .fifoin_empty(fifoin_empty),
    .fifoin_full(fifoin_full),
    .fifoout_full(fifoout_full)
  );
//--------------------------------------------------


endmodule


//---------FIFO modules (netlists are included in ipcore_dir/bmd_design/
    dma_performance_demo/fpga/BMD) ------------------
//---------FIFOs are different size can be used by generating ngc files
    and changing the following module declarations

//---------child-in-FIFO---------------
module fifo_generator_v5_3_std(
  rst,
  wr_clk,
  rd_clk,
  din,
  wr_en,
  rd_en,
  dout,
  full,
  empty,
  rd_data_count,
  wr_data_count);


input rst;
input wr_clk;
input rd_clk;
input [31 : 0] din;
input wr_en;
input rd_en;
output [31 : 0] dout;
output full;
output empty;
output [11 : 0] rd_data_count;
output [11 : 0] wr_data_count;

endmodule


//——— child-out-FIFO————————
module fifo_generator_v5_3_fwft(
  rst,
  wr_clk,
  rd_clk,
  din,
  wr_en,
  rd_en,
  dout,
  full,
  empty,
  rd_data_count,
  wr_data_count);
```

```
input rst;
input wr_clk;
input rd_clk;
input [31 : 0] din;
input wr_en;
input rd_en;
output [31 : 0] dout;
output full;
output empty;
output [12 : 0] rd_data_count;
output [12 : 0] wr_data_count;

endmodule
```

A.2 Data Monitoring Wrapper Logic

```
//
    -----------------------------------------------------------------------------

//-- Filename:      bridge_fifo_plugin.v
//-- Author:        Sushrutha Vigraham, Virginia Tech
//-- Last Updated: Jan 25th 2011
//--
//-- Description: Instatiates the data monitoring wrapper
//--        Receives signals from the data monitoring plugin and asserts
    read enable and write enable to in-FIFO and out-FIFO respectively
//--        Instantiates the DCM primitive to generate the clock signal
    for the data monitoring module
//--
//
    -----------------------------------------------------------------------------


module bridge_fifo_plugin(

    pcie_clk,
    plugin_rst,
    din64,
    fifoin_empty,
    fifoin_full,
    plugin_clk,
    dout64,
    fifoin_rd_en,
    fifoout_wr_en,
    fifoout_full

  );
//------------Input Ports-------------
input pcie_clk;
input plugin_rst;
input [63:0] din64;
input [1:0] fifoin_empty;
input [1:0] fifoin_full;
input [1:0] fifoout_full;
//------------Output Ports--------------
output plugin_clk;
output [63:0] dout64;
output [1:0] fifoin_rd_en;
output [1:0] fifoout_wr_en;

//--------------To plugin----------------------
reg data_read;
reg data_written;
wire clear_data_read;      //
wire clear_data_written;    //

//--------------From plugin--------------------
wire update_data;
wire data_ready;
//---------------------------------------------

wire plugin_clk_frm_dcm;

assign plugin_clk = plugin_clk_frm_dcm;
assign fifoin_ready = ( (fifoin_empty==2'b00) ? 1'b1 : 1'b0);
assign fifoin_rd_en = ( (update_data && fifoin_ready ) ? 2'b11 : 2'b00);

assign fifoout_ready = ( (fifoout_full == 2'b00) ? 1'b1 : 1'b0 );
assign fifoout_wr_en = ( (data_ready && fifoout_ready) ? 2'b11 : 2'b00 )
    ;

always @ (posedge plugin_clk)
begin
  if(plugin_rst)
  begin
    data_read <= 1'b0;
    data_written <= 1'b0;
  end
  else
  begin
    if(fifoin_rd_en)
      data_read <= 1'b1;
    else
      data_read <= clear_data_read;    //
    if(fifoout_wr_en)
      data_written <= 1'b1;
    else
      data_written <= clear_data_written; //
  end
end
```

```
//————————————————Data Monitoring Module Clock from DCM
    ————————————————————

wire clkfx;
wire clk2x;
wire clkdv;
wire clkfb;
wire clkin;

assign clkin = pcie_clk;

DCM_BASE #(.CLKIN_PERIOD(10.0),    // Specify period of input clock in ns
     from 1.25 to 1000.00
  .CLKDV_DIVIDE(2),    // Divide by:
      1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5,7.0,7.5,8.0,9.0,10.0,11.0,12.0,13.0,
       or 16.0
  .CLKFX_DIVIDE(2),    // Can be any integer from 1 to 32
  .CLKFX_MULTIPLY(2),    // Can be any integer from 2 to 32
  .CLKIN_DIVIDE_BY_2("FALSE"),  // TRUE/FALSE to enable CLKIN divide by
      two feature
  .CLK_FEEDBACK("1X"),     // Specify clock feedback of NONE, 1X or 2X
  .DFS_FREQUENCY_MODE("HIGH")
  )

