Design and Implementation of an FPGA-based Partially Reconfigurable Network Controller

Aditya Prakash Chaubal

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
Computer Engineering

Dr. Peter Athanas, Chair
Dr. Mark Jones
Dr. Cameron Patterson

July 16, 2004
Bradley Department of Electrical and Computer Engineering
Blacksburg, Virginia

keywords: Partial reconfiguration, Network, Virtex, Xilinx, IIM7010, FPGA

Design and Implementation of an FPGA-based Partially Reconfigurable Network Controller

Aditya Prakash Chaubal

Abstract

There is currently a strong trend towards embedding Internet capabilities into electronics and everyday appliances. Most network controllers used in small appliances or for specialized purposes are built using micro controllers. However there are many applications where a hardware-oriented approach using Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs) is more suitable. One of the features of FPGAs that cannot be integrated into ASICs is runtime reconfiguration in which, certain portions of the chip are reconfigured at runtime while the other parts continue to operate normally. This feature is required for network controllers with multiple data transfer channels that need to preserve the state of the static channels while reconfiguration is taking place. It is also required for controllers with space constraints in terms of FPGA resources or time constraints in terms of reconfiguration times. This thesis explores the impact of partial reconfiguration on the performance of a network controller. An FPGA-based network controller that supports partial reconfiguration has been designed and constructed. Partial bitstreams that can configure certain channels of the network controller without affecting the functioning of others have been created. Experiments have been performed that quantify the manner in which, the performance of the controller can be changed by loading these partial bitstreams onto the FPGA. These experiments demonstrated the advantages of using partial reconfiguration to change network-related parameters at runtime to optimize performance of the network controller.

Acknowledgements

First of all, I would like to thank my advisor, Dr. Athanas, for his never ending support, encouragement, guidance and patience. My graduation from Virginia Tech is as much a result of his hard work as it is of mine. I am also indebted to Dr. Patterson and Dr. Jones for their help and technical insights. I would also like to thank the the United States Office of Naval Research (ONR) for supporting the Secure Communications Project at Virginia Tech (and thereby providing me with a means of livelihood).

Special thanks to those who worked with me on the Secure Communications project- Ryan Fong, Jason Zimmerman, Jon Graf, Nikhil Bhatia and Deepak Agarwal. I also appreciate the help and guidance I received from Dr. Scott Harper and Dr. Jae Hong Park.

I would also like to thank my friends at Virginia Tech, especially my roommates who put up with me for two years, for their support.

Above all, I would like to thank my parents whose love and faith in me has always been my inspiration for every activity that I undertake. They were always there for me when I needed them and I cannot imagine myself going through all this without their help and support.

Dedicated to my parents who gave me everything I ever wished for and more.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

There is a strong trend towards embedding Internet capabilities into electronics and everyday appliances because of the wide-spread access to the Internet in today's world. These appliances can vary in their size and applications, ranging from Personal Digital Assistants (PDAs) to small systems such as sensors and actuators that are controlled or accessed over the Internet [1]. Forecasts predict that the sales of Personal Computers (PCs) will be exceeded by the sales of non-PC Internet devices by 2005 [2]. Most network controllers used in small appliances or for specialized purposes are built using micro controllers. However there are many applications where a hardware-oriented approach using Application Specific Integrated Circuits (ASICs) or Field Programmable Gate Arrays (FPGAs) is more suitable [3]. Of these two alternatives, ASICs have a long and therefore expensive development cycle. Customizing ASICs for network protocols that are subject to change or extension is not feasible for most applications. These issues have led designers towards using reconfigurable hardware such as FPGAs that offer both the performance of ASICs and the flexibility of micro controller-based designs.

FPGAs have generally been considered prototyping devices for developing circuits that will

ultimately be implemented on ASICs. While it is true that the development cycle of FPGAs is much shorter than ASICs, FPGAs also have certain unique features that cannot be integrated into ASICs [4]. One of these features is partial reconfiguration, whereby a specific portion of the FPGA can be reconfigured at runtime without affecting the logic in the other parts. This approach is of immense benefit to network controlling devices implemented in hardware that need to support variable network configurations but are also bound by one of the following constraints:

- *Space Constraint*: If the network controller is implemented on an FPGA that has space constraints which do not allow for storing more than one configuration at any given time, partial bitstreams can be created for the various alternate configurations of the network controller. These can be loaded onto the FPGA as required to change the configuration at runtime.

- *Time Constraint*: Partial bitstreams, which program pre-specified sections of the FPGA, are smaller than the bitstreams required for reconfiguring the entire FPGA. Hence they take correspondingly less time for loading. As such, these bitstreams are beneficial in circumstances where the system cannot tolerate the time delay caused by full reconfiguration.

- *Preservation of State*: Partial reconfiguration can be used to initialize additional data transfer channels in a network controller while the existing channels remain fully operational. As a result of this property, partially reconfigurable network controllers are useful in systems where it is important to preserve the state of certain data paths while others are being reconfigured.

As such, it can be seen that there are numerous critical advantages offered by adding the feature of partial reconfiguration to FPGA-based network controllers.

1.2 Contributions

This thesis attempts to explore the impact of partial reconfiguration on the performance of an FPGA-based network controller. The overall contributions of the research that led to this thesis are as follows:

- Design of an FPGA-based network controller that uses minimal silicon resources and that can support partial reconfiguration.
- Implementation of this design on an FPGA and analysis of its performance in terms of network throughput and packet loss.
- Design and creation of partial bitstreams that can configure certain sections of the network controller without affecting the functioning of others.
- Examination of the manner in which the performance of the network controller can be affected by the changes brought about when these partial bitstreams are loaded onto the FPGA.

The FPGA-based network controller designed in this thesis was also used as part of a larger system for transferring FPGA bitstreams over an Ethernet network. Details of this system are included in this thesis in Chapter 6.

1.3 Organization of Thesis

This thesis is organized in the following manner. Chapter 2 gives an overview of the network architecture model that is currently in use and also looks at the latest research being conducted into FPGA-based network controllers. Chapter 3 explains the design of the network controller that was created as part of this thesis and also introduces the hardware that is used for building the controller. Chapter 4 details the actual hardware implementation of the network controller and also provides information concerning performance and resource

utilization of the the FPGA-based design. Chapter 5 deals with partial reconfiguration and the process of creating partial bitstreams for the network controller. Chapter 6 presents the results of various performance tests run on both, the static and partially reconfigurable versions of the network controller. It also gives a brief overview of one of the immediate applications of the network controller. Finally, Chapter 7 summarizes the thesis and discusses the possibilities for future research.

Chapter 2

Background

The function of a network controller is to gather data packets that are addressed to it off a particular network medium and make them available to application software. To do so, it has to follow certain steps that separate the data from a series of headers that make up a network packet. At each of these steps, there are a number of possible protocols that can be used for packetizing and de-packetizing. These steps can be implemented either as programmed instructions in micro controllers or they can be built into hardware on ASICs or FPGAs. A new method of implementing these steps and certain characteristics that result from it form the basis of the work done in this thesis.

This chapter contains an overview of the network architecture model that is used in the design of network controllers. This information is provided in Section 2.1. Section 2.2 examines the different approaches that have currently been adopted for creating network controllers using reconfigurable hardware. Finally, Section 2.3 presents an overview of partial reconfiguration, which is one of the unique features of the network controller that was developed during this research.

2.1 OSI Model

An important characteristic common to all network architectures is the use of a layered approach with regard to the implementation of various functions required for communication between two devices. To manage the complexity of the various devices in the network, the functions performed by the devices are divided into independent functional layers. The architecture that is used currently for networking computers is part of a standard defined by the International Organization for Standardization (ISO). In 1984, the ISO defined an overall model of computer communication called the Reference Model for Open Systems Interconnection or OSI model [5]. This model is meant to define a common basis for the coordination of development in the field of interconnecting open systems. The OSI model does not define the networking software or detailed standards for such software. It simply defines the broad categories of function that each layer should perform.

The OSI model defines seven functional layers as shown in Figure 2.1. This section briefly describes each of the seven layers of the OSI model, starting with the lowermost layer.

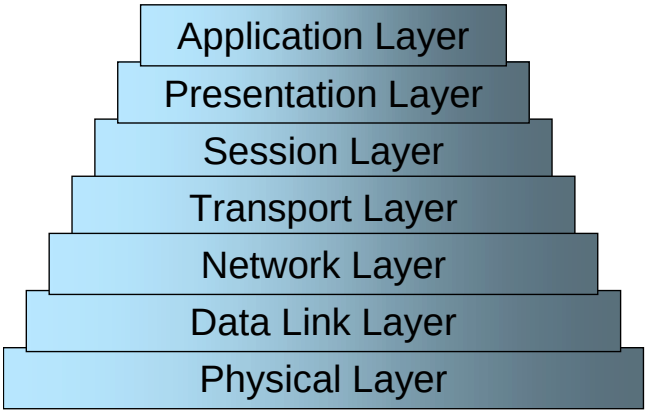

Figure 2.1: OSI Model of network architecture [5]

Physical Layer: This layer is responsible for the actual transmission of a stream of bits across a physical medium such as a wire or a cable [6]. The implementation of this layer and the

protocols associated with it varies according to the actual interconnection medium. Some examples of the types of transmission media are twisted-pair cable, fiber-optic cable and wireless transmission.

Data Link Layer: This layer is responsible for providing data transmission over a single connection from one system to another. Control mechanisms in the data link layer handle the transmission and synchronization of data units, known as frames, over a physical circuit [5]. Functions operating in this layer allow data to be transmitted in a relatively error-free fashion over a sometimes error-prone physical circuit. The data link layer is divided into two sub-layers: the Medium Access Control (MAC) layer and the Logical Link Control (LLC) layer [7]. The MAC sub-layer controls how a computer on the network gains access to the data and permission to transmit it. The LLC sub-layer controls frame synchronization, flow control and error checking.

Network Layer: This layer provides switching and routing technologies, creating logical paths, known as virtual circuits, for transmitting data from node to node. Routing and forwarding are functions of this layer, as well as addressing, internetworking, error handling and packet sequencing [5]. The protocol that is most widely used at this layer is the Internet Protocol (IP). The current version of the Internet Protocol that is in use is Version 4. However, Version 6 of this protocol (IPv6) that features, among other things, an expanded address space to allow for more networked devices, is currently in the process of replacing IPv4.

Transport Layer: This layer provides transparent transfer of data between end systems, or hosts, and is responsible for end-to-end error recovery and flow control. It ensures complete data transfer [6]. It hides from the higher layers all the details related to the actual moving of packets and frames from one device to another. This layer can also be used to provide delivery of packets in the proper sequence for those applications requiring such functionality. Protocols such as the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) are used at this layer.

There is a fundamental difference between the bottom four layers discussed thus far and

the top three layers of the OSI model. The bottom four layers are concerned with the network itself and provide a general data transport service useful to any application. The top three layers are more concerned with services that are oriented to the application programs themselves [7]. As such these three layers are sometimes referred to as 'Application Program Services' as shown in Figure 2.2.

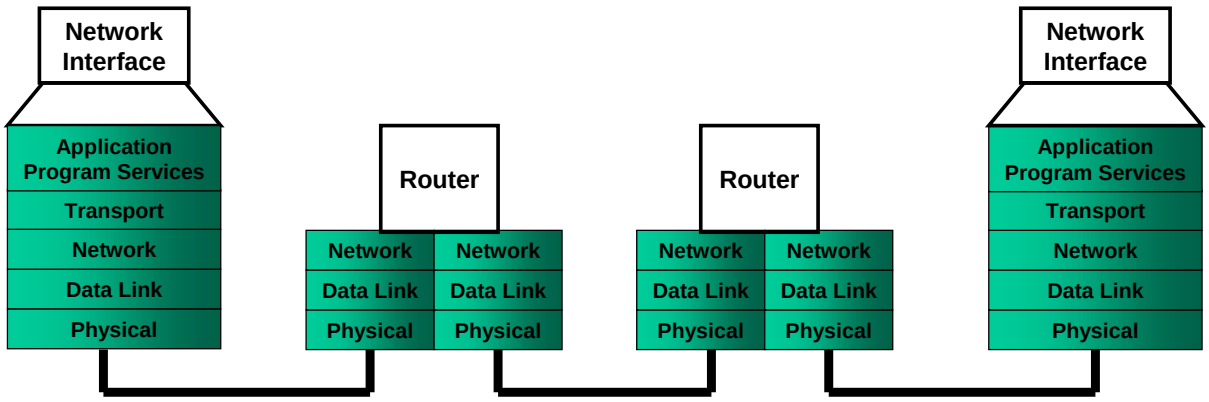

Figure 2.2: Network communication across multiple links [5]

Application Program Services: The first of the three layers included in this category is the *Sessions layer*. This layer is responsible for organizing the dialog between two application programs and for managing the data exchanges between them. The next layer is the *Presentation Layer*, that provides a standard representation for sending data across the network and hence frees the application layer from having to contend with compatibility problems related to data representation. The top-most layer of the OSI model is the *Application Layer*. It supports applications and end-user processes. This layer provides application services for file transfers, e-mail and other network software services [5].

Based on the OSI model of network architecture, there are two identifiers associated with any given device on a network. These identifiers are necessary for routers to forward packets to their proper destinations over the network. They are:
MAC Address: Also called Physical Address or Ethernet Address, this identifier is unique

to a particular network controller. No two network controllers may have the same MAC address. In most networks, this address is six bytes long. This address is associated with the data link layer [8].
IP address: This address is associated with the TCP/IP protocol family that is used at the network and transport layers. It consists of four bytes and is used to identify a particular machine on a network [8].

2.2 Current Network Controller Implementations

The implementation of the physical and data link layers of the OSI model is uniform in all network controllers for a particular Local Area Network (LAN) technology. Numerous options are available today for providing the functionality of these layers, each of which is based on a different LAN technology. Widely used LAN technologies include Ethernet, Fiber Distributed Data Interface (FDDI) and token ring. Among these, Ethernet is the most popular and is used in a vast majority of corporate networks and numerous home networks [7]. The Ethernet technology is defined by the IEEE standard 802.3, which defines the physical layer specifications and the Carrier Sense Multiple Access with Collision Detection (CSMA/CD) access method for Ethernet [9]. It includes necessary specifications for the following family of systems: 10 megabit per second (Mb/s) baseband system, 100 Mb/s baseband system and 1000 Mb/s baseband system. Out of these, the 10 Mb/s Ethernet LAN technology is currently the most widely used [7]. Any particular network controller can be interfaced only with a network belonging to a particular type of LAN technology. This is because characteristics of the network such as physical medium and frame size used for transmission differ from one implementation to another, as was noted in the previous section [5]. The network controller designed and built as part of this thesis is for the 10 Mb/s Ethernet LAN technology.

As of now, most Ethernet network controllers are used for connecting computers to a network and hence take the form of Network Interface Cards (NIC). However, as pointed out in Chapter 1, there are an increasing number of network controllers that are embedded into

smaller devices to provide them with networking capability. These network controllers differ from NICs with regards to the manner of implementation of the various layers of the OSI model. NICs handle only the data link and physical layers. The higher layers of the network architecture are handled by the computer that houses the NIC. Different applications in a computer may need to use different protocols for higher network layers and hence they are not built into the hardware. The network controllers in smaller devices, on the other hand, can be specialized depending on their application. These controllers generally provide the functionality of the network and transport layers in the form of a TCP/IP stack in hardware, in addition to the data link and physical layers.

As explained in Chapter 1, designers have been gradually moving away from micro controllers and ASICs and towards FPGAs for creating flexible, high-performance network controllers. A significant amount of research in academia has been aimed at designing Ethernet network controllers based on FPGAs as can be seen from the examples shown in this section. This section examines some of the FPGA-based network controllers successfully created by researchers. It also analyzes the resources available in the open source community for implementing a network controller on an FPGA.

2.2.1 Network Controller for Small Internet-connected Appliances

Because of the growing demand for small, low cost, low power and flexible computing devices that attach to the Internet, researchers have created the prototype of a platform that connects an FPGA directly to the Internet [2]. The platform hardware in their design consists of a Xilinx Virtex XCV100 FPGA and an interface device for physically connecting to the network. The MAC core that is placed in the FPGA supports collision-detect, exponential back off and formats frames according to the Ethernet Maximum Transfer Unit (MTU). Block RAMs in the FPGA were used to create the transmitter and receiver buffers. The authors estimate that their design occupies approximately 500 Configurable Logic Blocks (CLBs) and a total of nine block RAMs. Despite the fact this design is not optimized for speed or size, it is significant because it demonstrates that generic FPGA logic can be used to provide network interfaces that are usually designed using a general-purpose processor.

However, it fails to provide any major advantages which could prove that the FPGA-based solution is better than a micro controller-based one.

2.2.2 Network Controller for Cluster Computing Purposes

Another application that suits the characteristics offered by FPGA-based network controllers is cluster computing. Researchers have demonstrated the feasibility and flexibility of FPGA-based communication architecture for use in cluster computing [10]. Because of the high overhead of generic protocols like TCP/IP, they developed four custom protocols for use in this project. These protocols were implemented on Xilinx XC4085XLA FPGAs. Table 2.1 compares the functionality of the various protocol configurations and the space used by them on the FPGA.

Protocol	*Reliable*	*Packets ordered*	*Duplicate elimination*	*Number of flip-flops*	*Number of LUTs*
Configuration 1	No	No	No	1643	1977
Configuration 2	Yes	No	No	2165	3380
Configuration 3	Yes	No	Yes	2534	3784
Configuration 4	Yes	Yes	Yes	2650	3933

Table 2.1: Comparison of protocols used in cluster computing [10]

The key characteristic of the architecture developed in this research is the ability to modularize a communications protocol and selectively use those modules to form specialized protocols.

2.2.3 Evolvable Reconfigurable Internet Platform

Research focused on the development of an evolvable Internet hardware platform is also being carried out at the Applied Research Lab at Washington University [3]. The Field Programmable Port Extender (FPX) system uses FPGAs to allow reconfigurable hardware

modules to be dynamically installed into network devices. An FPX module contains two FPGA devices. One of these is a static Network Interface Device (NID) FPGA, which is a Xilinx Virtex XCV600E part. The other is a single Reconfigurable User Application Device (RAD), which is a Xilinx Virtex XCV1000E part. Figure 2.3 shows a block diagram of the FPX system.

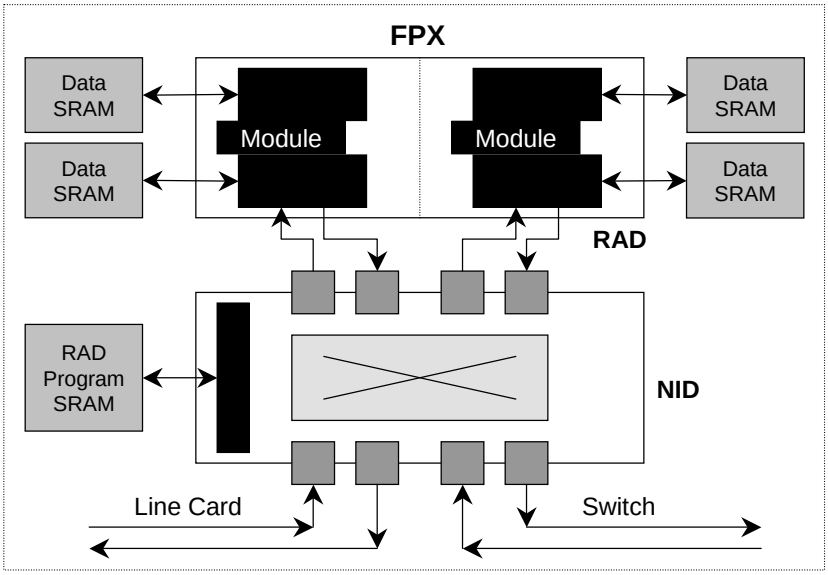

Figure 2.3: Field Programmable Port Extender System [3]

The FPX modules are inserted into an existing network stream to provide user-specified traffic processing including routing, buffering, and packet content modification. Recent work on this system has included the development of an interface that allows FPX modules to be programmed remotely via a TCP/IP network.

2.2.4 Gigabit Rate IP Security

Researchers at the University of Southern California have created an architecture that shifts a significant amount of network processing from the host CPU onto a network interface [11]. This system uses a Gigabit Ethernet MAC along with an FPGA platform to create the

prototype Gigabit Rate IP Security (GRIP) card. The FPGA platform used is the SLAAC-1V platform [12] that has three Xilinx XCV1000 FPGAs. Figure 2.4 shows a block diagram of the GRIP prototype.

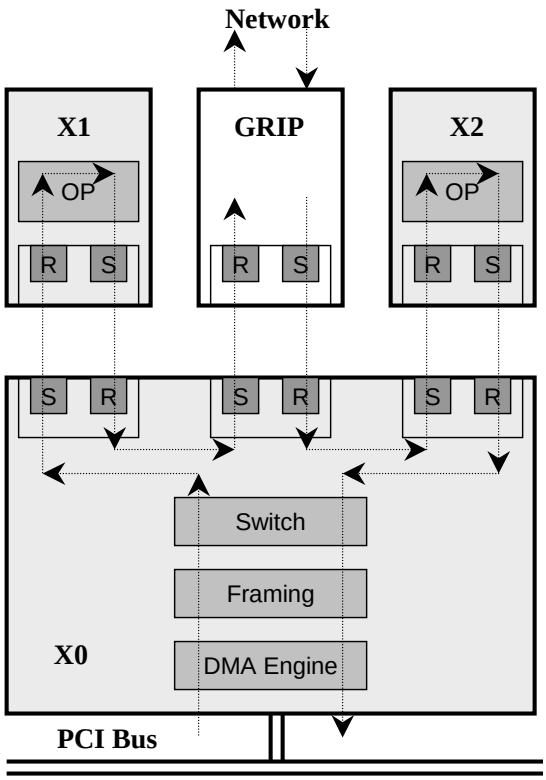

Figure 2.4: Gigabit Rate IPSec System [3]

In Figure 2.4, the X0 FPGA does the packet header processing function, while the X1 and X2 FPGAs contain an encryptor and decryptor for outbound and inbound packets respectively. Applications other than encryption and decryption can also be performed in those FPGAs. This work shows the feasibility of using reconfigurable hardware in the creation of high performance network controllers.

2.2.5 Open Source Ethernet MAC Core

To gain an understanding of current benchmarks for Ethernet controllers on FPGAs with regards to resource utilization and performance standards, it was decided to examine an FPGA-based network controller implementation that was available in the open source community. Hardware Descriptor Language (HDL) source code for the Ethernet 10/100 Mbps MAC core was obtained from the database of the configurable hardware group 'Open Cores' [13]. Figure 2.5 provides a block diagram showing the general architecture of the entity that was synthesized by the code.

