A DEVICE FOR SYNCHRONOUS ETHERNET PACKET DELAY

by

ROSS VONFANGE

B.S., Kansas State University, 2007

A THESIS

submitted in partial fulfillment of the
requirements for the degree

MASTER OF SCIENCE

Department of Electrical and Computer Engineering
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas
2009

Approved by:

Major Professor
Don Gruenbacher

Copyright

Abstract

This thesis presents a novel device for delaying Ethernet traffic in a lab setting. Ethernet is the leading standard for communications between computing devices. With the advent of streaming media such as voice over IP phone service and real-time control systems over Ethernet, applications are being rapidly developed that must meet strict communication reliability and timing constraints. Increasingly, these systems must be examined in real world scenarios before actual hardware deployment or protocol release. This increases the demand for both testing equipment and well trained network engineers. Commercial Ethernet delay testing devices are expensive, hardware specific, and not flexible enough for educational purposes. These short-comings make it necessary to design a robust Field Programmable Gate Array (FPGA) based Ethernet delay device that is up to the rigor of educational and research settings.

Our approach is based on the inexpensive, high performance Altera Stratix II GX PCI Express development board which can easily be adapted for different delay scenarios. The system's FPGA hardware was developed in Verilog, an industry standard hardware description language, so users will be able to quickly learn, adapt and operate the system. Software for the system's soft processor was developed in C.

The device provides a wide range of packet delay from nearly zero up to over fifty milliseconds, as well as providing an easy to use interface with on-the-fly variable delay adjustment. Theoretical throughput was up to 1Gb/s; skew and jitter measurements were comparable with common network switches. These properties allow the device to provide an easy-to-use, inexpensive method to delay Ethernet traffic in lab settings and the device also creates a starting point for future students and researchers to develop high speed traffic

delay testbeds. Future work will include 10Gb/s throughput, additional memory capacity and additional software implemented delay profiles.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we present a device to delay frames in an Ethernet network to simulate routing and wire time delay, which provides an accurate test bed for new Ethernet protocol development.

Networks developed to connect computers started to become mainstream in the early 1960's [2]. Out of need to connect mainframe computers, the Advanced research Projects Agency (ARPA), part of the Department of Defense (DOD), developed ARPANET to handle the task. The development of ARPANET spurred the development of a host of ideas and protocols, including the still widely used Transmission Control Protocol (TCP) [2]. Since those early networks, major advancements have been made in all aspects of networking, including improvements to speed, reliability, capacity and broadened use.

As networks became less expensive and more broad in scope of use, new types of information began to need different and more sophisticated protocols for transport. These new protocols would accommodate increased demand for reliability and speed, greatly increasing the bandwidth of networks. Different network connection types were also created to interconnect smaller networks with high-capacity backbones. This setup allowed developers to stream live communications across a large array of networks. Ethernet and its standard has emerged as one of the most widely used and cost effective wired solutions for networks, and remains dominate in the field of computer networking.

Several standards committees, including the International Organization for Standardization (ISO), Institute of American National Standards Institute (ANSI), and Electrical and Electronics Engineers (IEEE), currently exist to further develop Ethernet standards which help to increase interoperability of these systems. In this chapter, we will present a brief overview of current networks, background of the project, and develop the motivation for creating an Ethernet frame delay device for academic and professional use. We will provide an overview of the delay device, its composition and its perfomance and also provide details and concepts that can be researched and tested with the delay device.

1.1 Ethernet Delay

Ethernet currently dominates all other wired communications standards in the end consumer and business markets. Because Ethernet is so widespread, it is being adapted to fit a variety of roles for data transmission, such as hard, real time industrial controls. Multimedia exchange formats such as video, Voice Over Internet Protocol (VOIP), and audio streaming technologies for services such as Internet radio are increasing in popularity. These services produce massive quantities of data transmission on a daily basis, much of which travels over an Ethernet network at some point on its way from the server to the user. Delays are inherent in these systems, because of the time it takes an electrical signal to propagate down a copper wire and through any machines on its path to the end user.

1.1.1 Delay Effects

Real-time Ethernet systems and real-time controls have been developed to work on Ethernet and other packet switched networks. Ethernet is an enticing target to engineers because of its availability, reliability and most importantly, cost. Lee et. al. states,"Recently, the real-time industrial network has become an important element for intelligent manufacturing systems. Especially, as the systems are required to be more intelligent and flexible, the systems should have more field devices such as sensors, actuators, and controllers. [3]" Real-time devices require strict timing for signal transmission to function correctly.

Voice phone service is now almost entirely operated over packet switched networks that include Ethernet. VOIP service is quickly becoming cost effective and attractive to consumers due to its portability and cost. Phone conversations are a good example of a service where the user could directly observe delays in the communication. To keep the delay low, so that it is not noticeable to the user, priority queuing has been implemented in some networks to speed up latency affected protocols. However, certain delays, like routing and line delays, will always exist. These unavoidable delays in Ethernet systems are important to account for and test, which is why they are the focus of this project.