  DCM_for_plugin_clk(
      .CLK0(clkfb), // 0 degree DCM CLK ouptput
      .CLKFX(clkfx), // DCM CLK synthesis out (M/D)
      .CLKFB(clkfb), // DCM clock feedback
      .CLKIN(clkin), // Clock input (from IBUFG, BUFG or DCM)
      .RST(plugin_rst) // DCM asynchronous reset input
      );


   BUFG bufg(.I(clkfx), .O(plugin_clk_frm_dcm));  // clock drive buffer

//————————Data Monitoring Wrapper———————

plugin_wrapper_cw plugin (

      .clk(plugin_clk),
      .rst(plugin_rst),
      .data_read(data_read),
      .data_written(data_written),
      .din64(din64),
      .dout64(dout64),
      .update_data(update_data),
      .data_ready(data_ready),
      .clear_data_read(clear_data_read),
      .clear_data_written(clear_data_written)

  );

//---------------------------------------------------------------
endmodule

//---Use this only if plugin_wrapper_cw.ngc file is used for the data
  monitorign plugin
//---else, if using a verilog module i.e plugin_wrapper_cw module,
  comment this module declaration
//--------------Instatiate the plugin module--------------
module plugin_wrapper_cw (
  clk,
  rst,
  clear_data_read,
  clear_data_written,
  data_read,
  data_ready,
  data_written,
  din64,
  dout64,
  update_data
);

  input  clk;
  input  [0:0] rst;
  input  [0:0] data_read;
  input  [0:0] data_written;
  input  [63:0] din64;
  output [0:0] clear_data_read;
  output [0:0] clear_data_written;
  output [0:0] data_ready;
  output [63:0] dout64;
  output [0:0] update_data;
endmodule
```

A.3 Software Process Data Transfer Loop

```
if (ioctl(xbmd_descriptors.g_devFile, INIT_TRANSFER, dmacr_reg) < 0)
  {
    printf("INIT_TRANSFER Failed\n");
    return CRIT_ERR;
  }

  for(int recv_packet = 0; recv_packet < a0; recv_packet++)
  {
    if((recv_bytes = recvfrom(sock, recv_buffer, MESSAGE_SIZE,0,(
        struct sockaddr *) &DP_sock, &DP_sock_len)) < 0)
    {
      printf("\nERROR: Failed to receive message throught he DP_sock\n
          ");
      return -1;
    }
    // Place the contents of the recv_buffer in a FIFO or a Ring Queue
        (Future Work)
  }

  // Update WriteBuffer with received data from the DP Subsystem
  WriteData(xbmd_descriptors.g_devFile, (char*) gWriteData, BUF_SIZE);

  // Update ReadBUffer with previously obtained data from the FPGA
  ReadData(xbmd_descriptors.g_devFile, (char *) gReadData, (wrwdmatlps
      * wrwdmatlpc * 4));

  while(ioctl(xbmd_descriptors.g_devFile, RDDDMACR, &reg_value1) >= 0)
  {
    if(reg_value1 == compare_value)
    {
      break;
    }
    else if((reg_value1 & 0x80000000) != 0)
    {