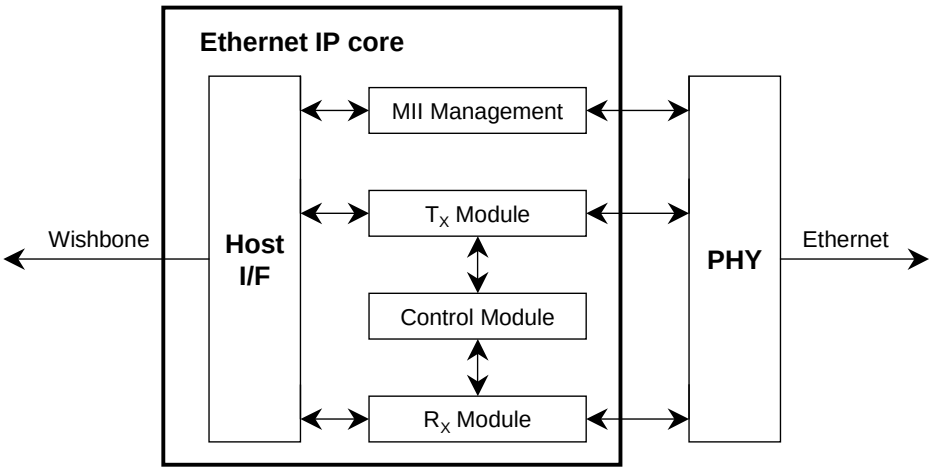

Figure 2.5: Block diagram of Ethernet 10/100 Mbps MAC core [13]

The functions of each of the blocks shown in Figure 2.5 are:

- *TX and RX Modules*: These modules provide full transmit and receive functionality. Cyclic Redundancy Check (CRC) generators are incorporated in both modules for error detection purposes. The modules also handle preamble generation and removal. The preamble field is necessary for synchronizing the clocks of the transmitting and receiving network controllers.
- *MAC control Module*: The control module provides full duplex flow control. Flow

control is achieved by transferring pause control frames between the communicating stations.

- *MII Management Module (MIIM)*: This module provides the standard IEEE 802.3 Media Independent Interface (MII) that defines the connection between the physical and data link layers.
- *Host Interface*: The host interface from the MAC core has a 32-bit bus width and is generally connected to a Reduced Instruction Set Computer (RISC) or some micro controller.

When this design was synthesized on a Xilinx Virtex XCV1000 FPGA, it was able to run at a maximum speed of about 33 MHz. With regards to resource utilization, it used four block RAMs and 2799 Look-Up Tables (LUTs).

Apart from the designs mentioned in this section, there are several other FPGA-based network controllers that are being developed for use in specialized applications; however these systems do not offer any unique characteristics that cannot be built into controllers designed using micro-controllers or on ASICs. The network controller designed as part of this research offers the property of partial reconfiguration that is unique to FPGAs. The basic concepts involved in partial reconfiguration of FPGAs are discussed in the next section.

2.3 Partial Reconfiguration

2.3.1 Overview

One of the features of the Xilinx Virtex architecture is the ability to reconfigure a portion of the FPGA while the remainder of the design is still operational. Partial reconfiguration is useful for applications that require the loading of different designs into the same area of the device or the flexibility to change portions of a design without having to either reset

or completely reconfigure the entire device. This capability leads to a host of new possibilities with regard to applications implemented on FPGAs. Some of the benefits of partial reconfiguration are:

- *Runtime reconfiguration*: Using a technique called Active Partial Reconfiguration, parts of an FPGA can be reconfigured while other parts continue to function as normal. This is especially useful for applications such as adaptive hardware algorithms that need to changed at runtime.
- *Efficient use of FPGA resources*: If it is possible to swap hardware configurations in the FPGA, it is not necessary to have all configurations loaded into the FPGA at startup. Since partial reconfiguration facilitates swapping configuration, it leads to a reduction in the number of entities present in the FPGA at a given time. As a result of this, more FPGA resources are available for carrying out other functions.

There are two main flows for partial reconfiguration: Module-based and Difference-based.

- *Module-Based*: This method of partial reconfiguration involves separating the design into a static module and multiple reconfigurable modules [14]. Based on constraints given in the User Constrain File (UCF), these modules are confined to particular sections in the FPGA. These constraints and the method used for generating them are discussed in detail in Chapter 5. The Module-Based partial reconfiguration flow is then used to produce partial bitstreams for each of the individual modules .
- *Difference-Based*: This method of partial reconfiguration is accomplished by making a change to a design using software such as the Xilinx FPGA Editor [15] and then generating a bitstream based on the difference between the initial and final design [14]. This flow is generally used only when a small change needs to be made.

The partial bitstreams created for the network controller as part of this thesis are generated using the Module-based partial reconfiguration flow.

2.3.2 Partial Reconfiguration of a Network Controller

There are a number of advantages of incorporating the property of partial reconfiguration into a network controller. It gives the controller the ability to activate additional channels for data transfer without halting the ones that are already in operation. Without partial reconfiguration, the only way to change the number of operational channels would be to reprogram the whole FPGA, which would necessitate stopping channels that had previously been initialized. Similarly, partial reconfiguration allows the network controller to switch the protocols used for communication at runtime. For instance if one of the channels is initialized with the TCP protocol and needs to switch to UDP at a later time, this can be done by creating a partial bitstream for that channel which would re-initialize it in UDP mode. Additionally, several network related parameters such as the maximum packet size used for transmitting UDP and TCP packets can be changed dynamically in response to network conditions by partially reconfiguring the affected channels.

Chapter 5, which details the generation of partial bitstreams, provides more information about the various features based on partial reconfiguration that have been incorporated into the network controller designed in this research.

2.4 Summary

This chapter provided a brief overview of the OSI model of network architecture and explained the various layers that constituted it. A number of different research projects that have used reconfigurable hardware for implementing network controllers were examined. All of these projects have resulted in prototype products that showcase the benefits of using FPGAs for building network controllers. However, these designs do not demonstrate features that cannot be implemented in other forms of hardware such as ASICs. One such feature is partial reconfiguration, which forms the basis of the work done in this research. The final section of this chapter introduced the concept of partial reconfiguration and enumerated the benefits that it can provide to a network controlling device.

Chapter 3

System Design and Platform Overview

As mentioned in Chapter 2, the network controller designed and built in this research has the property of partial reconfiguration. To implement partial reconfiguration, a different approach had to be taken to the design of the controller from the ones described in the previous chapter. Section 3.1 examines the design of the network controller developed in this research and the manner in which it differs from those discussed earlier. This chapter also examines the architecture of the various hardware components that were used to construct the Ethernet network controller. Section 3.2 provides details of the hardware that implements the physical and data link layers of the controller while Section 3.3 gives the specifications of the FPGA platform that was used to control protocols operating at the higher network layers. A different FPGA platform was used to implement the partial reconfiguration aspect of the network controller. The details of this platform are documented in Section 3.4.

3.1 Network Controller Design

The prototype architectures discussed in Section 2.2 construct FPGA-based network controllers by implementing the protocols that handle the transport, network and data link layers in the FPGA. Only the physical layer is handled by separate hardware. In the architecture developed in this research, however, the data link layer as well as the physical layer are handled by dedicated hardware. Because of this, the FPGA only needs to control the transport and network layer protocols. Additionally, since the special-purpose ASIC that handles the data link layer has built-in hardware for implementing some of the transport and network layer functions, these functions do not need to be rebuilt on the FPGA. This reduces the complexity of the design in the FPGA and also reduces space consumption. Both these factors are crucial if partial reconfiguration is to be incorporated into the design. On the negative side, handling more functions in hardware reduces the flexibility of the design by limiting its ability to handle different protocols. Figure 3.1 illustrates the design of the network controller and compares it to previous implementations.

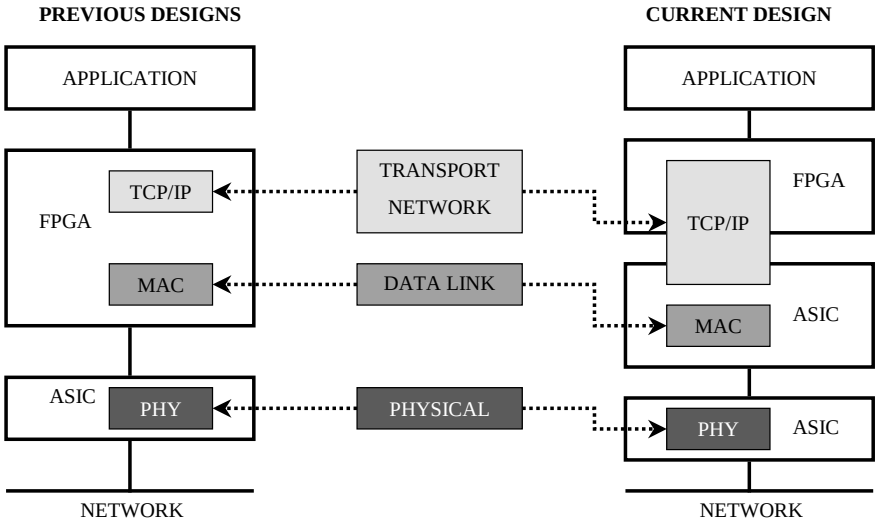

Figure 3.1: Network Controller Design

As explained in the OSI model of network architecture [5], the data link layer is responsible for providing data transmission over a single connection from one system to another. Assuming that the physical medium of the network and the LAN technology being used remain the same, the implementations of the physical and data link layers will not change [6]. Since the network controller being designed is only meant for connecting with a 10 Mbps Ethernet network, it can have static implementations of these two layers. This is the basic idea behind handling the data link layer in an ASIC in this system. Implementing the MAC core in hardware along with some of the functions of the TCP/IP protocol family leads to a number of advantages for the overall system. They are:

- *Minimal Resource Utilization*: Placing an Ethernet MAC core in an FPGA uses up a significant amount of resources on that device as shown in Section 2.2.5. It requires a significant number of CLBs as well as block RAMs. When transport and network layer functions are implemented along with the MAC core, it will take up even more space. Because of this, it may not be possible to build other complex applications on the same FPGA that houses an Ethernet network controller implemented in this fashion. If the MAC core is placed on separate hardware, it frees up a lot of FPGA resources.

- *Increase in Speed*: The open source Ethernet MAC core [13] can run at a maximum speed of 33 MHz in a Xilinx XCV1000 FPGA, as mentioned in Section 2.2.5. Since the MAC core is in an ASIC in the current design, it leads to an overall speed advantage. This was verified experimentally after the network controller had been implemented on a Xilinx XCV1000 FPGA as shown in Section 4.5 of the next chapter.

- *Partial Reconfiguration*: The interface to any implementation of an Ethernet MAC core will contain a lot of signals because of the specifications of the Media Independent Interface. [13]. Such a large number of interconnections poses a problem while creating the separate modules required for partial reconfiguration. Placing the MAC core on a different piece of hardware reduces the number of interconnections in the interface with the FPGA and thereby facilitates the creation of partial bitstreams.

The next sections will focus on the various hardware resources used for the building the

network controller.

3.2 Platform for implementing the Physical and Data Link layers

Since the physical and data link layers are implemented on different devices, it was initially thought that two different pieces of hardware would be required with a custom interface for communicating between them. After an extensive product search, however, a product was obtained that contained hardware for handling both layers. This product was the IIM7010 network module from the Wiznet Corporation [16]. This section deals with the hardware make-up of the IIM7010 module. Figure 3.2 shows a block diagram of the IIM7010. The components that make up the module and their functions are:

- *W3100A from Wiznet Corporation*: Data link layer (MAC)
- *RTL8201L from RealTek Corporation*: Physical layer (PHY)
- *RJ45 connector with transformer*: Connection to the physical medium

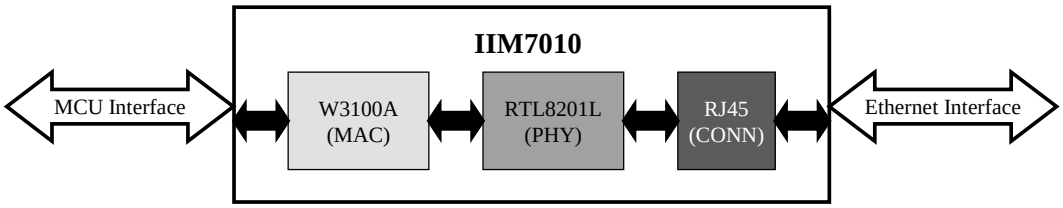

Figure 3.2: Block diagram of the IIM7010 Ethernet module [16]

3.2.1 W3100A

The W3100A is an LSI (Large Scale Integration) of hardware protocol stacks that can provide high-speed Internet connectivity for digital devices [17]. It contains hardware implementations of the MAC protocol and the LLC protocol, which together make up the data link

layer. It also contains hardware to handle some functions of higher-level protocols such as TCP and UDP. Some of the features of the W3100A that are important from the point of view of the network controller design are [17]:

- The W3100A supports four independent connections simultaneously. This facilitates the operation of multiple communication channels using either the same or different transport layer protocols.
- It has an internal implementation of ICMP (Internet Control Message Protocol), which is needed to respond to 'Ping' commands. It also has an internal implementation of the ARP (Address Resolution Protocol) that is used to map IP network addresses to hardware addresses. Since these protocols are implemented in hardware, they do not need to be rebuilt on the FPGA.
- The W3100A has 16 Kilo Bytes (KB) of internal dual-port SRAM for receive and transmit buffers. The 8 KB available for each buffer can be divided as desired among the four channels.
- It offers an interface for connecting to Intel/Motorola micro-controllers. This is the interface that is used to communicate with the FPGA.
- The interface from the MAC core to the hardware for the physical layer is MII (Media Independent Interface). In the IIM7010 module, this interface of the W3100A is connected to the RTL8201L physical layer chip.
- It has a 3.3V internal operation with 5V tolerant 3.3V I/O pads. The operational voltage is important because this device has to be powered by the FPGA platform.

Figure 3.3 gives a block diagram illustration of the W3100A.

From the point of view of the network controller design, the interface that is most important is the one that is connected to the FPGA. It is referred to as the MCU interface since it is generally connected to a Micro Controller Unit. This interface consists of the clock (CLK) and reset (RST) signals along with a 15-bit wide address (ADDR) bus and a bi-directional

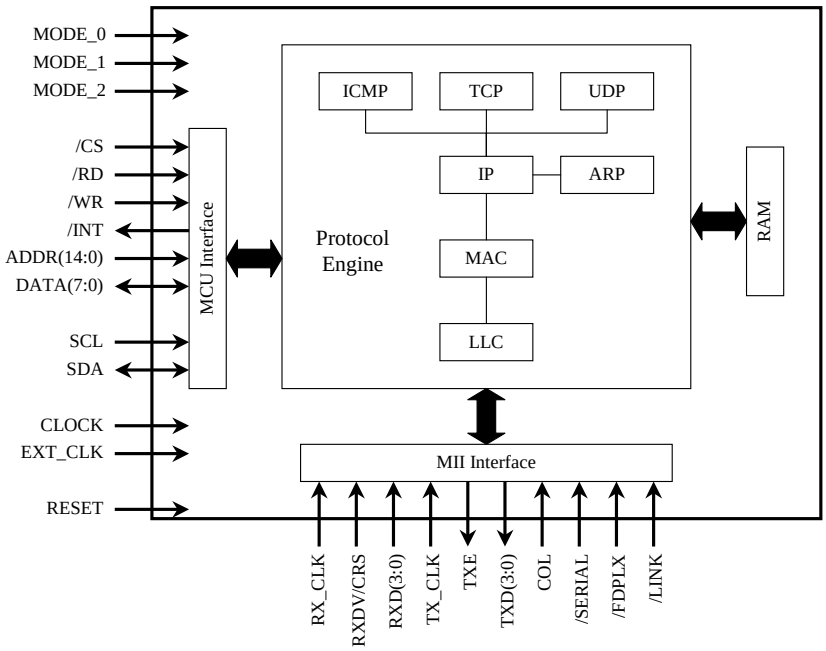

Figure 3.3: Block diagram of the architecture of the W3100A chip [17]

8-bit wide data (DATA) bus. The reset signal is active high. An interrupt (INT) signal is provided to alert the FPGA when there is data in the receive buffer. The chip select (CS), read (RD) and write (WR) signals are meant for use during the memory read/write cycles as will be explained in detail in Chapter 4. These three signals are active low. The SCL signal is the clock used by the I^2C interface when the device is in the I^2C mode, while the SDA signal is the data signal in that mode. Both of these signals require an external pull-up resistor of 4.7 kΩ. The three mode select input pins (MODE[2:0]) are used to select the MCU interface and operating mode of the W3100A. Table 3.1 shows the different modes and their corresponding mode select values.

Memory map: The memory of the W3100A consists of 8-bit wide registers [17]. Figure 3.4 shows the organization of the memory. The address space from 0x0000 to 0x00FF consists of 256 control registers. The next 256 locations from 0x0100 to 0x01FF are pointer registers. The registers from 0x0200 to 0x3FFF are not used by the W3100A and can be used by other

M2	*M1*	*M0*	*Description*
0	0	0	Clocked mode: MCU bus signals are analyzed using clock
0	0	1	External clocked mode: MCU bus signals are analyzed using external clock
0	1	0	Non-clocked mode: MCU bus signal are analyzed directly by W3100A
0	1	1	I^2C mode
1	X	X	Test mode

Table 3.1: Modes of operation of the W3100A [17]

devices. The remaining address space is divided equally between the transmit buffer (0x4000 - 0x5FFF) and the receive buffer (0x6000 - 0x7FFF). Each of these buffers are 8 KB in size.

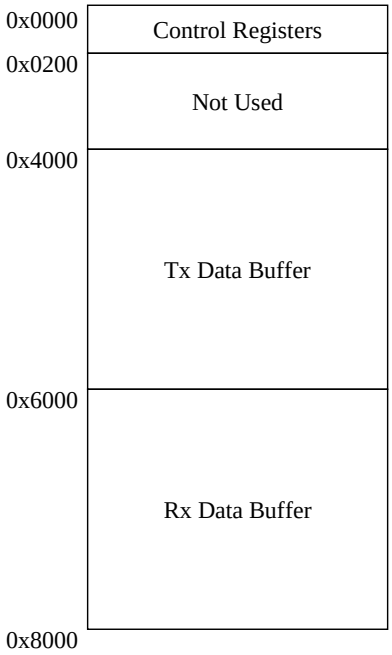

Figure 3.4: W3100A top-level memory map [17]

The control registers in the W3100A are categorized as follows:

- *Command, status and interrupt registers*: These include command registers (CR) and interrupt status registers (ISR) for each of the four channels. There is also an interrupt register (IR) that is used to find out which channel caused the interrupt and an interrupt mask register (IMR) for masking the interrupts [17].

- *System registers*: These include registers used for setting up system parameters such as the IP address, the hardware address (MAC address), the gateway address and the subnet mask address [17]. There are also registers that control the manner in which transmit and receive buffer space is divided between the four channels.

- *Pointer registers*: The W3100A has four-byte pointer registers that are used to obtain the location of receive and transmit pointers. Each pointer register has a shadow register associated with it, which must be accessed before the value stored in the register can be read [17]. The details of this procedure are included in the next chapter.

- *Channel registers*: This category includes registers that are associated with each of the four channels. The most important of these are the socket state registers (SSR) and the socket option and protocol registers (SOPR) [17]. There are also additional registers that store the values for various network parameters such as IP address and port number of the destination computer, maximum segment size etc.

3.2.2 RTL8201L

The RTL8201L is a single chip physical layer transceiver with a Media Independent Interface. It implements all Ethernet physical-layer functions including the Physical Coding Sublayer (PCS), Physical Media Attachment (PMA), 10 Base-T_X Encoder/Decoder and Twisted Pair Media Access Unit (TPMAU) [18]. This chip requires an oscillator running at 25 MHz. This is ideal for interfacing with the W3100A because the default frequency for the clocked mode of that device is 25 MHz [17]. As such, both chips can derive clocks from a single oscillator without extra circuitry for changing frequency. The RTL8201L supports a Media Independent Interface for Ethernet logic. Since this interface is available directly from the W3100A, there is no circuitry required for connecting the W3100A to the RTL8201L.

3.2.3 IIM7010 Overview

Sections 3.2.1 and 3.2.2 specified the hardware details of the W3100A and RTL8201L chips that are used in the IIM7010. However, the interface to the IIM7010 module as a whole is the only one that is visible to the network controller. Hence it is important to study the interface offered by the IIM7010. The IIM7010 has a 25 MHz oscillator that drives the clock signals to both the W3100A and the RTL8021L [16]. Additionally, the 'Mode' signals on the W3100A have fixed values so that the only mode in which the W3100A can operate is the 'Clocked' mode. The actual IIM7010 interface to the FPGA consists of the following signals [16]:

- Control signals: RST (reset), INT (interrupt), CS (Chip Select), RD (Read), WR(Write)
- Data signals: DATA[7:0] (8-bit wide data bus)
- Address signal: ADDR[14:0] (15-but wide data bus)

Another characteristic of the IIM7010 that needs to be considered is its power consumption. This is important because the IIM7010 module will derive its power from the FPGA platform that contains the network controller. According to the IIM7010 data sheet [16], the average current usage for the W3100A when connected to a 10 Mbps network is 9 mA, whereas the average current usage for the RTL8021L for a 10 Mbps network is 129 mA. Thus the average power consumption of the IIM7010 module for a 10 Mbps network is calculated as follows:

$$\begin{aligned} Power &= Voltage * Current \\ &= 3.3V * (9mA + 129mA) \\ &= 3.3V * 138mA \\ &= 455mW = 0.455Watts \end{aligned}$$

Similar calculations yielded the maximum current requirement for the system to be about 135 mA and the maximum power consumption to be about 0.479 Watts. These figures were considered when deciding how many VCC and ground pins on the FPGA platform need to be used for powering the IIM7010. The next section presents the FPGA platform that was used for creating the network controller.

3.3 SLAAC-1V FPGA platform

The network controller on the FPGA was implemented on the SLAAC-1V platform from the Information Science Institute (ISI) at the University of Southern California [12]. The reason for this was that one of the applications of the network controller was the Secure Communications System, which was built on the SLAAC-1V platform [19]. The functionality of the network controller as part of this system is elaborated upon in Chapter 6. This section gives a brief overview of the architecture of the SLAAC-1V and the manner in which it was interfaced to the IIM7010 module. SLAAC stands for *Systems Level Applications of Adaptive Computing*. The SLAAC-1V board consists of three Xilinx Virtex XCV1000 FPGA devices [20]. These devices are labeled X0, X1 and X2 in Figure 3.5 and are referred to as Programming Elements (PE).