While some devices currently exist to simulate delays in systems for testing, most of these devices are prohibitivly expensive, do not offer high throughput due to software processing, and are not based on flexible field programmable gate array (FPGA) technology. Current devices that can be used to generate delays with throughput of 1Gb/s and higher cost hundreds of thousands of dollars [4]. Higher costs prohibit large-scale production and limit availability for research and educational use in labs and classrooms.

Because network technology has continued to swell in popularity, there are more students and developers, creating an obvious demand for an inexpensive, flexible platform for these users to test new protocols on.

1.1.2 Delay Device Description

The FPGA platform, which was chosen to develop the frame delay device, is produced by Altera and shown in Figure 1.1. This development board was specifially chosen because

of the large number of logical elements on the Stratix II GX, onboard Dual Data Rate Synchronous Dynamic Random Access Memory (DDR2 SDRAM), Flash, Synchronous Dynamic Random Access Memory (SDRAM) and two Small Form Pluggable (SFP) cages for Ethernet transceivers.

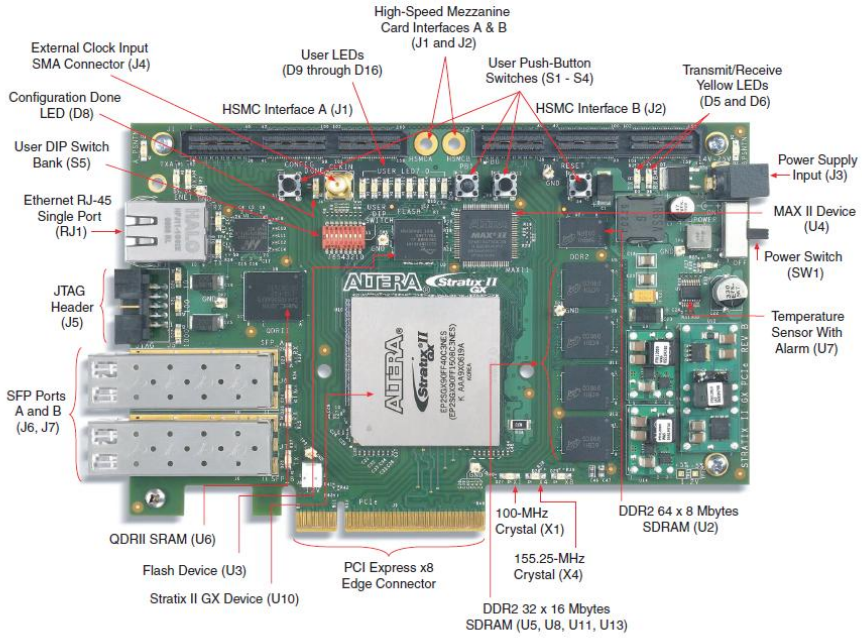

Figure 1.1: *Stratix II GX PCIe Developemnt Board [5]*

Having the ability to swap SFP modules allows users to quickly change from copper to optical connectors. The onboard memory provides enough capacity and memory bandwidth to handle high throughput rates simultaneously with large time delays.

The hardware description for the FPGA was written in Verilog. Several IP cores were used to construct the system, which were provided by donation from Altera. These cores include the Nios II soft processor, Triple Speed Ethernet Media Access Control (MAC), High Performance DDR2 Memory Controller and JTAG UART cores.

The software was written in C language and developed in Altera's Nios II EDS. The software provides a simple Command Line Interface (CLI) for the user to manipulate the frame delay. There is also a small, accompanying Application Programming Interface (API) that was created to provide users with simple-to-use functions when creating more complex delay profiles or timed delay periods with the device.

The majority of the cost for the system was in the initial cost of the FPGA development board. A less expensive hardware development board could be substituted into the design in the future, which would lower the per-unit cost substantially, even to as low as a few hundred dollars per board.

1.2 Motivation

As shown in the previous sections, delay generation is needed to properly develop new protocols and ensure they are tested correctly. These problems motivate our need to build such a device for use in educational and research settings that is flexible, easy to use and inexpensive. If classrooms and research labs then could feasibly use several devices concurrently, faster and more productive protocol development could be done.

This project was originally funded by Sandia National Laboratories. The Discom WAN group requested a device for testing their networks that could locally simulate the delay that was normally experienced with communications between other national labs and government agencies. This group has also motivated many of our device's fundamental platform specifications, such as using an FPGA-based device with hardware created in the widely-used Verilog language.

1.3 Key Contributions

This thesis presents the following key contributions:

In chapter 4 we present our approach to creating an Ethernet delay device for synchronous Ethernet frame delay and its features. We also provide the settings for the IP cores used in the project so future users can easily reproduce the device if they decide to create a new hardware variation.