      if(ioctl(xbmd_descriptors.g_devFile, RDINFIFOCOUNT, &reg_value1)
          >=0)
        printf("\nData available in the IN FIFOS before start of %d
            transfer = %d",ii, reg_value1);
      printf("\nBUFFER OVERFLOW @ %ld Transfer: Increase the
          Computation speed or decrease the Buffer Size or increase
          the PCIe FIFO BUFFER Size",ii);
      return -1;
    }
  }
```

Bibliography

[1] “Data Centric Computing.” URL: https://computation.llnl.gov/casc/dcca-pub/dcca/Data-centric_architecture.html.

[2] S. Ellingson, T. Clarke, A. Cohen, J. Craig, N. Kassim, Y. Pihlstrom, L. J. Rickard, and G. Taylor, “The Long Wavelength Array,” *Proc. IEEE*, vol. 97, pp. 1421–1430, Aug 2009.

[3] S. Ellingson, “Long Wavelength Array Station Architecture Ver.2,” Feb 2009. URL: http://www.ece.vt.edu/swe/lwa/.

[4] C. Wolfe, S. Ellingson, and C. Patterson, “Interface Control Document for Monitoring and Control System-Data Recorder (MCS-DR), Ver. 1.0,” Mar 2010. URL: www.ece.vt.edu/swe/lwavt.

[5] S. Che, J. Li, J. Sheaffer, K. Skadron, and J. Lach, “Accelerating compute-intensive applications with GPUs and FPGAs,” in *Application Specific Processors, 2008. SASP 2008. Symposium on*, pp. 101 –107, June 2008.

[6] B. Cope, P. Cheung, W. Luk, and S. Witt, “Have GPUs made FPGAs redundant in the field of video processing?,” in *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pp. 111 –118, Dec. 2005.

[7] “Comparing FPGAs and DSPs for high-performance DSP applications, BDTI.” URL: http://www.bdti.com/articles/fpga_article.pdf [Nov. 2010].

[8] "The evolving role of FPGAs in DSP applications, BDTI." URL: http://www.bdti.com/articles/fpga_article.pdf [Nov. 2010].

[9] "Virtex-5 FPGA family." URL: http://www.xilinx.com/products/virtex5/index.htm [Nov. 2010].

[10] "Digilent Inc." URL: http://www.digilentinc.com/Products/Detail.cfm?NavTop=2&NavSub=599&Prod=XUPV5 [Nov. 2010].

[11] "FPGA Features." URL: http://www.xilinx.com/company/gettingstarted/index.htm.

[12] "FPGA Design Flow." URL: http://www.xilinx.com/itp/xilinx8/help/iseguide/html/ise_fpga_design_flow_overview.htm.

[13] "Socket Programming in ANSI C: sys/ socket.h." URL: http://www.opengroup.org/onlinepubs/009695399/basedefs/sys/socket.h.html.

[14] "XAPP1052: Bus Master DMA Performance Demonstration Reference Design for the Xilinx Endpoint PCI Express Solutions." URL: http://www.xilinx.com/support/documentation/application_notes/xapp1052.pdf.

[15] "Virtex-5 FPGA Integrated Endpoint Block for PCI Express Designs User Guide." URL: www.xilinx.com/support/documentation/user_guides/ug197.pdf.

[16] "PCIe Base Specification 1.1." URL: http://www.pcisig.com/specifications/pciexpress/base.

[17] "Virtex-5 FPGA RocketIO GTP Transceiver User Guide." URL: http://www.xilinx.com/support/documentation/user_guides/ug196.pdf.

[18] "LogiCORE IP Endpoint Block Plus for PCI Express." URL: http://www.xilinx.com/support/documentation/ip_documentation/pcie_blk_plus_ug341.pdf.

[19] "LogiCORE IP FIFO Generator." URL: http://www.xilinx.com/support/documentation/ip_documentation/fifo_generator_ug175.pdf.

[20] "Personal communication, Christopher Wolfe, Viginia Polytechnic State University, Dec 1, 2010."

Nomenclature

10GbE	10 Gigabit Ethernet
ASIC	Application-Specific Integrated Circuit
ASP	Analog Signal Processor
BDTI	Berkeley Design Technology Inc
BFU	Beam-Forming Unit
BMD	Bus Master DMA Engine
CLB	Configurable Logic Block
CPLD	Completion with Data
DIG	Digitizer
DMA	Direct Memory Access
DP	Digital Processor
DRSU	Data Recorder Storage Unit
DRX	Digital Receiver
DSP	Digital Signal Processor
FFT	Fast Fourier Transform
FIFO	First-In-First-Out
FIR	Finite Impulse Response
FPGA	Field Programmable Gate Array
GPP	General Purpose Processor
GPU	Graphics Processing Unit

HDL	Hardware Description Language
IOB	Input Output Block
IP	Intellectual Property
LUT	Look-Up Table
LWA	Long Wavelength Array radio telescope
MCS-DR	Monitor and Control System Data Recorder
MCS-DR PC	Personal Computer of the Monitor and Control System Data Recorder
MRd	Memory Read
MWr	Memory Write
NCD	Native Circuit Description
NGC	Native Generic Circuit
NGD	Native Generic Database
PC	Personal Computer
PCIe	Peripheral Component Interconnect Express
PIO	Programmable Input Output
REQ	Request
RFI	Radio Frequency Interference
RTL	Register-Transfer Level
RX	Receive
TBN	Narrow-Band Transient Buffer
TBW	Wide-band Transient Buffer
TLP	Transaction Layer Packet
TX	Transmit
UCF	User Constraints File
XSG	Xilinx System Generator
XUPV5	Xilinx University Program board with a Virtex-5 FPGA