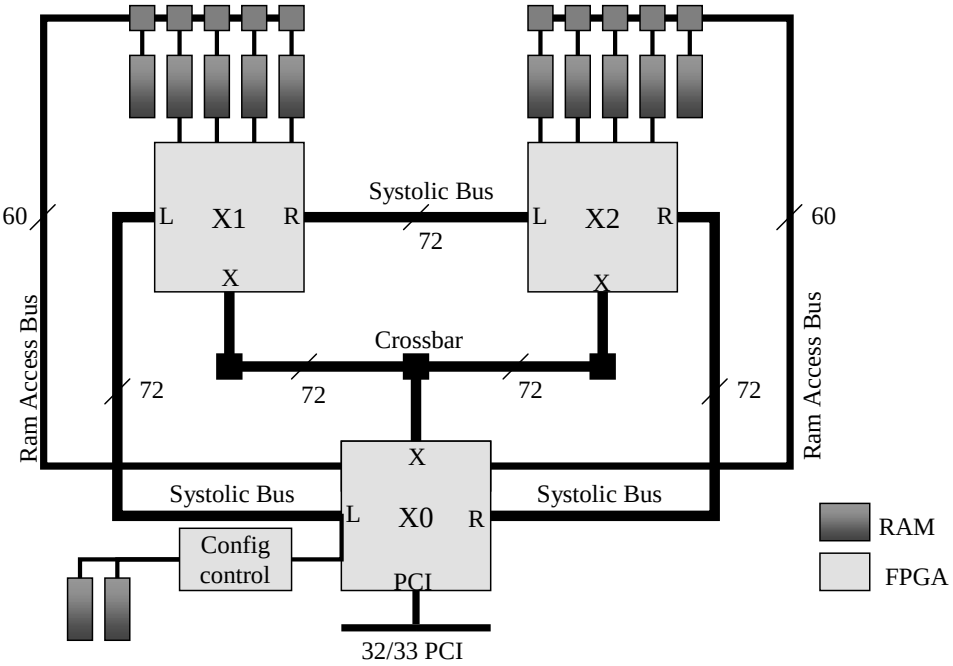

Figure 3.5: SLAAC-1V FPGA platform block diagram [12]

Figure 3.5 also shows the SRAMs associated with each of the PEs. Each of the memory banks consists of a 256K*36-bit Random Access Memory (RAM). The PEs X1 and X2 have four of these memory banks associated with them [12]. The controlling PE X0 has only two memory banks directly associated with it. X0 also has control over the other sets of memory banks and can swap the contents of those memory banks with its own. For communication between the FPGAs, there is either a 72-bit wide crossbar that allows broadcasting information or a 72-bit wide systolic array [21]. The crossbar also provides access to expansion headers on the SLAAC-1V board. The network controller was targeted towards the X2 FPGA since this has a large number of I/O pins that have been mapped to a header. To connect the IIM7010 to the SLAAC-1V, a connecting 'daughter' board was designed and constructed. The Eagle software [22] was used to create the schematic and the layout for the board. It also created Gerber files that were provided as input to a Model 5000 Quick Circuit prototyping system from T-Tech [23] to mill the daughter board. Apart from housing the IIM7010 module, the board has other circuitry and components that are related to the Secure Communications project as can be seen from Figure 3.6.

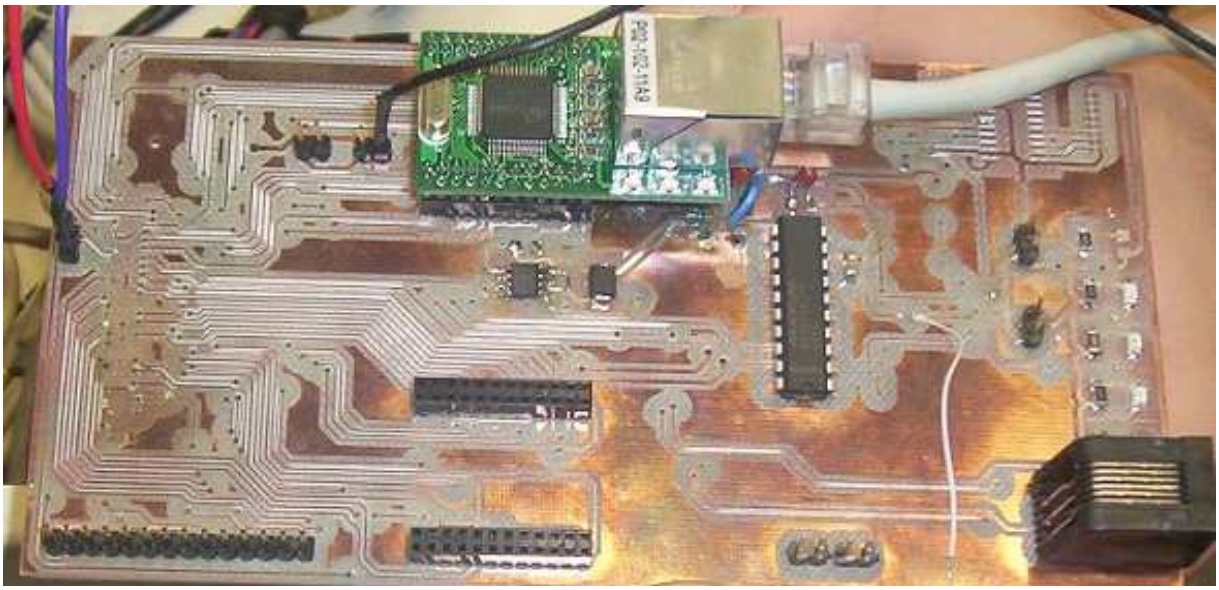

Figure 3.6: Connector board for the SLAAC-1V platform

All the power required for the IIM7010 module was drawn from the SLAAC-1V board. Based on the power requirements calculated in the preceding section, a number of VCC and ground pins were taken together to create power and ground planes so that the pins would be able to source and sink the maximum amount of current. The bitstreams that were produced for the network controller were downloaded from the host to the X2 FPGA over the PCI bus using SLAAC API calls [21].

After the network controller was designed and built on the SLAAC-1V platform, it was decided that the Xilinx Virtex-II architecture [24] would be better suited for creating partial bitstreams. Because of this another FPGA platform was selected that had Xilinx Virtex-II FPGAs. The next section briefly describes the architecture of this new platform.

3.4 DN3000k10 FPGA Platform

The platform chosen for creating the partial bitstreams for the network controller was the DN3000k10 from the DINI group [25]. Figure 3.7 shows a block diagram of the platform.

As can be seen from Figure 3.7, the DN3000k10 contains five Xilinx Virtex-II XC2V4000 FPGAs [24]. Different versions of this platform can have different FPGAs based on the requirements. It also has five external memories; four 36-bit SSRAMs and one 72-bit SDRAM [25]. Like the SLAAC-1V platform, the DN3000k10 can be hosted in a 32/64-bit PCI/PCI-X slot, or can be used as a stand-alone device. A total of 437 test pins are provided on the top of the circuit board using high-density connectors for logic analyzer-based debugging. These high-density connectors can be interfaced with the DN3000k10SD daughter card. This card allows external connection to the signals coming from the FPGAs. The network controller design was targeted to FPGA A on the DN3000k10 platform. To interface the IIM7010 module with that FPGA, another connector board had to be constructed. This board connects to the DN3000k10SD daughter card. It was designed and built using the same tools that were used previously for the SLAAC-1V connector board. Figure 3.8 shows a picture of the interface board made for connecting to the IIM7010 to FPGA A on the

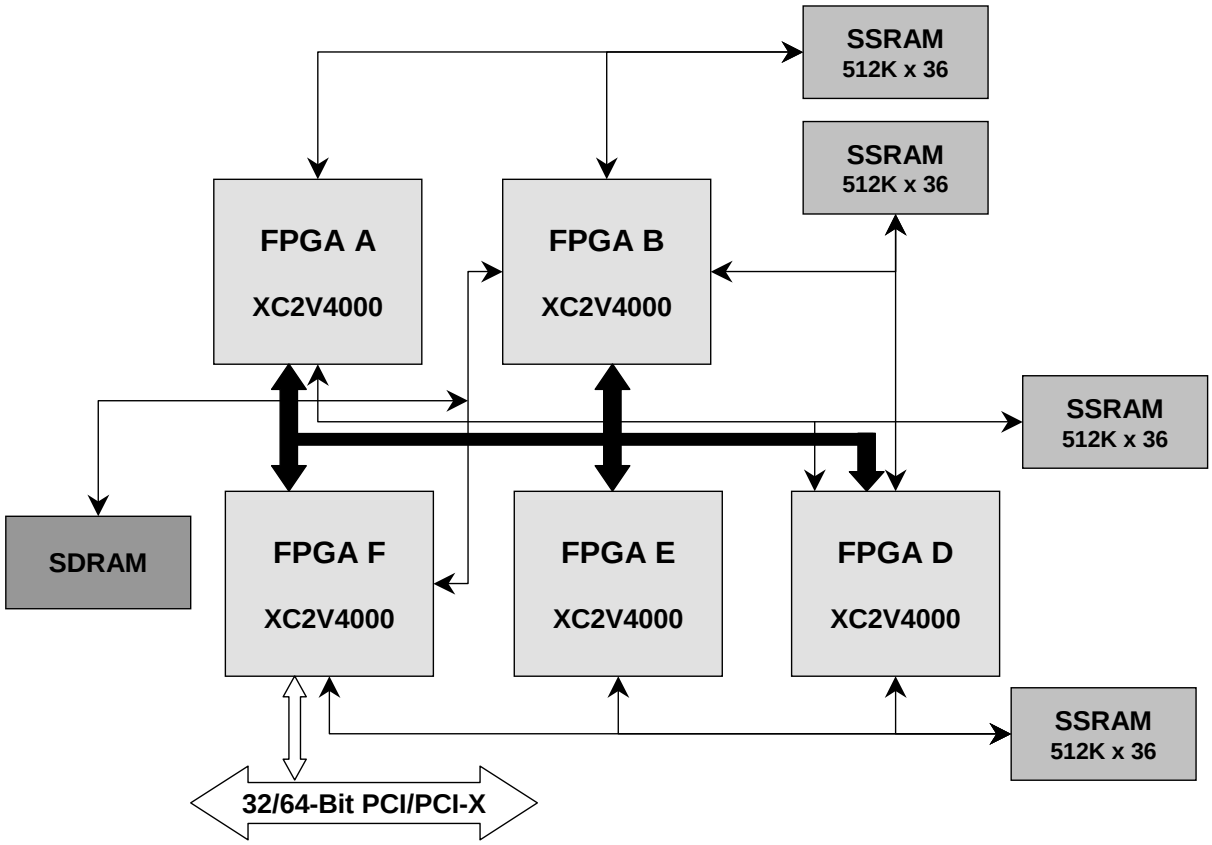

Figure 3.7: DN3000k10 FPGA platform block diagram [25]

DN3000k10.

As can be seen from Figure 3.7, only FPGA F is directly accessible to the host computer over the PCI bus. However, this FPGA could not be used for the network controller since it did not have enough I/O pins mapped to the headers. So FPGA A was programmed by sending the bitstream over the PCI bus and through FPGA F [1].

[1]The program for configuring FPGA A over the PCI bus through FPGA F was written by Dr. Peter Athanas.

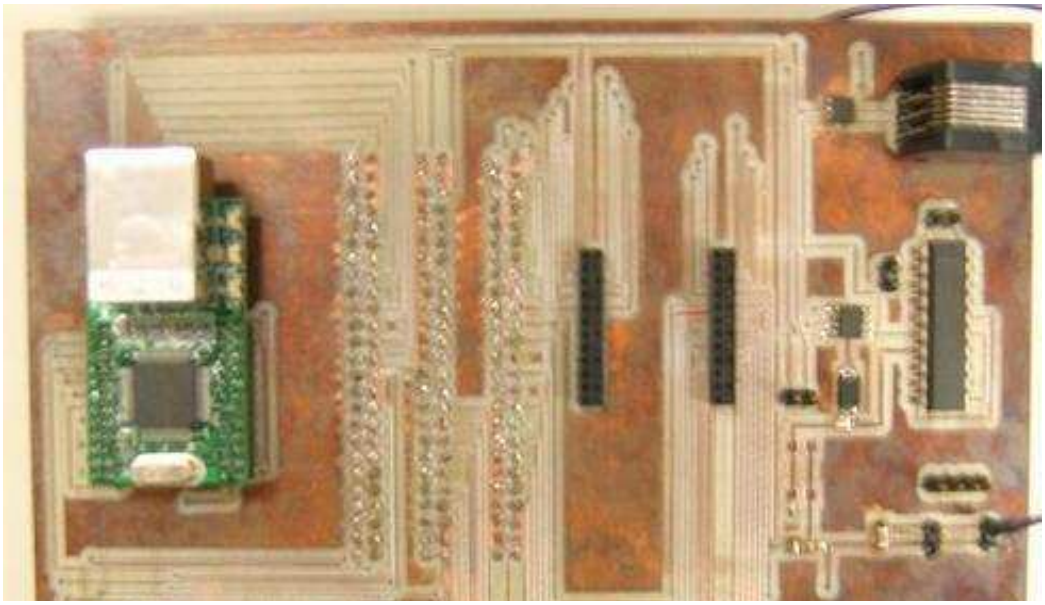

Figure 3.8: Connector board for the DN3000k10 platform

3.5 Summary

This chapter introduced the design of the IIM7010 network controller and provided details of the manner in which it would be implemented. It also introduced the hardware components that are used in creating the network controller and examined some of their important specifications. The hardware components examined include the IIM7010 module that handles packetization at the lower network layers and the SLAAC-1V and DN3000k10 FPGA platforms that house the actual implementation of the network controller.

Chapter 4

Implementation of the IIM7010 Network Controller

The 10 Mbps Ethernet network controller was implemented on the hardware discussed in Chapter 3. The first step in the implementation process was to create timing diagrams to enable the network controller in the FPGA to communicate with the memory of the W3100A. Following this, a state machine was designed for initializing the IIM7010 module. Once this was done, another set of state machines was designed to control the transmission and reception procedures for the TCP and UDP protocols. For the TCP protocol, additional logic was required to perform the connection and disconnection procedures. Due to the fact that multiple channels are used, arbitration logic was required during transmission to take in data from all the channels. A classifier was also required during reception to separate data for the different channels. Figure 4.1 provides a block diagram of the overall system.

In this chapter, Section 4.1 presents the timing diagrams for memory access while the procedure for initializing the IIM7010 is contained in Section 4.2. This section also briefly discusses the makeup of the classifier and arbiter that are used to separate the channel data. Section 4.3 provides details of the controlling state machines for the TCP and UDP protocols. Section 4.4 is focused on the state machines used for connection and disconnection in the TCP protocol. Finally, Section 4.5 examines the implementation of the IIM7010 net-

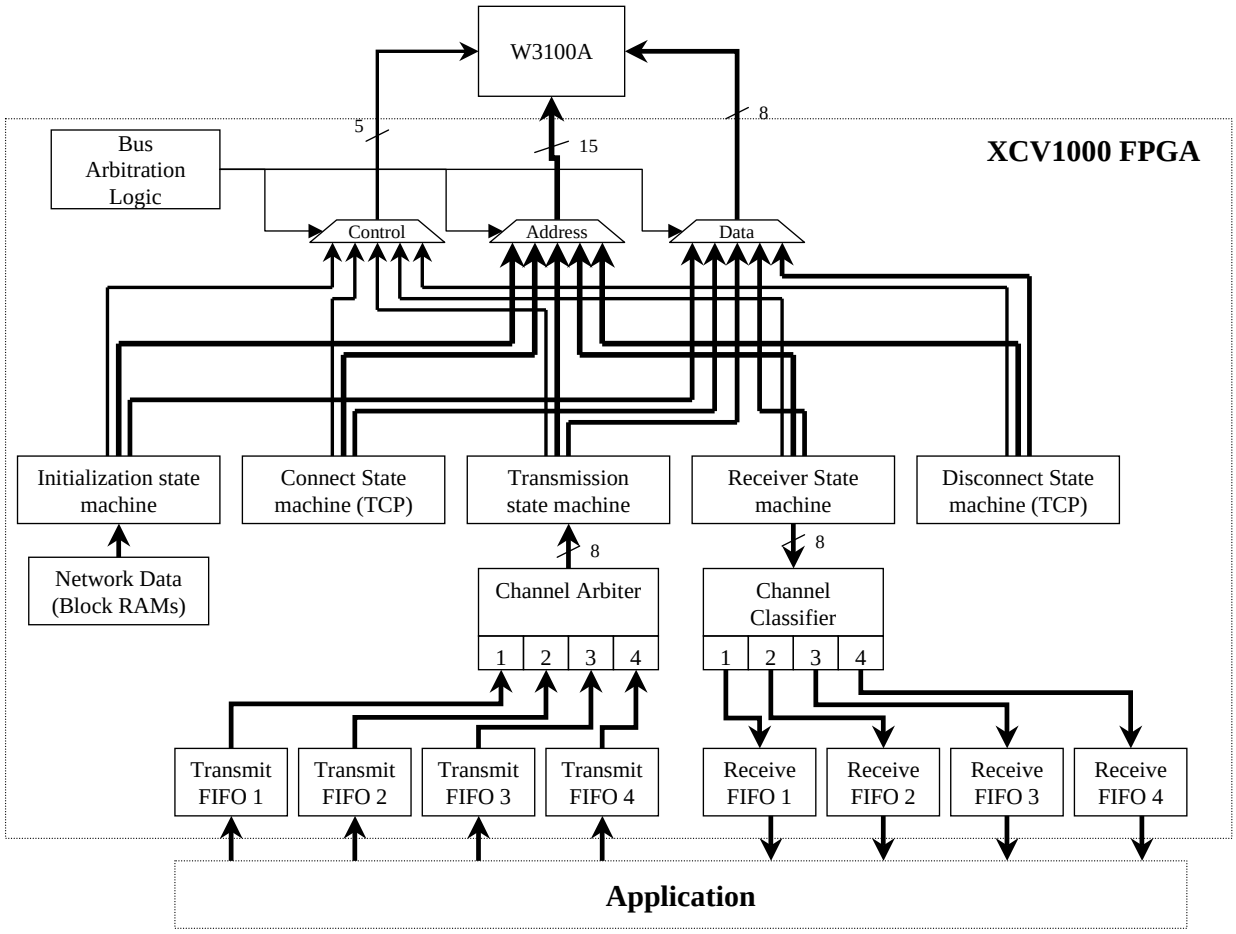

Figure 4.1: Block diagram of Network Controller in FPGA

work controller on a Xilinx Virtex XCV1000 FPGA and presents various implementation characteristics such as resource utilization.

4.1 W3100A Memory Access

To initialize the IIM7010 module, the memory read and write cycles need to be synchronized correctly. The IIM7010 module allows users to access the W3100A in only the clocked mode

with the internal clock running at 25 MHz [16]. As a result of this, there are certain conditions that need to be met to access the memory space of the W3100A. Figure 4.2 shows the timing diagrams of the memory read and write cycles.

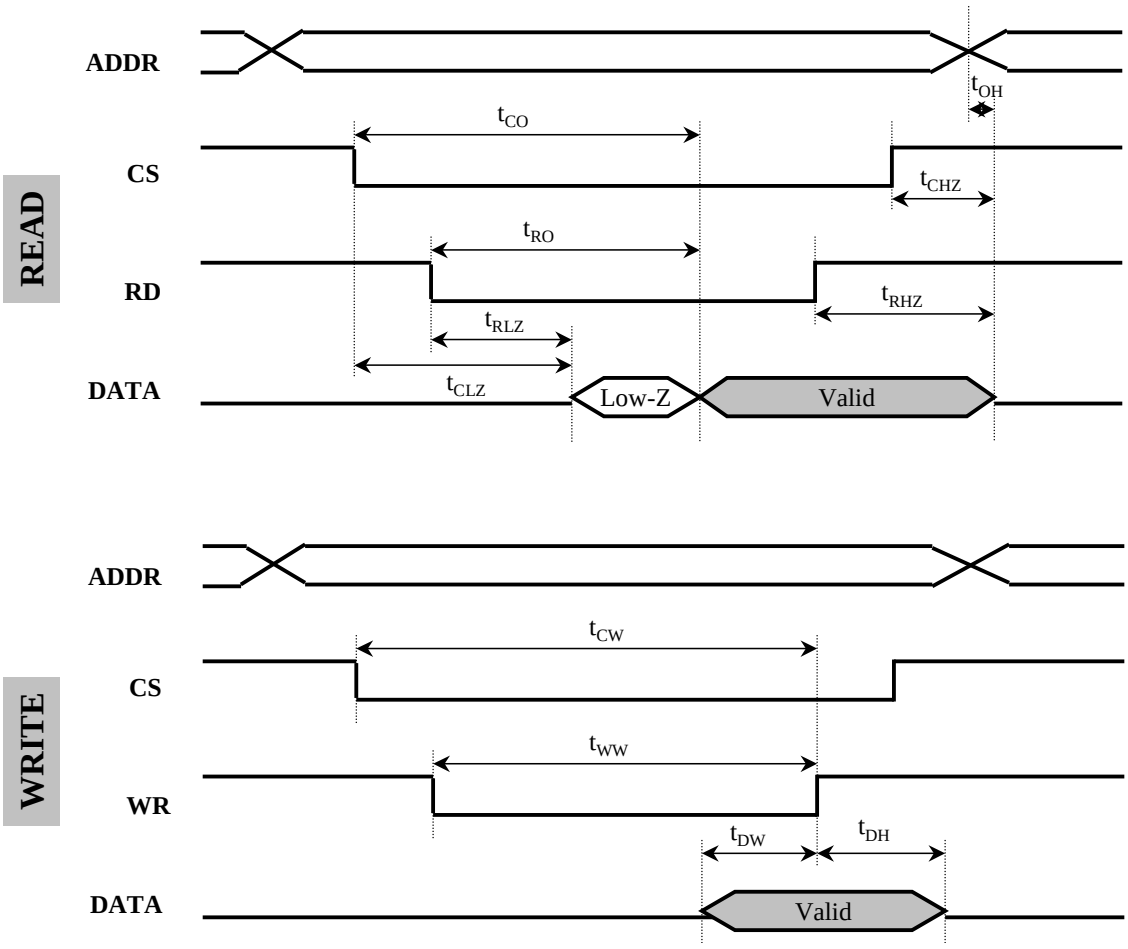

Figure 4.2: Timing diagram of the W3100A memory read and write cycles [17]

Table 4.1 gives the values of the timing constraints that are shown in Figure 4.2.

There are certain other conditions that also need to be met to satisfy the requirements for the memory read and write cycles. The address bus should carry the address of the location being accessed for a time period that is larger than or equal to the period of chip select (CS) signal. The duration of the CS signal should be greater than or equal to 100 ns. The

		Symbol	*Speed*		*Units*
			Min	*Max*	
Read	Chip select enable to data valid	t_{CO}		73	ns
	Read enable to data valid	t_{RO}		73	ns
	Chip select enable to low-Z data	t_{CLZ}	13	54	ns
	Read enable to low-Z data	t_{RLZ}	13	54	ns
	Chip select disable to high-Z output	t_{CHZ}	2	6	ns
	Read disable to high-Z output	t_{RHZ}	2	6	ns
	Data hold from address change	t_{OH}	0		ns
Write	Chip select enable to end of write	t_{CW}	56		ns
	Write pulse width	t_{WW}	56		ns
	Data to write time overlap	t_{DW}	24		ns
	Data hold from write time	t_{DH}	7		ns

Table 4.1: Constraints for the W3100A memory read and write cycles [17]

duration of the read (RD) and write (WR) signals should be smaller than the CS signal. After the CS signal is asserted, the RD and WR signals should be asserted after at least 20 ns. The RD and WR signals should be deasserted at least 20 ns before the CS signals is deasserted [17].

Because of the constraints imposed on the signals, it was decided that the period of the FPGA clock should not exceed 20 ns to avoid excessive delays in memory access; hence the FPGA clock was set to 50 MHz. Based on this, the read and write waveforms for memory access were determined. Figure 4.3 shows the timing diagrams for memory access based on a 50 MHz clock. As can be seen from this figure, the read cycle takes a total of 180 ns while the write cycle takes place in 160 ns. These constraints unfortunately mean that the reads and writes cannot occur on consecutive clock cycles.

4.2 Initialization

Once the memory read/write cycles were calculated and verified as working correctly, the next step was to initialize the IIM7010 module. As mentioned in the previous chapter,

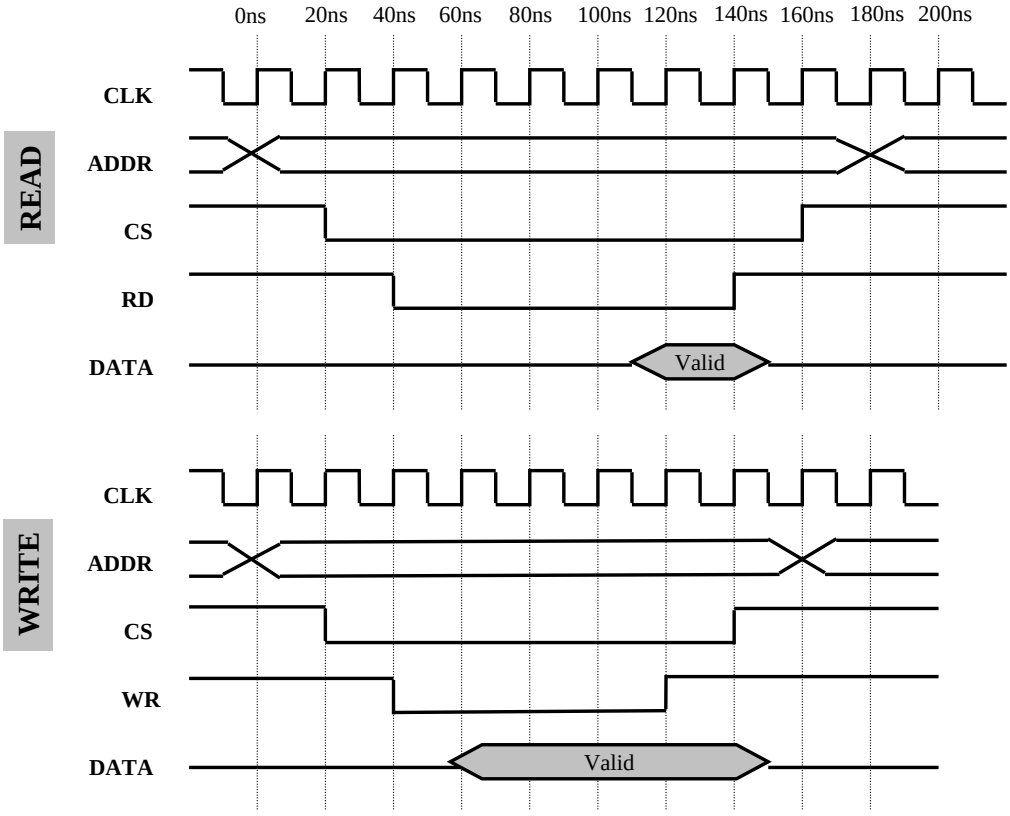

Figure 4.3: Memory access waveforms for 50 MHz clock

the address space of the IIM7010 contains control registers that need to be assigned values before the system can be initialized. First of all, a hardware or MAC address needs to be stored in the Source Hardware Address Register (SHAR). In commercially available network controllers, this address is pre-programmed and is based on the manufacturer. However, the W3100A offers the flexibility of choosing a MAC address since its purpose was to be used in the manufacture of network controllers [17]. To ensure that there is no conflict when the IIM7010 is connected to the network, the MAC address chosen was that of a non-functional NIC manufactured by 3Com. After the MAC address was assigned, the values for various network (IP) layer parameters were filled in. These included the Source IP Address, the Gateway address and the Subnet Mask Address. The Interrupt Mask Register was assigned a value of 0xFF, which enabled all interrupts. This is necessary because these interrupts are

used to inform the FPGA that data has been received on a particular channel.

After all the above-mentioned registers were filled with the correct values, the system was initialized by writing the value 0x01 to the command register (CR) for Channel 0. The initialization was verified by checking the contents of the Interrupt Status Register (ISR) for the same channel. If the value contained therein was 0x01, then the IIM7010 had been initialized. The success of the initialization could also be verified externally by connecting the IIM7010 to a network and trying to contact it using the ICMP 'Ping' command. Since the W3100A itself can respond to ICMP pings, there should be a ping response as long as the system is initialized correctly.

Once the IIM7010 is initialized, the state machines that control the data transfer are activated. But before the data moves from the application to the transmission and reception state machines, it has to pass through an arbiter in the case of transmission and a classifier in case of reception.

4.2.1 Arbiter

Whenever data is placed into any one of the four transmission FIFOs, it is passed to the arbitration entity. This entity ensures that all of the channels are given equal opportunity to transmit their data. It continuously checks if there is data in any of the channels. As soon as data is available in one of the FIFOs, it is sent to the transmission state machine. There is also a two-bit wide bus that conveys information to the state machine regarding the identity of the channel that sent the data. This enables the state machine to choose the network channel that the packet must be sent out on. The arbiter then examines the same channel for any additional data to be sent. The procedure is repeated till a maximum of 2048 bytes (2 KB) have been sent from the same channel. This number was decided upon because if all four channels are operating concurrently, each channel will have a transmission buffer of only 2 KB [17]. At any point, if the FIFO indicates that no more data is available, the arbiter sends a signal to the transmission state machine informing it to send the data that was placed in the buffer. After this the arbiter moves to the next numerically higher

channel and checks for any data to be sent. In this way, it is ensured that the there is an equitable distribution of resources among the four channels.

4.2.2 Classifier

The classifier separates data coming in from the receiver entity into different processing channels depending on which channel in the W3100A received the data. In the current implementation, the classifiers task is trivialized by the fact that the receiver entity can route the data that arrives on a given network channel directly into its corresponding processing channel; however, the classifier entity was created to facilitate the use of more than four processing channels if so required. This is especially true for UDP transmission where more than one remote computer might be transmitting to the same channel. In that case, the classifier would place the packets in their corresponding channels based on the IP address from which they were received.

4.3 Transport Layer Protocols

4.3.1 Transmission Control Protocol (TCP)

TCP is a connection-oriented protocol that guarantees delivery. This means that a packet is resent if the destination does not acknowledge its receipt. Each TCP packet has a unique sequence number associated with it for ordering and retransmission purposes [6]. TCP also has the concept of a port number to identify the sending and receiving process. Each TCP packet has certain flags associated with it that are used to determine the function of the packet. Figure 4.4 shows the format of a TCP packet.

Since the W3100A has built-in hardware to handle functions of the TCP protocol such as arranging the packets sequentially and resending packets, these are not re-created in the FPGA. Before a particular channel is activated in TCP mode, there are a number of TCP

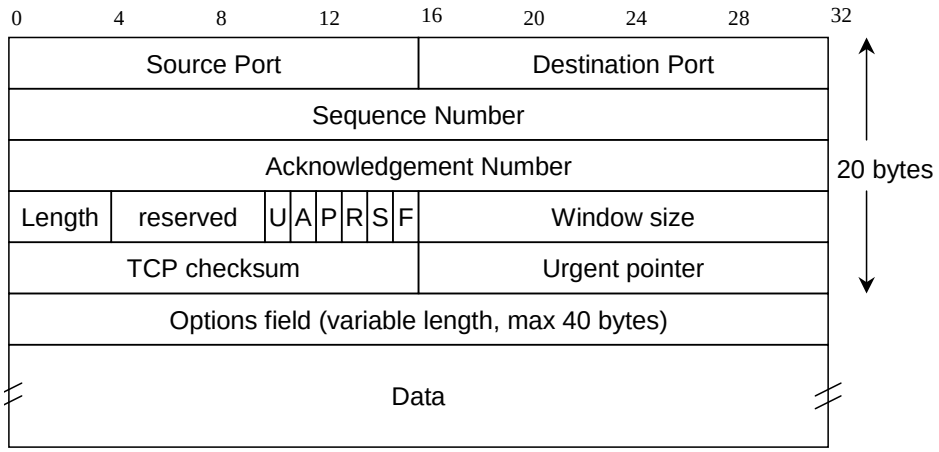

Figure 4.4: TCP packet format [8]

related parameters that need to be assigned values in the W3100A channel registers. First of all, the source port number is given. For starting a connection, there are two different modes available in TCP, which are discussed in detail in Section 4.4. In one of these modes, the IIM7010 initiates a connection while in the other one it waits for a connection. If the connection is initiated by the IIM7010, the network controller needs to be given the IP address of the destination computer and the port on which the connection is to be made. Following this, other parameters that affect network traffic such as the Type of Service (TOS) and Maximum Segment Size (MSS) for packets are assigned values. Finally, the data pointers in the transmit and receive buffers are initialized to the default (0x00) location. After a TCP connection is established with a particular computer, transmission and reception state machines are activated. Figure 4.5 shows the state machine for the TCP transmission process.

The first step in this process is finding the location of the transmission pointers. As mentioned in Chapter 3, there are two pointers associated with the TCP transmission process. These are the 'Transmission Write' pointer, which is manipulated by the state machine in the FPGA and the 'Transmission Acknowledge' pointer, which is manipulated by the W3100A itself. Once the location of these two pointers is read, the difference between them is calculated to find the amount of free buffer space. It should be noted that the transmission and

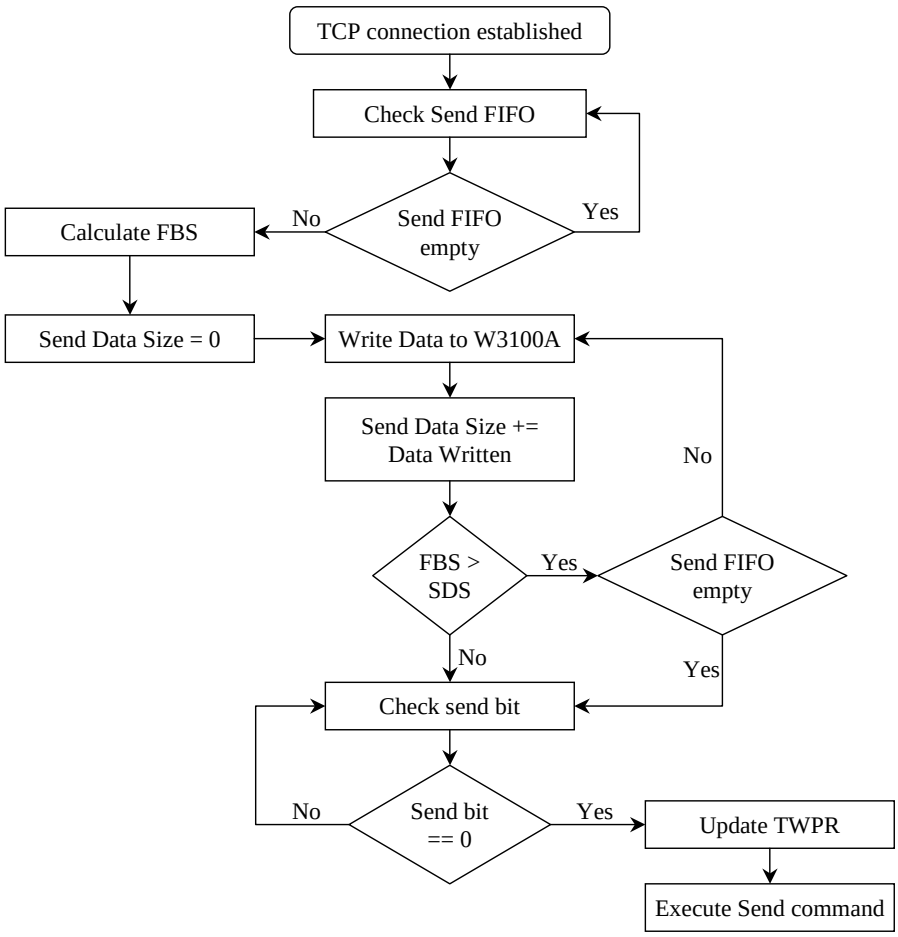

Figure 4.5: TCP data transmission process in the network controller [17]

reception buffers are circular. This is taken into account when the value of free buffer space is calculated using the following formula:

$$
\begin{aligned}
&If(TWPR > TAPR)then \\
&\qquad FBS = BufferSize - (TWPR - TAPR); \\
&Elsif(TWPR < TAPR)then \\
&\qquad FBS = TAPR - TWPR; \\
&Else \\
&\qquad FBS = BufferSize;
\end{aligned}
$$

where FBS is Free Buffer Space, TWPR is the position of the Transmission Write pointer and TAPR is the position of the Transmission Acknowledge pointer. Data is then written till either the transmission FIFOs are empty or there is no more FBS. After this is done, the 'send' bit in the Interrupt Status Register for the channel on which the transmission is occurring is checked to see whether the previous 'send' command has been completed. If it hasn't, the state machine is put into a 'wait' mode till the command is completed. After the send bit is deasserted, the TWPR pointer is updated to point to the top of the data that has been written to the transmit buffer. The send bit in the command register is then asserted to complete the transmission sequence.

The TCP data reception process with the W3100A is somewhat similar to the transmission process. Figure 4.6 shows the state machine for the TCP data reception process in the FPGA.

In the W3100A, whenever data is received from the peer, it is recorded in the receive buffer in the space between the 'Receive Read' pointer and the 'Receive Write' pointer. The Receive Write pointer is controlled internally by the W3100A while the Receive Read pointer is manipulated by the controller in the FPGA. When data is received on any of the channels, there are two ways in which the FPGA can be informed about this. The first method is to keep checking the location of the receive pointers for each of the four channels and see if there is a change in the position of any of the pointers. However, this process causes a significant delay because there a total of forty registers that need to be accessed to get the positions of pointers for all the channels. The second method and the one currently being used is to rely on interrupts. Whenever the state machine receives an interrupt, it checks the Interrupt Status Register to find which one of the channels caused the interrupt. If the interrupt is for a data receipt, the state machine examines at the values of the RWPR pointer and the RRPR pointer. The size of data received is calculated using the following formula:

$$
\begin{aligned}
&If(RWPR > RRPR)then \\
&\qquad RDS = RWPR - RRPR; \\
&Else \\
&\qquad RDS = BufferSize - (RRPR - RWPR);
\end{aligned}
$$

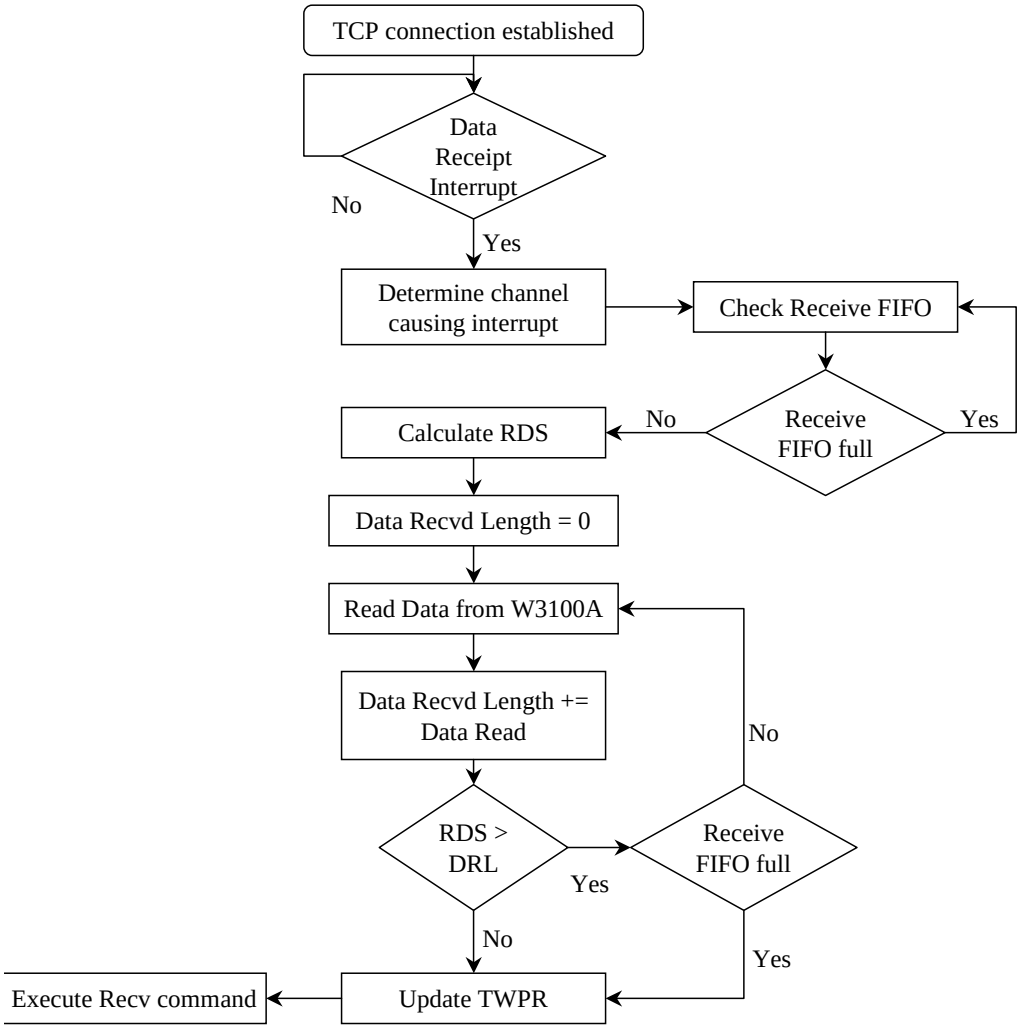

Figure 4.6: TCP data reception process in the network controller [17]

where RDS is the Received Data Size, RWPR is the Receive Write pointer and RRPR is the Receive Read pointer. Data is then read from the receive buffer till the location pointed to by the RWPR pointer is reached. When this happens, the RRPR pointer is increased to the value initially pointed to by RWPR. It should be noted that in the mean time, the value of RWPR might have changed because of more data being received; however, the new value for RWPR is not taken into account since the additional data has not yet been read by the

network controller. Finally the receive command is executed by asserting the 'receive' bit in the command register of that particular channel.

4.3.2 User Datagram Protocol (UDP)

UDP is a connection-less protocol that does not provide reliability. Like TCP, UDP also has the concept of a port number to identify the sending and receiving process. Unlike TCP, however, there is no notion of sequence or order of the packets [6]. Figure 4.7 shows the format of a UDP packet.

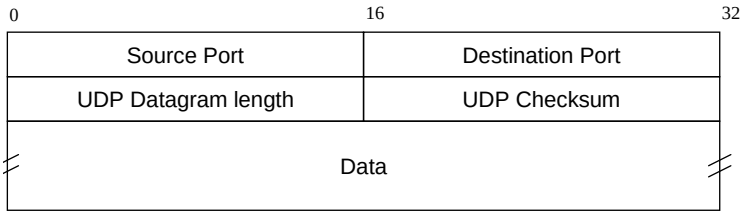

Figure 4.7: UDP packet format [8]

The state machines required for UDP data reception and transmission with the W3100A are very similar to the ones for TCP; however, there are some important differences. In the transmission process for TCP, the pointers that are used to determine the location of data in the transmission buffer are the 'Transmission Write' pointer (TWPR) and the 'Transmission Acknowledge' pointer (TAPR). For UDP transmission, on the other hand, the 'Transmission Read' pointer (TRPR) is used instead of TAPR [17]. This pointer is manipulated by the W3100A and its value informs the FPGA-based controller about the amount of data that has been transmitted.

In the UDP reception process, the state machine works exactly the same as the TCP reception state machine. However, the data that is read by the controller is still in a packetized form. Figure 4.8 shows the structure of the data obtained by the FPGA from the W3100A receive buffer.

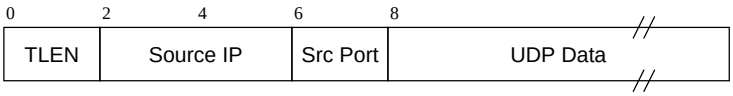

Figure 4.8: Received UDP packet structure [17]

The header for the received packet consists of the following parameters:

- The two-byte 'Length' field gives the length of the header and the UDP data combined.
- The four-byte 'Source IP' field, which gives the IP address of the network interface that sent the packet. This field is used by the classifier entity discussed earlier to put data into the proper processing channels.
- The two-byte 'Source Port' field, which gives the source port of the process that sent the packet. This can be used to differentiate between different processes transmitting data from the same IP address.

4.3.3 Manipulation of Network Parameters

There are several parameters that can be manipulated in the W3100A state machines to change the behavior of the network controller. One of them is the maximum size of packets (MSS). This influences both, the TCP and UDP protocols. This value is set using the Maximum Segment Size Register (MSSR) register for each channel. The value of MSS can have a significant impact on the functioning of the network controller as can be seen in the results obtained in Chapter 6. Another parameter that affects both protocols is the Type of Service (TOS) of the IP protocol. This field is normally left as 0x0 for normal service but can be changed to values 0x1, 0x2 and 0x4 and 0x8 to maximize reliability or throughput or to minimize delay [26]. This IP field can be manipulated using the TOSR register for the given channel. The rest of the parameters that can be manipulated affect only the TCP protocol. They are:

- *Initial Retry Time-value*: This is the time after which, retransmission of packets occurs when using TCP. It can be set to values of 100 ms, 200 ms or 400 ms by adjusting the value of the Initial Retry Time-value (IRTR) register.

- *Retry Count*: This assigns the number of retries in the case of retransmission before timeout occurs in TCP. The value is stored in the Retry Count Register (RTR) in the W3100A. Along with the Initial Retry Time-value, this parameter is determines the overall timeout value for TCP according to the following formula [17]:
$TimeoutValue = InitialRetryTimeValue * (2^{RetryCount} - 1)$

- *Silly Windows Syndrome*: This is a condition in which, small amounts of data are unnecessarily exchanged using full-size segments, thereby increasing network traffic [26]. This condition generally needs to be avoided and this can be done by placing an appropriate value in the Socket Option Protocol Register (SOPR) for a given channel.

- *No Delayed ACK*: Delayed ACKs are used in TCP as a means of reducing the number of packets on the network. When this option is used, an ACK is only sent for alternate segments or when no segment arrives for more than 200 ms [26]. This option is also set using the SOPR for each channel.

4.4 TCP Connection and Disconnection

As mentioned earlier, TCP is a connection-oriented protocol. Before data can be exchanged between two peers over TCP, a connection needs to be established between them. If the IIM7010 is given the IP address and port number of the peer, it can initiate the connection. This process is termed 'Active Open'. If the IIM7010 waits for a connection from the peer, the process is called 'Passive Open'. Similar terminology is applied to the disconnection process based on whether or nor it is initiated by the IIM7010 controller. The connection and disconnection process involves a three-way handshake [26]. This handshake is implemented with the help of the TCP flags 'SYN', 'FIN', 'RST' and 'ACK'.

4.4.1 Connection

Figure 4.9 illustrates the state machines for the Active Open and Passive Open processes.

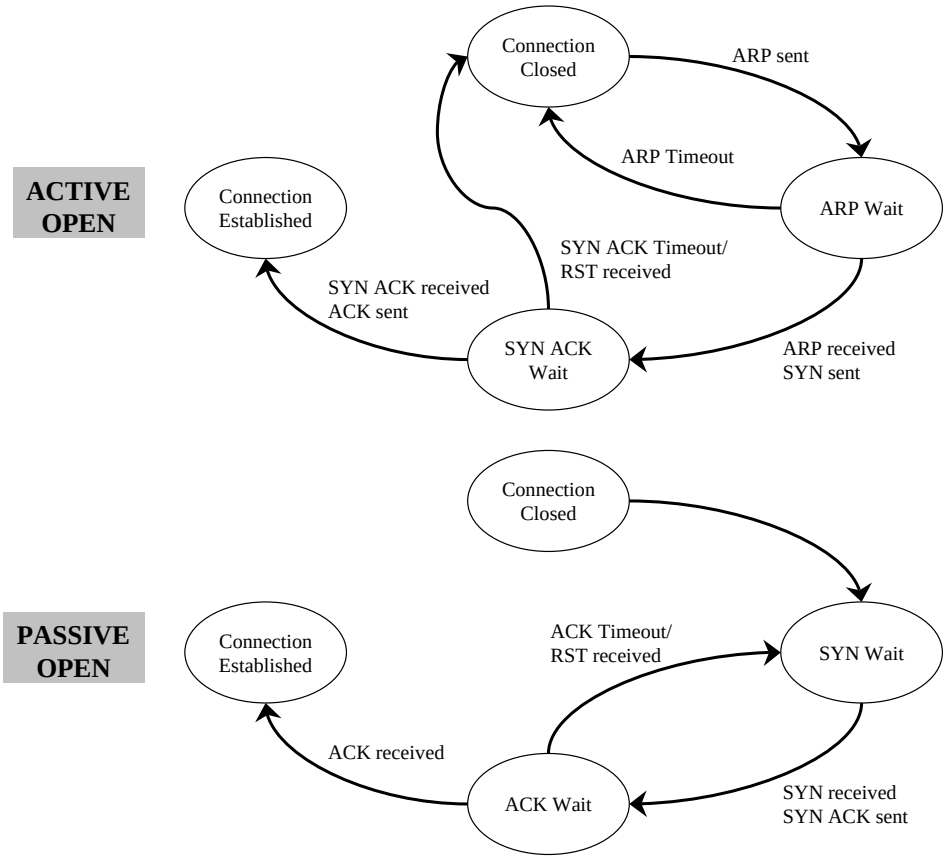

Figure 4.9: TCP connection state machines [17]

After a particular channel is initialized in TCP mode, it defaults to the 'Connection Closed' state. In the Active Open process, the first step in making the TCP connection is sending an ARP (Address Resolution Protocol) packet. This is a broadcast packet which tries to get the physical (MAC) address of the network interface that holds the IP address of the peer. The network controller sends an ARP request packet and goes into the 'ARP Wait' state. Whenever the ARP reply packet from the peer is received, the IIM7010 sends out a SYN packet and the controller moves to the 'SYN ACK Wait' state. However a timeout value has

been implemented for the ARP Wait state so that it returns to the Connection Closed state if an ARP reply is not obtained within a certain period of time. In the SYN ACK Wait state, the network controller waits for a specified time interval to receive a SYN ACK packet from the peer. If the timeout expires or if a RST packet is received, indicating that there is no process listening on the requested port on the peer, it goes back to the Connection Closed state. If a SYN ACK packet is received from the peer, the IIM7010 sends out an ACK packet and the controller switches to the 'Connection Established' state.

In case of the Passive Open process, the peer has the task of initiating the connection by sending a SYN packet. The network controller goes into the 'SYN Wait' state following initialization and stays there till a SYN packet is received. After this, the IIM7010 sends out a SYN ACK packet and the controller moves to the 'ACK Wait' state. If an ACK packet is received from the peer completing the three-way handshake before the timeout expires, it goes to the 'Connection Established' state. If the timeout does expire, the network controller goes back to the SYN Wait state and waits for the peer to re-initiate the connection. Once the connection is established, the IIM7010 controller can activate the transmission and reception state machines.

4.4.2 Disconnection

Figure 4.10 shows the state machines for the 'Active close' and 'Passive close' processes.

If a particular TCP channel no longer wants to exchange data with its peer, or if it wants to start communicating with another peer, the TCP connection needs to be closed before another one can be established. This is done by means of another three-way handshake. In the Active Close process, the IIM7010 controller initiates the disconnection by sending a FIN packet to the peer. The controller then moves to the 'FIN ACK Wait' state. In this state, the IIM7010 can either receive a FIN ACK or an ACK packet from the peer. If a FIN ACK packet is received, the disconnection is complete and the controller moves to the 'Connection Closed' state for that channel. Similarly if no response is received within a specified time period, the connection is ended and the IIM7010 controller goes to the Connection Closed

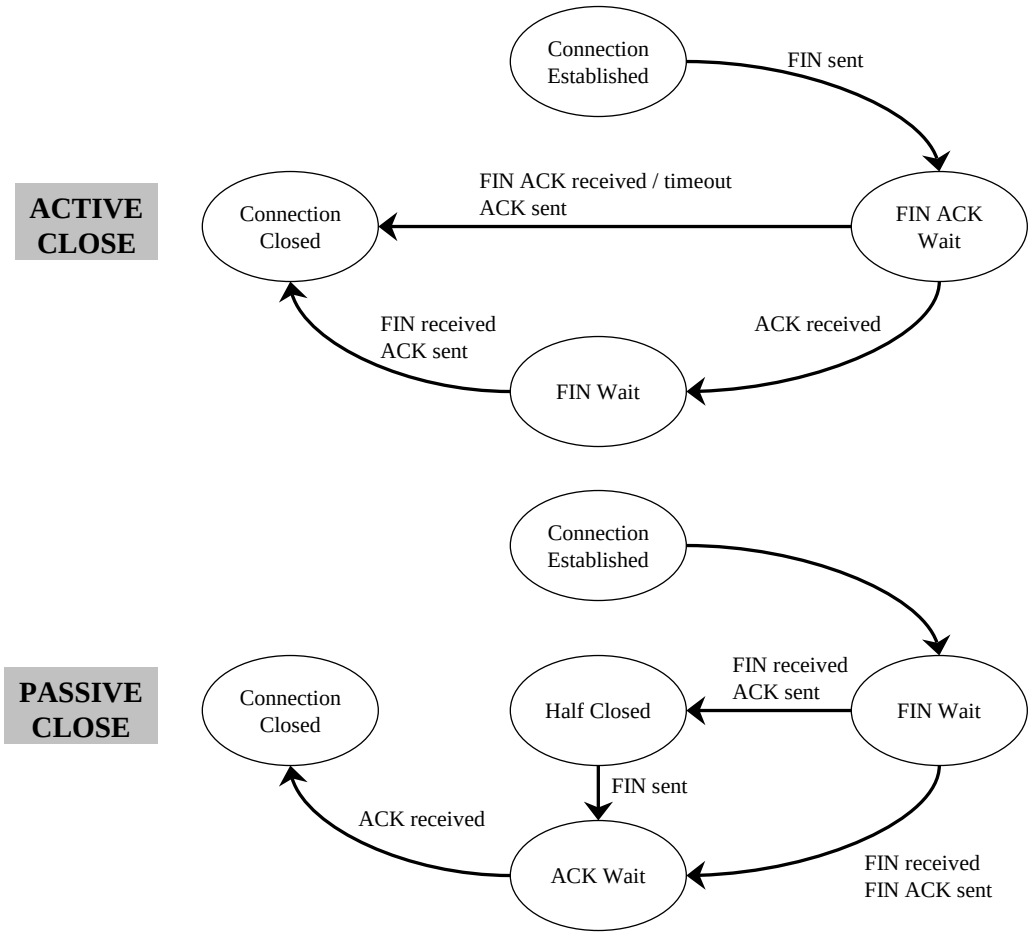

Figure 4.10: TCP disconnection state machines [17]

state. However, if the peer still has some data to send, it will only send an ACK packet acknowledging that no more data will be sent from the IIM7010's side. In this case the state machine moves to the 'FIN Wait' state. It should be noted that in this state, the receiver state machine is still functioning to receive data. Only the transmission state machine is stopped. When the peer wants to stop sending data, it will send a FIN packet to which, the IIM7010 responds with an ACK packet and closes the connection.

In the Passive Close process, the state machine goes to the 'FIN Wait' state and waits for a FIN packet from the peer. Once this packet is received, the network controller sends either

an ACK packet or a FIN ACK packet depending on whether or not it has more data to send. If it wants to continue to transmit data, it sends an ACK packet and moves to the 'Half Closed' state. In this state the transmission state machine can still continue to send data but the receiver state machine is stopped. Whenever the controller wants to stop transmitting data, it sends a FIN packet to the peer and moves to the 'ACK Wait' state. If it wants to stop transmitting data at the time the FIN packet from the peer is received, it transmits a FIN ACK packet and moves directly to the ACK Wait state. In this state, the IIM7010 waits for an ACK from the peer and moves to the 'Connection Closed' state whenever it does receive the ACK. It should be noted that in both the Active and Passive Close processes, the receipt of an RST packet from the peer moves the controller directly to the Connection Closed state. In this manner, the IIM7010 network controller handles the TCP connection and disconnection processes.

4.5 FPGA Implementation

The state machines discussed in sections 4.2, 4.3, and 4.4 were implemented on a Xilinx Virtex XCV1000 FPGA to obtain the IIM7010 network controller. The details of this implementation are discussed in this section.

The IIM7010 network controller occupies a total of 1082 LUTs on the XCV1000 FPGA. This is equivalent to only 4% of the 27,648 LUTs available in that FPGA. A total of five block RAMs out of the 32 available block RAMs are used up. It should be noted that out of these, only one block RAM is required by the network controller itself for storing network related initialization data. The other four block RAMs used in the current implementation are used for creating the FIFO interfaces to the application. This shows that the network controller in the FPGA does not take up significant amount of room in that logic device. Additionally, it uses only 29 I/Os out of a maximum of 512 in a XCV1000 FPGA. As such, it is possible to have multiple network controllers for different applications residing in the same FPGA. As mentioned at the start of this chapter, the IIM7010 controller has been currently designed for running at 50 MHz. The design tools used for implementation (Synplicity [27] and Xilinx

tools) estimated that the minimum period for the design was 17.224 ns, which gave a slack of 2.776 ns. This shows that the controller is capable of running at higher speeds if required.

Design Issues: One significant design change that had to be made during the implementation of the IIM7010 network controller was the addition of pull-down resistors on the data bus going from the FPGA to the IIM7010. This was required because it was observed that the internal logic in the W3100A was driving a weak zero signal. The observation came about from errors observed during reception of supposedly error-free TCP data. Based on recommendations on the manufacturers web site [28], pull-down resistors in the Virtex IOBs that were part of the data bus were activated. This eliminated the problems in the data reception.

4.6 Summary

This chapter provided an in-depth explanation of the procedure followed for developing the network controller on an FPGA. It presented the various steps taken for creating the controller such as designing state machines for controlling network data transfer using the TCP and UDP protocols. Finally, it showed the results of implementing these steps on a Xilinx Virtex XCV1000 FPGA to obtain the IIM7010 network controller.

Chapter 5

Partial Reconfiguration

Chapter 2 discussed the advantages of making the network controller partially reconfigurable and also gave a brief overview of the actual process. This chapter details the process of creating partial bitstreams for the IIM7010 network controller. Section 5.1 examines some applications where partial reconfiguration would benefit a network controller. To provide a better understanding of the flow used for partial reconfiguration, the creation of the partial bitstreams for one of the applications is examined in detail in sections 5.2, 5.3 and 5.4. Section 5.2 focuses on the creation of logic entities that conforms to the partial reconfiguration guidelines. The creation of the floor plan and constraints for the logic in the static and reconfigurable modules is explained in Section 5.3. Finally Section 5.4 presents an overview of the method used for actually creating the partial and complete bitstreams from the modules designed earlier.

5.1 Applications for Partial Reconfiguration in Network Controllers

The need for incorporating the feature of partial reconfiguration into a network controller arises from the presence of certain variables associated with the controller that could be

subject to change at run time. In the case of the IIM7010 network controller three such variables have been identified. They are:

1. Number of operational channels
2. Protocol used for a particular channel
3. Parameters associated with the network protocol in use

Based on these three variables, different scenarios were decided upon for demonstrating partial reconfiguration in the IIM7010 network controller.

5.1.1 Activating Additional Channels

Consider a situation in which, the IIM7010 network controller is communicating with different machines using some of the available channels. If additional channels now need to be activated, the whole FPGA would have to be reconfigured. As a result of this, the channels that were previously operational would have to cease data transfer while the FPGA is being reconfigured. This may not be acceptable in certain scenarios, especially if one of the original channels was using the TCP protocol. This is because if the network controller stops acknowledging packets sent by the peer, the peer is likely to have a timeout and close the connection. In this case, the connection would have to be re-established after the FPGA is reprogrammed and this may not necessarily be feasible for some applications. With partial reconfiguration, however, the original channels can be placed in a static module that is not affected when the new channel is activated. Hence, in general, partially reconfigurable network controllers could be useful for applications that require preservation of the state of the network channels.

5.1.2 Changing Channel Protocols

Another application that could benefit from partial reconfiguration is one in which, a network controller needs to change the communication protocol on one of its channels at runtime.

Consider a network controller design with a channel that is activated using a certain protocol, say TCP. If at a later time, the same channel needs to communicate with another machine using the UDP protocol, a partial bitstream can be created where the channel is setup using UDP. As in the previous application, the benefit of partial reconfiguration is two-fold. For one, the channel can be reconfigured without affecting any of the other channels that may be operating in the static part of the design. Secondly, it reduces the time required for reconfiguration since the partial bitstream can be significantly smaller than a bitstream for the entire device. So, in general terms, a network controller that can use a partial bitstream for faster reconfiguration would be useful for applications that are intolerant to the otherwise long full-reconfiguration process. Figure 5.1 shows a block diagram of the network controller design for this application. This is the design whose implementation is examined in detail in sections 5.2, 5.3 and 5.4.

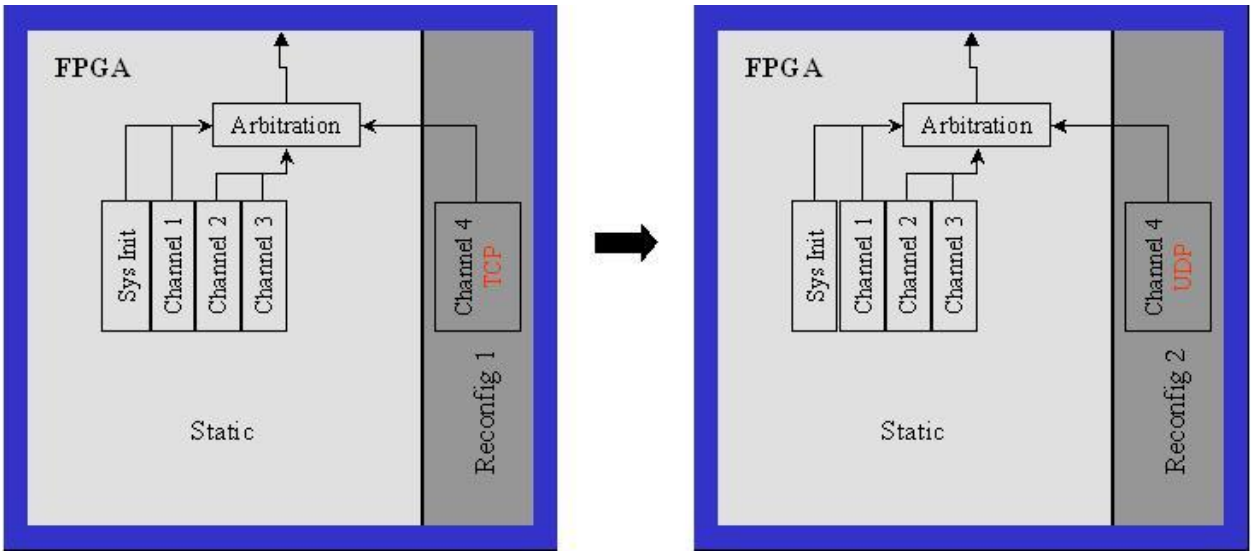

Figure 5.1: Changing the protocol for a channel using partial reconfiguration

5.1.3 Changing Network Protocol-related Parameters

Based on experimental analysis of the performance of the IIM7010 network controller, it was seen that certain protocol parameters such as the maximum segment size were having a significant effect on the throughput achieved by the network controller and packet loss in the case of UDP. These results are shown in Chapter 6. Based on this, it was decided that an application for the partially reconfigurable network controller could be one that required the ability to change network related parameters at run-time. To demonstrate this, a design was created for the IIM7010 controller that had two channels which used the UDP protocol. Both these channels had a certain pre-programmed value of maximum segment size for the UDP packets. A partial bitstream was then created in which one of the channels was reconfigured to have a larger segment size. Figure 5.2 shows a block diagram of the network controller design for this application.

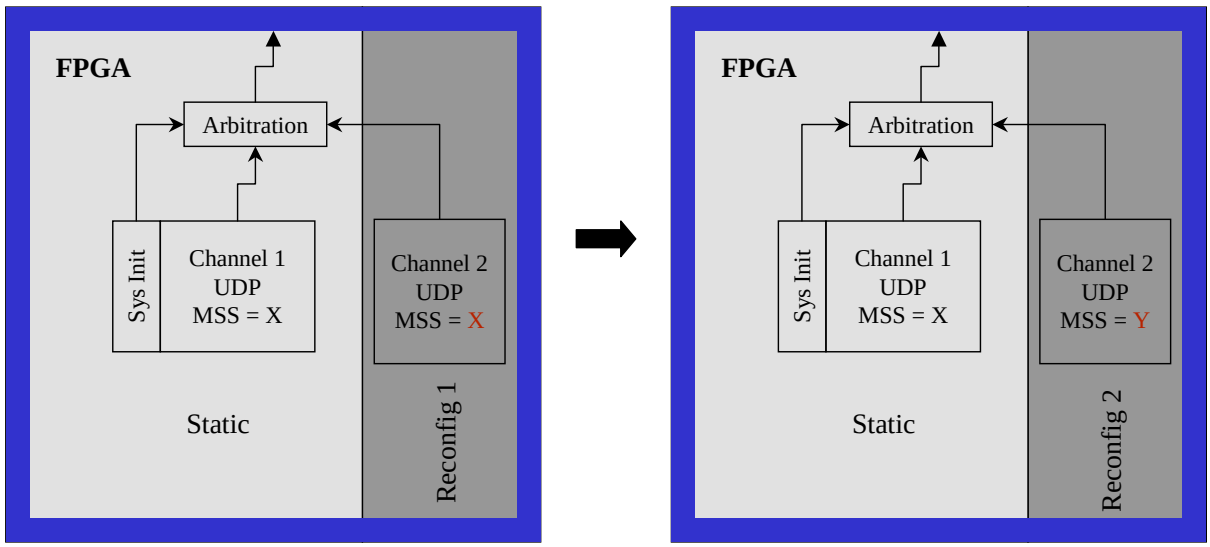

Figure 5.2: Changing the segment size for a channel's protocol using partial reconfiguration

The results of the partial reconfiguration on the network performance are detailed in Chapter 6. The next section examines the creation of the design discussed in Section 5.1.2.

5.2 Design Creation

As mentioned in Chapter 2, there are two main flows for partial reconfiguration: Module-based and Difference-based. In module-based partial reconfiguration, or 'Modular Flow', distinct portions of the FPGA are defined to be reconfigured while the rest of the device remains in active operation [14]. These portions are referred to as 'Reconfigurable Modules'. These modules can either be completely independent of each other or they can communicate across the module boundaries using special 'Bus Macros'. Partial bitstreams are generated for each of the reconfigurable modules. In difference-based partial reconfiguration or 'Difference Flow', a small change is made to the design using a software such as FPGA Editor from Xilinx. A bitstream is then generated based on the difference between the two designs. In both, modular flow and difference flow, the partial bitstream generated is smaller than the bitstream for the entire device; however, difference flow is feasible only when a small change needs to be made to the design. Since the design changes required for creating the partial bitstreams for the IIM7010 network controller are substantial in size, the modular flow was used in this case.

Reconfigurable modules used in the modular flow need to have certain properties. These properties are as follows:

- *Boundaries of a module*: The height of the module is always the full height of the device. Its width ranges from a minimum of four slices to a maximum of the full device width, in four-slice increments. Hence the horizontal boundaries of the modules should always be on slices that are multiples of four [0, 4, 8 ...] [14].

- *Resources of a module*: All logic resources encompassed by the width of the module are considered part of the reconfigurable module's bitstream 'frame' [14]. This includes routing resources, CLBs, IOBs, multipliers and block RAMs.

- *IOB allocation*: IOBs immediately above the top edge and below the bottom edge of a reconfigurable module are part of that specific module's resources. If a reconfigurable module occupies either the leftmost or rightmost slice column, all IOBs on the the

corresponding edge are part of that module's resources [14].

- *Design of reconfigurable modules*: The modules for creating the partial bitstreams must be designed so that the static portions of the FPGA do not rely on the state of the module under reconfiguration while reconfiguration is taking place. The implementation should ensure proper operation of the design during the reconfiguration process.

- *Communication between modules*: A reconfigurable module's boundary cannot be changed. The position and region occupied by any single reconfigurable module is always fixed. Reconfigurable modules communicate with other modules, both fixed and reconfigurable, by using a special bus macro [14].

- *Clocking resources*: Clocking logic is always separate from the reconfigurable module.

Bus Macros: Bus macros are used to allow communication across the boundaries of reconfigurable modules. For placing the bus macros in the Virtex-II FPGAs, a '.nmc' from Xilinx was used that hard placed a bus macro in the design. The bus macro forms a fixed routing bridge between the static and reconfigurable modules. It is pre-routed to specify the exact routing and hence it can be used at the boundary of the modules. The current implementation of the bus macro uses a total of eight tri-state buffers to allow four bits per row to travel either from left to right or right to left. One bit uses a single tri-state buffer longline [14]. Figure 5.3 shows the block diagram of a bus macro in the Virtex-II architecture.

The tri-state buffers need to be tied either high or low depending on the direction that a particular signal is traveling in. This is done by placing either a 'zero' or a 'one' in a Look-Up Table (LUT) near the bus macro and then tying the output of the LUT to the appropriate signal in the bus macro. Figure 5.4 illustrates the manner in which this is done. In this figure, outputs 1 and 3 of the bus macro are going from right to left while outputs 2 and 4 are going from left to right. When a signal goes from one side to another, the tri-state buffer on the source side has to be enabled (tied to ground) while the tri-state buffer on the received side has to be disabled (tied to VCC).

To create the partial bitstreams for the IIM7010 controller that were described in Sec-

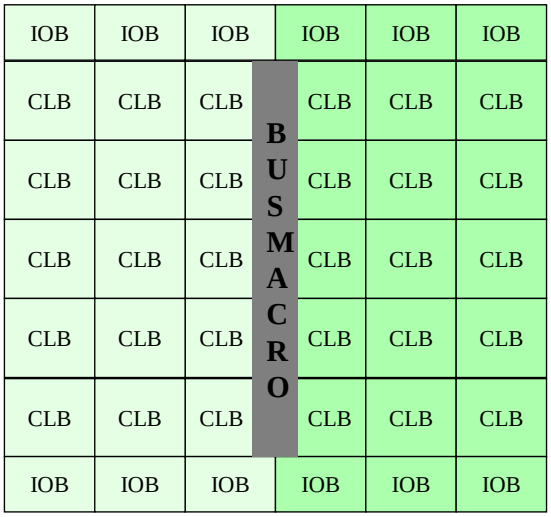

Figure 5.3: Block diagram of a bus macro in a Virtex-II architecture [14]

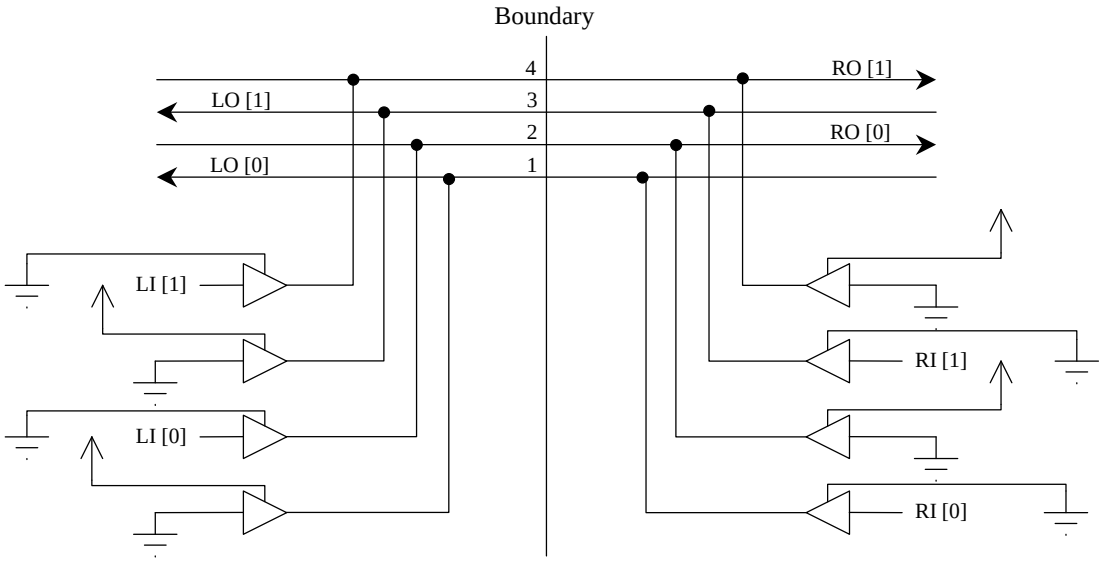

Figure 5.4: Bus macro routing [14]

tion 5.1.2, a complete design was initially created in which, three channels were activated using UDP and one was activated using TCP. The functionality of this design was checked

and it was verified that all channels were working correctly. Then a similar design was created in which, all four channels used the UDP protocol. This design was also checked to make sure that all channels were functioning correctly. The nature of the initial design was such that each channel was created using a separate entity. Because of this the next step, that involved grouping the three UDP channels into one entity and the lone TCP channel into another entity, was straightforward.

One problem that had to be remedied during the design stage was the location of the I/O signals required for communicating with the IIM7010 module. As shown in Chapter 4, there is only one set of data, address and control signals for the IIM7010 that all channels have to share. The problem was that all the I/O signals on the XC2V4000 FPGA that were designated for communication with the IIM7010 were along the top edge of the FPGA. The location of these signals could not be changed because the I/O pins were connected to the IIM7010 by means of the 'daughter' card described in Chapter 3 whose routing was fixed. This caused a problem when dividing the FPGA into static and reconfigurable parts because any reconfigurable portion at the edge of the chip would include a few of the I/O pins required by the static part. Ultimately this problem was resolved by having an interface between the static and reconfigurable portions that mapped the signals from the static module to their corresponding I/O pins through the reconfigurable module. This is illustrated in Figure 5.5.

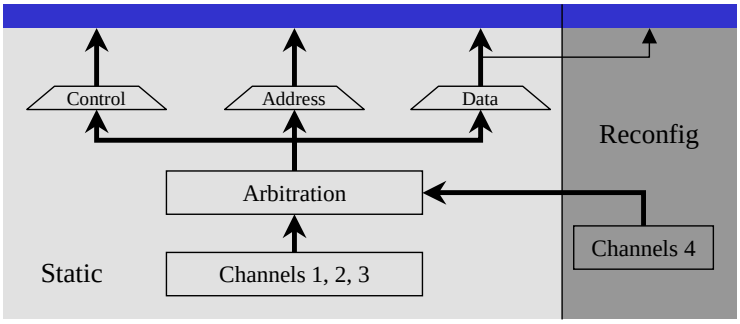

Figure 5.5: I/O pin access through reconfigurable module

Once this pattern was decided upon, the interconnections between the modules were created accordingly. After this, the designs were once again tested as a whole to ensure that the

separation of the entities had not created any problems with the functioning of the controller. The next step in the modular flow was to incorporate the bus macros, that have been discussed previously. Along with the bus macros, two additional entities were added to the design to support partial reconfiguration. These were a Digital Clock Manager (DCM) and a buffer for clocking resources. These resources are used to de-skew the clock that goes to both the static and reconfigurable parts.

In the Xilinx Application Note 290 [14], the modular flow is broken down into three main phases that are explained over the course of the next two sections. They are:

1. *Initial Budgeting Phase*, in which a floor plan and constraints are created for the overall design,
2. *Active Module Phase*, that involves implementing modules through the place and route process, and
3. *Final Assembly Phase*, in which individual modules are assembled together into a complete design.

5.3 Initial Budgeting

The floor planning of the entire design takes place in the Initial Budgeting phase. The output of this phase is a User Constraint File (.ucf) that contains all placement constraints for the design. All the top-level logic for static and reconfigurable modules is assigned fixed location constraints in this file [29]. The UCF file also has constraints for bus macros and other components such as DCMs, LUTs and buffers. The Floorplanner tool [30] was used to create the UCF file for the network controller. The aim of partial reconfiguration is to make the reconfigurable component as small as possible so that less time is required for changing the design and most resources on the FPGA can function unaffected at the time of reconfiguration. Hence it was decided that most of the FPGA will remain static and only a small number of slices will be reconfigured. But in the IIM7010 network controller design, the reconfigurable module needs one block RAM to store data for re-initializing the channel. Hence it was necessary to place the reconfigurable module in such manner that it would

have access to at least one column of block RAMs. To gain a better understanding of the placement of the static and reconfigurable modules and other components, it is necessary to examine the architecture of a Xilinx Virtex-II XC2V4000 FPGA.

5.3.1 Location of Resources in XC2V4000 FPGA

The XC2V4000 FPGA contains 5760 CLBs arranged in a 80 x 72 array of rows and columns [24]. Each CLB consists of four slices. As such, this FPGA has a total of 23,040 slices in a 160 x 144 array. Considering a Cartesian coordinate system for slices, the slices in the FPGA have coordinates ranging from [X0, Y0] (the slice at the bottom left) to [X143, Y159] (the slice at the top right). This was verified using the Xilinx FPGA Editor software. Each CLB in a Virtex-II FPGA has two tri-state buffers [24], which means that the XC2V4000 FPGA has 11,520 tri-state buffers. These buffers follow a similar coordinate system to that of the slices. In general, a CLB that contains slices with coordinates [X, Y], [X+1, Y], [X, Y+1] and [X+1, Y+1] has tri-state buffers with coordinates [X, Y], [X, Y+1]. For instance, consider the CLB that contains slices with coordinates [X108, Y44], [X108, Y45], [X109, Y44] and [X109, Y45]. The tri-state buffers in this CLB have coordinates [X108, Y44] and [X108, Y45]. As such, tri-state buffers in a XC2V4000 FPGA have coordinates ranging from [X0, Y0] to [X142, Y159].

The coordinate system used for the 18 Kbit block RAMs in the FPGA is different from that used for the slices and tri-state buffers. The Virtex-II XC2V4000 FPGA has 120 block RAMs arranged in six columns with twenty blocks in each column. So the coordinates for the block RAMs range from [X0, Y0] to [X5, Y19]. The multiplier blocks, that are associated with the block RAMs, follow the same coordinate system. These coordinate systems discussed above are important because they are used in assigning the area group constraints for the static and reconfigurable modules and location constraints for the other components.

5.3.2 Boundaries of the Static and Reconfigurable Modules

As mentioned earlier, the reconfigurable module needs to include one column of block RAMs. Since the static module was placed on the left and the reconfigurable module on the right, the rightmost column of block RAMs has to be part of the reconfigurable module. This column is located between the slices with X-coordinate 139 and X-coordinate 140. So the boundary of the static and reconfigurable modules could not be any further right than the slices with X-coordinates 136. Since space on the FPGA was not at a premium for this design, it was decided to put the boundary between the static and reconfigurable modules at the slices with X-coordinate 128. Based on this, the area group constraints for the static and reconfigurable modules were specified as follows:

Static Module

- Range of slices = **[X0, Y159] - [X127, Y0]**
- Range of tri-state buffers = **[X0, Y159] - [X126, Y0]**
- Range of BlockRAMs = **[X0, Y19] - [X4, Y0]**
- Range of Multipliers = **[X0, Y19] - [X4, Y0]**

Reconfigurable Module

- Range of slices = **[X128, Y159] - [X143, Y0]**
- Range of tri-state buffers = **[X128, Y159] - [X142, Y0]**
- Range of BlockRAMs = **[X5, Y19] - [X5, Y0]**
- Range of Multipliers = **[X5, Y19] - [X5, Y0]**

In addition to the static and reconfigurable modules, there are other entities, namely the LUTs, the bus macros, and the DCM that need to be assigned constraints in the UCF file. There are fifteen address lines, eight data lines and three control lines that need to cross between the static and reconfigurable parts. Since each bus macro can route four signals across the boundary, a total of six bus macros are needed. These bus macros need to be placed so that they straddle the boundary properly. The composition of a bus macro was shown in Section 5.2. It consists of eight tri-state buffers. As such, it stretches across four

CLBs. Hence, for the signal to be routed correctly, the bus macro needs to be placed two CLBs before the boundary. In the case of the current design, the boundary is at X-coordinate 128. So the X-coordinate for the first tri-state buffer in all bus macros has to be 124. The location constraints for the bus macros are as follows:

- Location of bus macro for control signals = Tri-state buffer [**X124, Y30**]
- Location of bus macro for data [3:0] signals = Tri-state buffer [**X124, Y34**]
- Location of bus macro for data [7:4] signals = Tri-state buffer [**X124, Y38**]
- Location of bus macro for address [3:0] signals = Tri-state buffer [**X124, Y42**]
- Location of bus macro for address [7:4] signals = Tri-state buffer [**X124, Y46**]
- Location of bus macro for address [11:8] signals = Tri-state buffer [**X124, Y50**]
- Location of bus macro for address [14:12] signals = Tri-state buffer [**X124, Y54**]

The LUTs that enable or disable the tri-state buffers in the bus macros are placed in slices close to the bus macros on either side of the boundary. Hence the location constraints for the four LUTs are as follows:

- Location of ground LUT in static part = [**X120, Y38**]
- Location of VCC LUT in static part = [**X120, Y42**]
- Location of ground LUT in reconfigurable part = [**X134, Y38**]
- Location of VCC LUT in reconfigurable part = [**X134, Y42**]

Finally, all the IOB positions and attributes are locked in the UCF file. The complete list of constraints that were applied to the network controller design can be found in Appendix A. After these constraints were finalized, the complete design was routed using Xilinx tools to verify that the signals and the logic were indeed being placed correctly. The tools that were used to assist in the verification were Floorplanner and FPGA Editor. Figure 5.6 shows screen captures from the Floorplanner tool. The figure on the left shows the separation of the static and reconfigurable parts along the boundary at X=128. The figure on the right shows the top-level signals and I/Os in the static portion of the design. It can be clearly seen that none of the top level signals from the static module cross over into the boundary of the reconfigurable module.

The next section details the manner in which the final bitstreams are assembled.

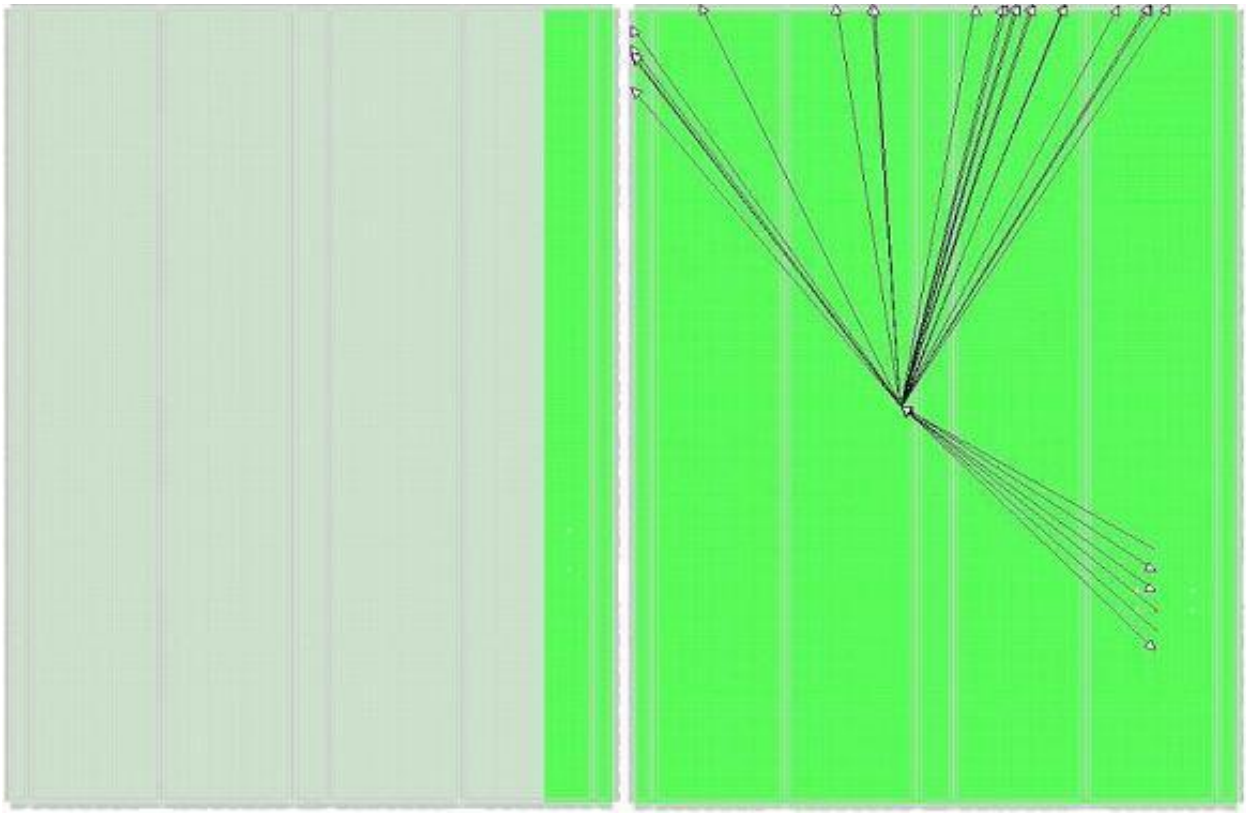

Figure 5.6: Module boundaries as seen in Floorplanner

5.4 Active Module and Final Assembly

Before commencing the active module phase, the directory structure recommended by Xilinx for the modular design flow was created [14]. This structure separates the source files from the files needed for initial budgeting and those for assembly. Figure 5.7 shows the directory structure that was created for the partial bitstream generation.

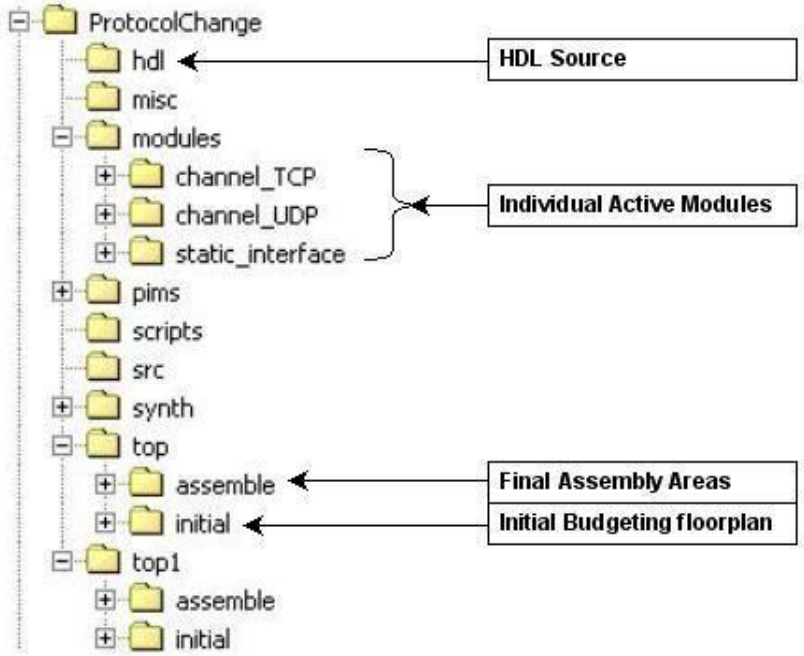

Figure 5.7: Directory structure for modular flow

5.4.1 Active Module

In the active module phase, bitstreams are generated for each of the reconfigurable modules. The process for creating the bitstream uses the same Xilinx tools that are used for creation of complete bitstreams: *ngdbuild*, *map*, *par* and *bitgen*. However, different options in the tools are used. The *ngdbuild* tool, which creates a design database, is run for each of the reconfigurable modules with the '-modular module active' option. This informs the tool to create a modular design in active mode. The *map* tool, which maps the logic gates of the design to CLBs and IOBs of the FPGA, and the *par* tool, which places and routes the logic components are run without any special options. The final tool used is *bitgen*, which creates the configuration bit file for the FPGA. This tool is run with the option 'ActiveReconfig: Yes'. This option prevents the assertion of the GSR (reset) signal during configuration. As such it ensures that the rest of the FPGA remains in operation when the partial bitstream is being downloaded. Another option used is 'Persist: Yes'. This option is necessary if the

FPGA is programmed using the SelectMAP mode, as is the case with the current setup. This is because it allows the SelectMAP pins to persist after the device is configured, which in turn allows the SelectMAP interface to be used for reconfiguration.

At the end of the active module phase, the *pimcreate* tool is used to publish the routed design to the 'Pims' folder from where they will be used in the final assembly phase [14].

5.4.2 Final Assembly

The final assembly phase involves combining the individual modules created by the Active Module phase into a complete FPGA design. The final assembly stage is required because it is necessary that the bitstream that is initially loaded into the FPGA be a complete one [14]. This is to ensure that all global, non-reconfigurable portions of the design are locked down and that only the reconfigurable modules change when the partial bitstream is loaded.

In the case of the current design, the initial bitstream has to have the static module and the reconfigurable module in which the fourth channel is initialized using the TCP protocol. The partial bitstream that was generated in the Active Module phase would be used to reconfigure the fourth channel for using the UDP protocol. To generate the complete bitstream, the Xilinx tools mentioned earlier were used once again but with different options. The most important one is that the *ngdbuild* tool needs to be given the option '-modular assemble' so it can combine the different files placed by the Active Module phase in the 'Pims' folder. Once this is done, the rest of the tools are used with the same options as in the Active Module phase. The only difference is that the *bitgen* tool is not given the 'ActiveReconfig: Yes' option since this is not necessary when creating a complete bitstream.

In this manner, the partial and complete bitstreams were created for reconfiguring one of the channels of the IIM7010 network controller using a different protocol. The same procedure was also used to create the partial bitstreams for initializing additional channels and for changing the protocol parameters in the channels. The next chapter shows the effects that partial reconfiguration can have on the performance of the network controller.

5.5 Summary

This chapter presented the process of creating partial bitstreams for the IIM7010 network controller. It started off with an overview of applications that would benefit from having a partially reconfigurable network. Then it examined in detail the procedure for creating the modules, the floorplan and finally, the bitstreams for the partial designs.

Chapter 6

Results and Analysis

The purpose of this research was to implement a partially reconfigurable network controller on an FPGA. The first step in this direction was taken by creating the design for a network controller on a Xilinx XCV1000 FPGA. The functioning of this controller was then verified. This verification is discussed in Section 6.1. For the network controller to be used in different applications, it was necessary to know its performance. As such, this section also examines the performance of the network with respect to some standard parameters. In Section 6.2, an application that uses the IIM7010 network controller is studied. Finally, Section 6.3 examines the working of the network controller after partial reconfiguration was implemented on the Xilinx Virtex-II XC2V4000 FPGA.

6.1 Verification of Functioning of Network Protocols

6.1.1 TCP

Chapter 4 detailed the creation of state machines on the FPGA for controlling the TCP and UDP protocols. To test the functioning of the TCP protocol implementation on the network controller, custom-built as well as commercially available test software was used. The setup

shown in Figure 6.1 was used for conducting the tests.

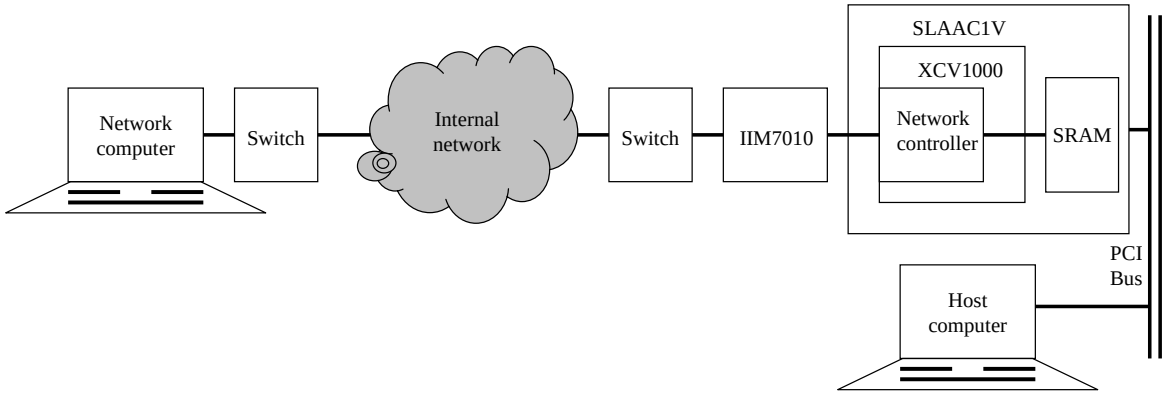

Figure 6.1: Setup for verifying TCP protocol implementation

The network computer shown in Figure 6.1 was running a program called 'Connecting Sockets' that is available for free from Iego Development [31]. This software was initialized in the 'Server' mode so that the IIM7010 network controller would make a connection to it. After that, a file was sent over the network to the IIM7010. This file was stored by the network controller in one of the SRAMs connected to the X2 FPGA in the SLAAC-1V platform. The file was then copied to the host computer over the PCI bus and compared to the original file sent over the network. Following are the contents of the first few locations of Memory 0, which is the first memory in the X2 SRAM bank, as viewed in hexadecimal format.

```
0000000: a0d3 c09b 84a1 c304 d28a f52c ebad b20d ...........,....
0000010: 619c 669e b1d8 a74d 74b6 2eae 5f23 a6ed a.f....Mt..._#..
0000020: a5f0 fb96 f552 0180 2ab3 7ab8 e192 dfe9 .....R..*.z.....
0000030: 3b40 9541 5107 a43c 2556 7fbe 27aa b49a ;@.AQ..<%V..'...
0000040: f75c 831a 63d3 1c7d e3e1 1484 a472 8cbb .\..c..}.....r..
0000050: 62bd adea 9b9e 55ac 8cb5 9de2 431d 8817 b.....U.....C...
```

This memory content was compared to the file that was transferred over the network, whose initial contents are shown below in hexadecimal format.

```
0000000: a0d3 c09b 84a1 c304 d28a f52c ebad b20d ...........,....
0000010: 619c 669e b1d8 a74d 74b6 2eae 5f23 a6ed a.f....Mt..._#..
0000020: a5f0 fb96 f552 0180 2ab3 7ab8 e192 dfe9 .....R..*.z.....
0000030: 3b40 9541 5107 a43c 2556 7fbe 27aa b49a ;@.AQ..<%V..'...
0000040: f75c 831a 63d3 1c7d e3e1 1484 a472 8cbb .\..c..}.....r..
0000050: 62bd adea 9b9e 55ac 8cb5 9de2 431d 8817 b.....U.....C...
```

When the two files were compared, it was seen that they were exactly the same. The comparison was performed using the utility *xxdiff*. This test was repeated several times and with different types of data untill it was verified that the TCP protocol implementation was functioning correctly.

6.1.2 UDP

Figure 6.2 shows the set-up for testing the UDP protocol implementation.

For testing the UDP protocol, two previously written C programs, *talk* and *listen-udp* were modified [1]. The *talk* program sends packets of a user-specified size at certain intervals to a remote IP address. The packets were numbered sequentially and were of a pre-specified format. This was to ensure that it would be possible to see the order of the packets on the receiving side and to locate the number of lost packets. The *listen-udp* program receives the packets on a given port and places them into a text file, where a parsing program can go through them and determine the packets that were missing and out of order. Figure 6.3 shows screen captures of the *talk* and *listen-udp* programs.

Based on the information obtained from running a large amount of data through the UDP test setup, it was concluded that the UDP protocol was working correctly. The above-mentioned programs were also used to obtain information on the effective throughput of the system and the packet loss in the case of UDP.

[1] The *talk* and *listen-udp* were written by Dr. Scott Harper.

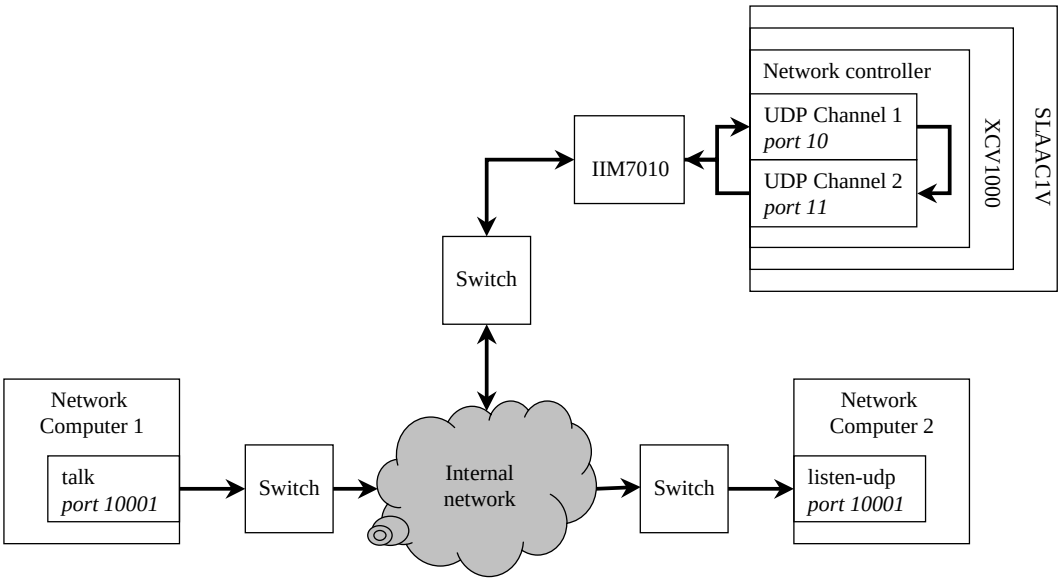

Figure 6.2: Setup for verifying UDP protocol implementation

To gain an understanding of the performance of the network, experiments were conducted to find the throughput of the IIM7010 network controller when connected to a 10 Mbps Ethernet network. The first set of experiments consisted of finding the throughput of the network controller for varying sizes of packets for both the TCP and UDP protocols. To find the throughput, a large amount of data was sent to the controller on a single channel. A counter was incorporated in the network controller design to keep track of the time elapsed between reception of the first and last packets. After the last packet was received, a timeout occured and the value of the counter was written to memory. It was later read by the host computer. Figure 6.4 shows a graph of the throughput of the network controller in Kilobits per second (Kbps) versus the effective packet size in bytes for both the TCP and UCP protocols. The packet sizes for both TCP and UDP are obtained by adding the size of a typical header to the size of the packet itself. The typical size of the header is 20 bytes for TCP and 8 bytes for UDP [32].

As can be seen from the graph, the throughput for both protocols is low when the packet size is less than 100 bytes. This is mainly because of the underlying IP protocol. Regardless

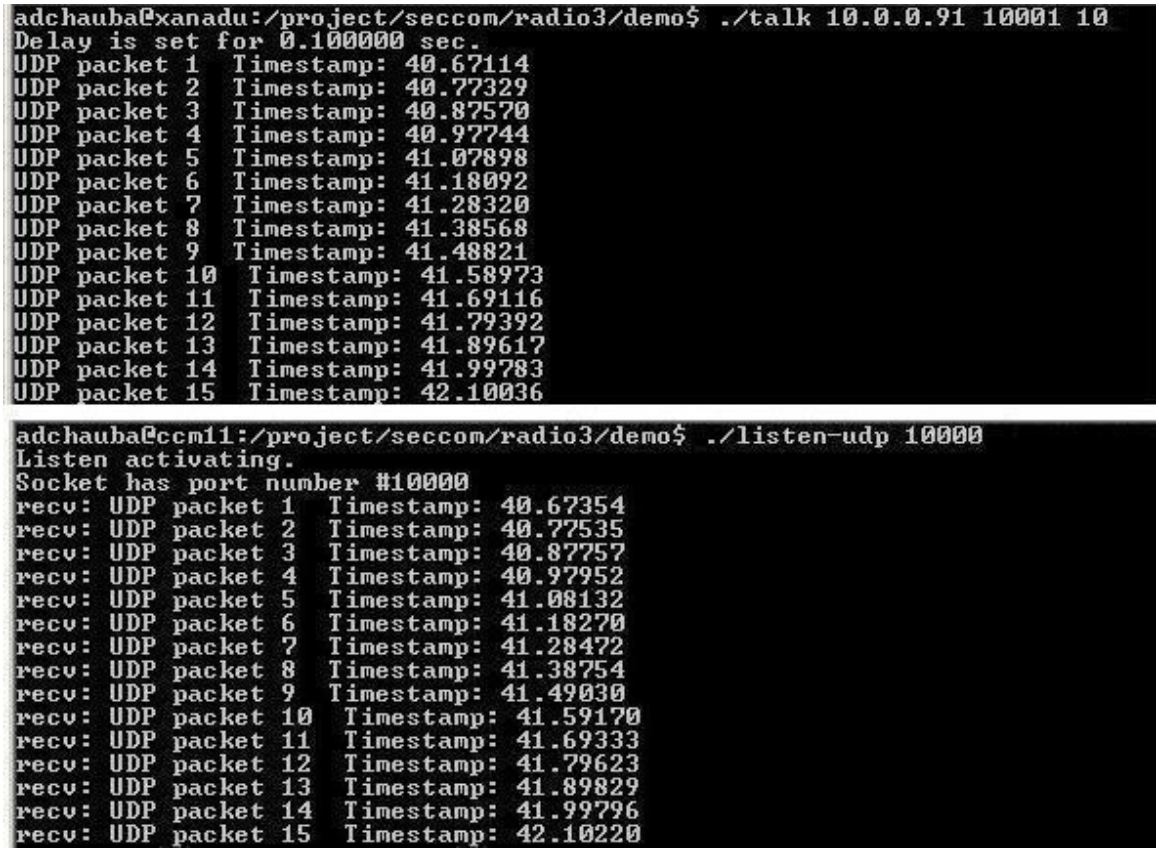

Figure 6.3: Screen captures of *talk* and *listen-udp* programs

of the packetization in TCP and UDP, IP fragments have to be a certain length. For very small TCP and UDP packets, a significant amount of network traffic is generated because of the large number of IP packets that hold little data and a large amount of padding. The padding is needed to bring the packets to the required IP packet size, which is usually 1480 bytes [26]. Both protocols peak at a maximum packet size in the range of 1024 to 2048 bytes. This conforms to expectations because the Maximum Transmission Unit [MTU] for Ethernet protocol is 1500 bytes [26]. Beyond this point even though the TCP or UDP packet may be bigger, it is still fragmented as per the underlying protocols. It is also seen that the throughput of the UDP protocol is significantly higher than that of TCP. This is also expected because the TCP protocol has significantly more overhead than UDP. The UDP protocol peaks at a maximum throughput of more than 7 Mbps, which is fairly close to the

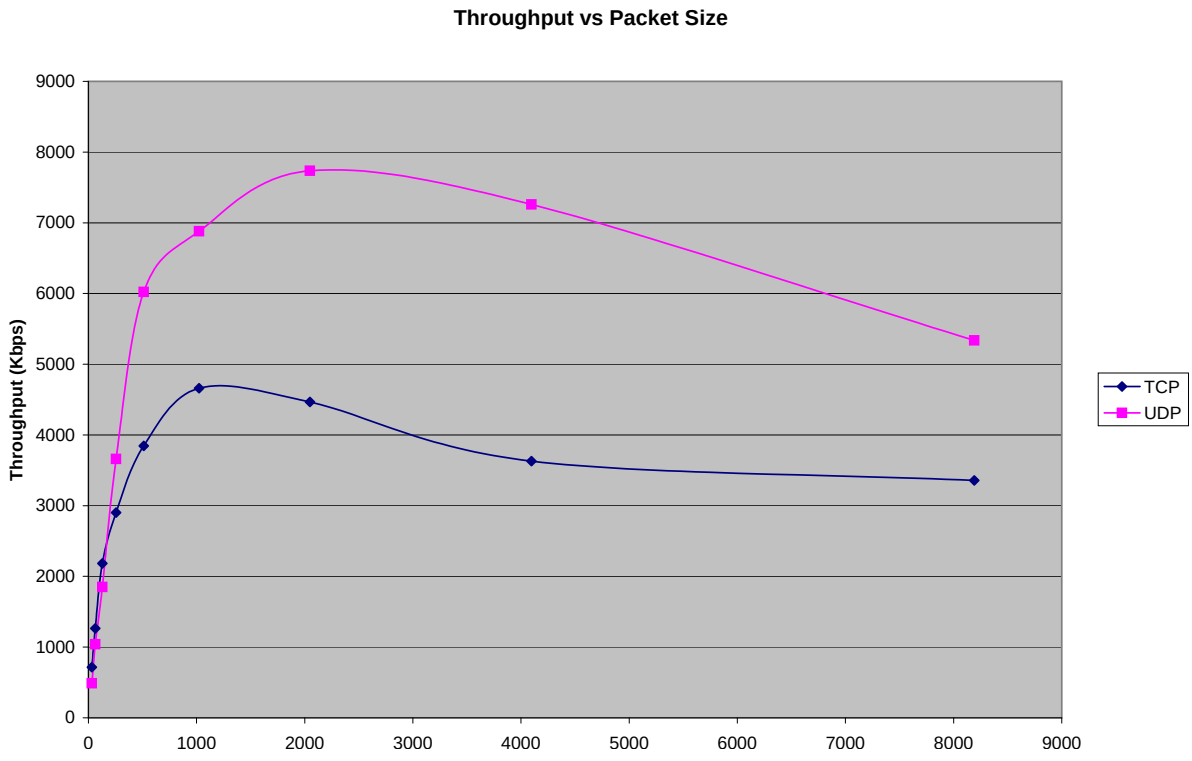

Figure 6.4: Chart of Throughput vs. Packet size through the IIM7010 network controller

limit of 10 Mbps limit for the network controller set by the W3100A.

The next set of experiments conducted was for determining the packet loss when the network controller was used as a router. The set-up used was the same as shown in Figure 6.2. The data being exchanged was going over one LAN network between different switches. The packet size for the TCP and UDP packets was varied between 32 bytes and 8192 bytes.

It is seen from the graph in Figure 6.5 that there is no packet loss for TCP. This is expected because TCP is a reliable protocol. However, it is seen that for small packet sizes, UDP has an unacceptably high percentage of packet loss. As the size of the packet increases, the packet loss for UDP falls to more acceptable levels. The reason for this is the same as the reason for the increase in throughput that accompanies an increase in packet size. Because

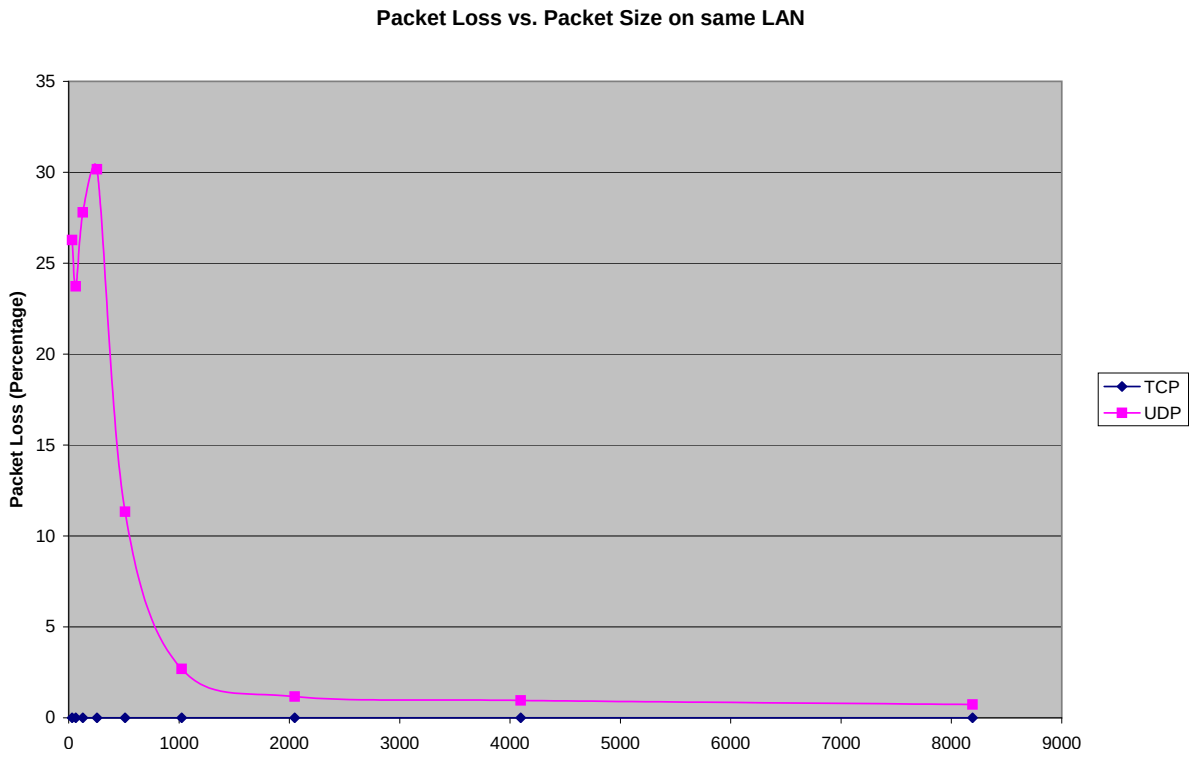

Figure 6.5: Chart of Packet loss vs. Packet size through the IIM7010 network controller (same LAN)

of the underlying protocols, there are more packets created for the same amount of data when the packet size is very small. The same experiment was later repeated but with the sending and receiving computers and the IIM7010 controller all attached to the same switch. The graph in Figure 6.6 shows the change in packet loss against packet size for that setup.

It is seen that the packet loss for UDP is much less than in the previous case because there are less collisions on the same switch. However it follows the same pattern as the previous graph in that the packet loss decreases significantly as the packet size increases.

The next section provides a brief overview of an application of the IIM7010 network controller.

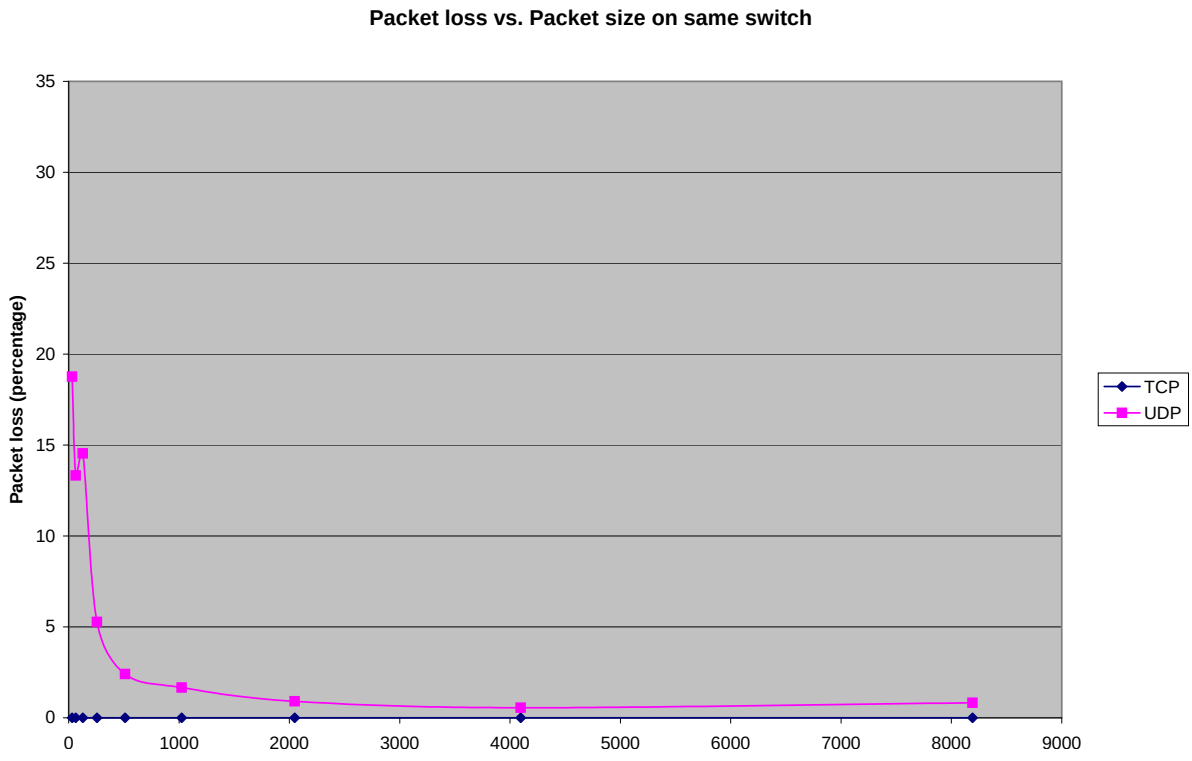

Figure 6.6: Chart of Packet loss vs. Packet size through the IIM7010 network controller (same switch)

6.2 Application for the IIM7010 Network Controller

The IIM7010 network controller that was developed in this research forms a part of the Secure Communications System [19]. This system is also implemented on the SLAAC-1V FPGA platform on which the network controller was built. Figure 6.7 shows a block diagram of the system.

In this system, the IIM7010 network controller initially establishes a TCP connection over a single channel with a remote bitstream server. Whenever a user is authenticated by the system, the network controller receives a tag, which it sends out to the bitstream server. The server then sends back a bitstream to configure FPGA X1 with a user application. One

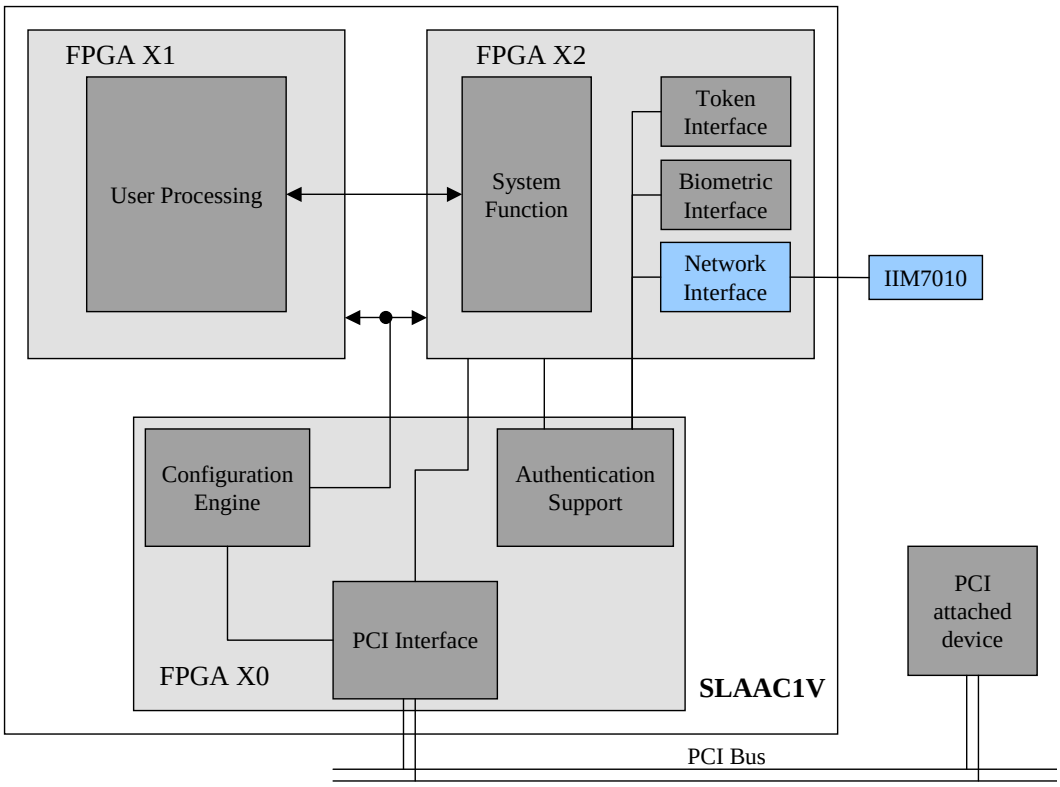

Figure 6.7: Block diagram of Secure Communications System [19]

of the user applications created involved a PCI-based software radio interfaced with another IIM7010 network controller. The data received by the radio controller was sent over the Ethernet network by the IIM7010 controller using the UDP protocol and vice versa. The interface between the network controller and the radio consisted of two FIFOs. Both the designs that contained the IIM7010 controller were tested and were found to be functioning correctly. The IIM7010 was able to connect with the remote bitstream server, obtain a bitstream and store it in SRAM memory. The network controller was also tested in a loop-around configuration with the radio receiver and transmitter in the same PCI-based radio device and was found to be working correctly.

The next section contains the results of experiments that tested the partial reconfiguration capability of the IIM7010 network controller.

6.3 Partial Reconfiguration Results

It was mentioned in Section 5.1.3 that one of the experiments designed to test a practical application of partial reconfiguration involved changing certain protocol parameters and observing their impact on the network controller's performance. As such, a complete bitstream was created that had two different channels in the IIM7010 communicating with two different network computers using the UDP protocol. The maximum packet size for both channels was set to 128 bytes. The Xilinx Virtex-II XC2V4000 FPGA, on which the network controller was implemented, was then divided into static and reconfigurable modules so that Channel 2 would be placed in a separate module on the right edge of the chip. Following this, a partial bitstream was created in which Channel 2 was reconfigured to have a maximum segment size of 1024 bytes while still using UDP for communication.

The experimental setup for this test was similar to that shown in Figure 6.2. The packets received by the IIM7010 network controller were stored in the SRAM memory connected to the FPGA and later read from it using the PCI interface to the host computer. To determine the value for throughput at various intervals, a counter was added to the design and its value was written along with each packet into memory. When the data from the memory was obtained, a program was written to find the time of reception of each packet based on the value of the counter and the frequency of the FPGA clock. This allowed the calculation of throughput over different intervals of time for each of the two channels. The graph in Figure 6.8 shows the value for throughput for the two channels calculated over intervals of two seconds from zero to twenty seconds. The partial bitstream was downloaded to the XC2V4000 FPGA approximately eight seconds after the start of the experiment.

It can be seen from Figure 6.8 that the throughput for both channels stabilizes at approximately 1200 Kbps after a time interval of about four seconds. For Channel 1, which is not reconfigured, the value of throughput remains between 1100 Kbps and 1200 Kbps over the entire twenty second interval. But the throughput for Channel 2 dips to about 700 Kbps over the time interval from eight to ten seconds. After this, the throughput climbs up to over 3700 Kbps and then stabilizes at around 3500 Kbps.

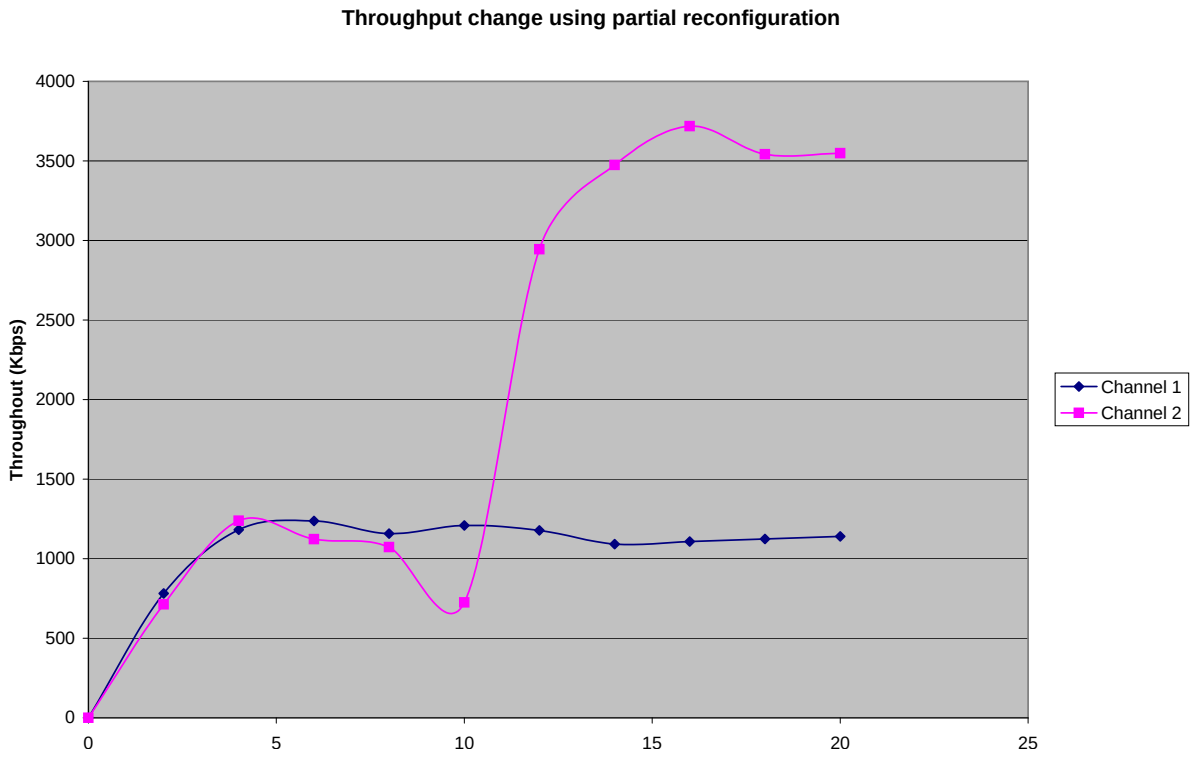

Figure 6.8: Throughput change as a result of partial reconfiguration

This graph provides a good evidence of the partial reconfiguration operation. It is seen that Channel 1, which always has a maximum packet size of 128 bytes always has a throughput of around 1200 Kbps. This channel continues to transmit data even as the partial bitstream is being downloaded to the FPGA. On the other hand, the throughput of Channel 2 increases from 1200 Kbps to 3500 Kbps after it is reconfigured to have a maximum packet size of 1024 bytes. This conforms to the data obtained from the graph in Figure 6.4, which shows that throughput for UDP increases when the packet size is increased. The decrease in the throughput over the interval from eight to ten seconds is caused by the reconfiguration of Channel 2 because during the time required for reconfiguration, the throughput for that channel is zero.

During this same experiment, the packet loss of each channel was also measured using the

numbering of the packets captured. Figure 6.9 shows a graph of the packet loss for the two channels over the same interval from zero to twenty seconds.

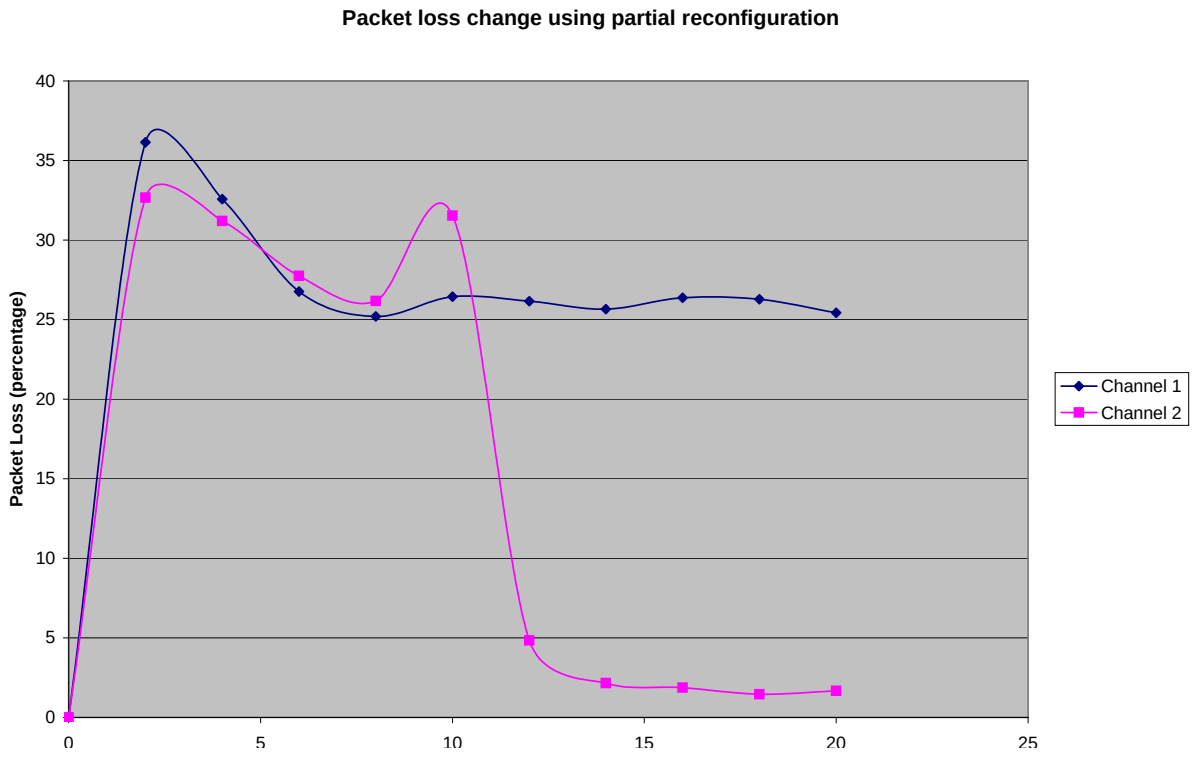

Figure 6.9: Change in packet loss as a result of partial reconfiguration

It can be seen that the packet loss for both channels is initially around 30% and stabilizes at a little over 25% for Channel 1. In the case of Channel 2, it is seen that the packet loss momentarily increases to over 30% in the period form eight to ten seconds and then drops dramatically to around 2%. This behavior conforms to expectations because it was seen from the graph in Figure 6.5 that the packet loss for a channel using UDP over a LAN was approximately 27% when the maximum packet size was 128 bytes. For 1024 byte packets, the packet loss was less than 2%. Channel 1, which always has a maximum packet size of 128 bytes has a constant packet loss of a little over 25%. In the case of Channel 2, the packet loss drops to less than 2% after it is reconfigured. As in the case of throughput, the packet

loss in the interval in which reconfiguration occurs increases slightly because the channel drops all packets while it is being reconfigured.

Based on these observations, it was inferred that partial reconfiguration of the channels of the network controller works correctly.

6.4 Summary

This section presented the results obtained from conducting various tests on the IIM7010 network controller. Some of these tests were meant to verify the functioning of the protocols implemented in the network controller while others gave proof of its ability to undergo partial reconfiguration. The results of all the tests conformed to expectations, leading to the conclusion that the IIM7010 partially reconfigurable network controller had been implemented successfully.

Chapter 7

Conclusion

7.1 Summary

This thesis has demonstrated the successful implementation of a partially reconfigurable network controller on an FPGA. It has conducted tests to quantify the impact of partial reconfiguration on the performance of network controllers. These tests showed the benefits of incorporating the property of partial reconfiguration into a network-controlling device. Tests were also conducted on the IIM7010 network controller itself to measure its performance in terms of throughput and packet loss. These tests showed that the IIM7010 controller is capable of achieving throughput rates close to the limits imposed by the hardware. While these rates are not impressive when compared to Gigabit Ethernet transceivers, the current implementation was intended as a proof-of-concept to demonstrate the feasibility of using a combination of an FPGA and an ASIC to create a network controller that lent itself to partial reconfiguration. One of the advantages of the strategy of using an ASIC for part of the processing was the minimal use of FPGA resources as noted in Section 4.5. Overall the contributions made by the research that led to this thesis can be summarized as under:

- A network controller was created using an FPGA for controlling the higher layer network protocols and an ASIC for implementing the lower layer packetization.

- The throughput and packet loss for this network controller were measured using different network configurations and its performance was evaluated.
- Partial bitstreams were created to initialize additional channels, change the network protocol for a given channel and to alter the maximum packet size for specified channels in the network controller.
- The effectiveness of partial reconfiguration was demonstrated by reconfiguring one channel in the network controller with a different maximum packet size and showing its positive impact on the performance of that channel.

The design of the IIM7010 network controller also has some drawbacks as compared to an entirely FPGA-based implementation. These drawbacks are in terms of flexibility. The IIM7010 network controller can only communicate using the TCP/IP family of protocols over a 10 Mbps Ethernet LAN network. Changing the type of network or the communication protocols will require redesign of the state machines in the FPGA and a change in the hardware used for handling the data link and physical layers. But this system is well-suited for use in the wide majority of today's networks. This is because most networks in homes and offices today are 10/100 Mbps Ethernet LAN networks and the TCP/IP family of protocols is widely used in various applications, as was noted in Chapter 1.

7.2 Future Work

The research conducted for this thesis has opened up a number of possibilities for future work. The most important one is the shifting of the processing that is currently being performed in the W3100A ASIC to the FPGA to make the design more flexible. If the implementation of the MAC core can be reduced in size and included in the FPGA instead of the ASIC, this would lead to the network controller being able to interface with different LAN technologies without requiring a change in the hardware. Another area of research involves the modification of the network controller design to accommodate higher data rate transmission since a lot of potential applications require much higher data transfer rates than

those achieved by the IIM7010 network controller. Incorporating these additional features into the partially reconfigurable network controller designed in this thesis will greatly increase its applications in industry.

Appendix A

UCF File

The constraints shown here are obtained from the UCF file for the partially reconfigurable network controller discussed in sections 5.2 and 5.3. The segment of the file shown here contains area group constraints for the static and reconfigurable entities. The whole file is not reproduced since most of its constraints are related to I/O pins and are not directly related to partial reconfiguration. It also contains location constraints for the bus macros, LUTs and other entities that are part of the design.

```
-- UCF file segment for XC2V4000 FPGA
-- Contains area group constraints for 'STATIC_PART' and 'RECONFIG_PART'
-- Contains location constraints
-- Aditya Chaubal


INST "STATIC_PART" AREA_GROUP = "AG_STATIC";
AREA_GROUP "AG_STATIC" RANGE = SLICE_X0Y159:SLICE_X127Y0;
AREA_GROUP "AG_STATIC" RANGE = TBUF_X0Y159:TBUF_X126Y0;
AREA_GROUP "AG_STATIC" RANGE = RAMB16_X0Y19:RAMB16_X4Y0;
AREA_GROUP "AG_STATIC" RANGE = MULT18X18_X0Y19:MULT18X18_X4Y0;
AREA_GROUP "AG_STATIC" MODE = RECONFIG;

INST "RECONFIG_PART" AREA_GROUP = "AG_RECONFIG";
AREA_GROUP "AG_RECONFIG" RANGE = SLICE_X128Y159:SLICE_X143Y0;
```

```
AREA_GROUP "AG_RECONFIG" RANGE = TBUF_X128Y159:TBUF_X142Y0;
AREA_GROUP "AG_RECONFIG" RANGE = RAMB16_X5Y19:RAMB16_X5Y0;
AREA_GROUP "AG_RECONFIG" RANGE = MULT18X18_X5Y19:MULT18X18_X5Y0;
AREA_GROUP "AG_RECONFIG" MODE = RECONFIG;

INST "BUSMACRO_CONTROL" LOC = TBUF_X124Y30;
INST "BUSMACRO_DATA_1" LOC = TBUF_X124Y34;
INST "BUSMACRO_DATA_2" LOC = TBUF_X124Y38;
INST "BUSMACRO_ADDR_1" LOC = TBUF_X124Y42;
INST "BUSMACRO_ADDR_2" LOC = TBUF_X124Y46;
INST "BUSMACRO_ADDR_3" LOC = TBUF_X124Y50;
INST "BUSMACRO_ADDR_4" LOC = TBUF_X124Y54;

INST "LUT_GND_left" LOCK_PINS;
INST "LUT_GND_left" LOC = "SLICE_X120Y38";
INST "LUT_VCC_left" LOCK_PINS;
INST "LUT_VCC_left" LOC = "SLICE_X120Y42";
INST "LUT_GND_right" LOCK_PINS;
INST "LUT_GND_right" LOC = "SLICE_X134Y38";
INST "LUT_VCC_right" LOCK_PINS;
INST "LUT_VCC_right" LOC = "SLICE_X134Y42";

INST "mem_clk_loop_bufg" LOC = "BUFGMUX6P";

INST "mem_clk_dll" LOC = DCM_X0Y0;
```

Bibliography

[1] J. Riihijarvi, P. Mahonen, M. Saaranen, J. Roivainen, and J. Soininen, "Providing Network Connectivity for Small Appliances: A Functionally Minimized Embedded Web Server," *IEEE Communications Magazine*, vol. 39, pp. 74–79, October 2001.

[2] H. Fallside, M. John, and S. Smith, "Internet Connected FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 289–290, April 2000.

[3] J. Lockwood, "Evolvable Internet Hardware Platforms," *Third NASA/DoD workshop on Evolvable Hardware*, pp. 271–279, July 2001.

[4] H. Kalte, D. Langen, E. Vonnahme, A. Brinkmann, and U. Ruckert, "Dynamically Reconfigurable System-on-Programmable-Chip," *10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pp. 235–242, January 2002.

[5] J. Martin, K. Champman, and J. Leben, *Local Area Networks: Architectures and Implementations*. Englewood Cliffs, NJ: P T R Prentice Hall, second ed., 1994.

[6] J. Kurose and K. Ross, *Computer Networking: A Top Down Approach Featuring the Internet*. Boston, MA: Addison Wesley Longman, 2001.

[7] E. Skoudis, *Counter Hack*. Upper Saddle River, NJ: Prentice Hall PTR, 2002.

[8] V. Irwin and H. Pomeranz, "Intrusion detection & packet filtering," Blacksburg, VA, 2002.

[9] IEEE Computer Society, ed., *IEEE 802.3: CSMA/CD Access Method*, (New York, NY), The Institute of Electrical and Electronics Engineers, March 2002.

[10] R. Jaganathan, K. Underwood, and R. Sass, "A Configurable Network Protocol for Cluster Based Communications using Modular Hardware Primitives on an Intelligent NIC," *11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2003)*, pp. 286–287, April 2003.

[11] P. Bellows, J. Flidr, T. Lehman, B. Schott, and K. Underwood, "GRIP: A Reconfigurable Architecture for Host-Based Gigabit-Rate Packet Processing," *10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2002)*, pp. 121–130, April 2002.

[12] B. Schott, C. Chen, S. Crago, J. Czarnaski, M. French, I. Hom, T. Tho, and T. Valenti, "Architectures for System-Level Applications of Adaptive Computing," *7th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 1999)*, pp. 270–271, April 1999.

[13] T. Markovic, K. Nguyen, and I. Mohor, "Ethernet MAC 10/100 Mbps." http://www.opencores.org/projects.cgi/web/ethmac/overview, May 2004.

[14] Xilinx, San Jose, CA, *Two Flows for Partial Reconfiguration: Module Based or Difference Based*, November 2003. Application Note.

[15] Xilinx, San Jose, CA, *Xilinx FPGA Editor Version 6.2.01i*, 2004. Software Tool.

[16] WIZnet, Seoul, Korea, *IIM7010 Training Guide*, 2002. Datasheet.

[17] WIZnet, Seoul, Korea, *W3100A Datasheet Rev 1.3*, 2002. Datasheet.

[18] Realtek Semiconductor Corp, Taiwan, *RTL8201CL: Single-Chip/Single-Port 10/100M Fast Ethernet Phyceiver Datasheet Rev 1.2*, January 2004. Datasheet.

[19] S. J. Harper, *A Secure Adaptive Network Processor*. PhD thesis, Virginia Polytechnic Institute and State University, Blacksburg, VA, April 2003.

[20] Xilinx, San Jose, CA, *The Programmable Logic Data Book*, 1999.

[21] Information Sciences Institute - East, USC, Arlington, VA, *SLAAC-1V VHDL Users Guide Release 0.1.3*, 2001. Datasheet.

[22] CadSoft, Delray Beach, FL, *Eagle Schematic and Layout Editor Version 4.09*, 2001. Software Tool.

[23] T-Tech, Atlanta, GA, *Model 5000 Quick Circuit System*, 2002.

[24] Xilinx, San Jose, CA, *Virtex-II Handbook*, 2002. Datasheet.

[25] The DINI Group, La Jolla, CA, *DN3000k10 User's Manual Version 1.4*, 2003. Datasheet.

[26] W. R. Stevens, *TCP/IP Illustrated: The Protocols*, vol. 1. Boston, MA: Addison-Wesley, 1994.

[27] Synplicity, Sunnyvale, CA, *Synplify Pro Version 7.2*, 2002. Software Tool.

[28] WIZnet, "Wiznet Technical Support Q&A." http://www.iinchip.com/wiznet/qna_old.html, May 2003.

[29] Xilinx, San Jose, CA, *Status and Control Semaphore Registers Using Partial Reconfiguration*, June 1999. Application Note.

[30] Xilinx, San Jose, CA, *Xilinx Floorplanner Version 6.2.01i*, 2004. Software Tool.

[31] Iego Development, Enschede, The Netherlands, *Connecting Sockets Version 1.11*, 2001. Software Tool.

[32] W. R. Stevens, *TCP/IP Illustrated: The Implementation*, vol. 2. Boston, MA: Addison-Wesley, 1995.

Vita

Aditya Chaubal was born in Mumbai (formerly Bombay), India in 1980. He grew up in the suburbs of Mumbai and lived there till he was 17. In 1998, he enrolled at Rowan University, NJ and began work towards a degree in Engineering. He graduated *summa cum laude* from Rowan University in May 2002 with a B.S. in Electrical & Computer Engineering and a minor in Computer Science. Following this, he decided to continue his studies and obtain an M.S. degree in Computer Engineering. To this end, Aditya accepted admission at Virginia Tech University in the Fall of 2002 and started working in the Configurable Computing Laboratory under the guidance of Dr. Athanas. After graduating from Virginia Tech in 2004, he will commence work at Xilinx in Longmont, CO.