1. Development on an inexpensive platform for a low-cost implementation. Because the design cost is low there is a low introduction cost and quick implimintation, which is especially useful in academic settings where FPGA resources are often avilable to the institution.

2. Development was done in Verilog HDL, which is widely used in industry and education. This also provides students with a learning tool as well as easy-to-modify code.

3. Software was developed in C because of the available programming environment (Nios II EDS) and C's superb ability to control hardware. The project also provides a small API for users so they can easily change the device's software parameters. These parameters include pause frame forwarding and generation, MAC address overwriting and filtering capabilities, and the ability to implement new delay profiles.

4. This approach provides inexpensive hardware products because it was produced on an FPGA (not an ASIC) and has reproducible, accurate results because it is not dependent on operating system resources and traditional fixed computer buffer sizes.

5. Development using the Avalon interface and the creation of custom components along with Altera's SOPC builder environment allows for future upgrades and system change to be quick and automated with wizards so the user will need to know little about the system in order to make changes in its operation or to reprogram the device.

6. The solution uses on-chip memory in the FPGA. This allows the design to migrate and scale easily across several development board and FPGA platforms.

7. On-the-fly adjustable frame delay time provides the user with the ability to implement scheduled delays and delay profiles to more closely model network traffic. Constant delays closely model a set number of routing and line delays with a higher accuracy and higher throughput then software implementations.

8. Easy device reconfiguration to allow the user to study buffer size effects in store forward systems. With the ability to monitor the hardware buffer level.

In chapter 5 we present the test results for the device:

1. Throughput of up to 200 MBits/s; when compared to other systems this is a substantial throughput.

2. Jitter and skew were comparable to most common cut-through switches.

3. Expected results were obtainted when a linear ramped delay profile is used.

4. Tested and attained reproducible delays of well over the targeted 50ms for any traffic type.

To summarize, the device fills its intended use and offers a wide range of features to the user. It can also be recreated on different FPGA low-cost devices. This could help institutions afford enough units to use in classroom settings where a limited number of units per room would normally bottle neck the students who need access to the device.

Chapter 2

Applicable Standards

In the following sections we will discuss the OSI model and its relation to Ethernet, Ethernet protocols and user applications. Applicable portions of the IEEE 802.3 protocol standard for Ethernet are explained in this chapter to provide a background on Ethernet, specifically Ethernet frames. A brief description of the medium and the mechanics of the transmission are also included. Finally, we will discuss the Avalon bus standard, a standard for communication between on-chip FPGA modules. This explanation will focus on the goal of informing the reader about the interconnection of the delay device, and to provide insight to the interoperability of the device's modules with future hardware implementations.

2.1 OSI Model

The Open Systems Interconnection Reference Model was developed to describe communication between devices. The model takes the process of transmitting application-level data and breaks it into its fundamental pieces [2]. The OSI model is shown in 2.1. When data is sent from an application, the data is encapsulated by the lower layers as the data travels down the model until, finally, the data is transmitted at the physical layer (layer 1).

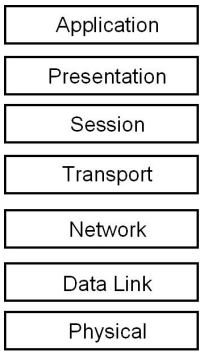

Figure 2.1: *OSI Reference Model [2]*

This project primarily deals with only the first two layers. The device only needs to accept and send the information on a frame by frame basis because the device is simply reproducing its incoming data after adding the desired delay time. No higher layers or higher level protocols need to be involved in the device's functionality, and they will be omitted from further discussion. The majority of this project's focus will be in layer two, the data link layer.

At the data link layer, the information that is sent and received is called a frame. An Ethernet frame consists of the fields described in Figure 2.3. The data is encapsulated into the frame at this stage by a Media Access Control header which includes, source address, destination address (MAC not IP), an optional virtual LAN tag called a Q-tag (to virtually divide shared networks), an Ether type, a Logical Link Control header. To complete the encapsulation a Cyclical Redundancy Check (CRC) is appended. The frame is passed down from the MAC layer to be transmitted by the Ethernet controller where the preamble and start of frame delimiter are generated on transmission and an interframe gap is inserted.

Generally, new protocols are developed at the layers above layer two. Developing hardware to delay information at the second layer, as opposed to a higher layer, is the best option because it reduces complexity and increases interoperability.

2.2 802.3 Ethernet Standard

This section summarizes some of the necessary portions of the IEEE 802.3 standard to bring the reader up-to-speed on the various parts and requirements made by the standard. Standard 802.3 contains nearly all the necessary specifications for a general use delay device because most networks use Carrier Sense Multiple Access with Collision Detection (CSMA/CD) at the physical layer when in half duplex mode, and adhere to the 802.3 specification for full duplex as well. It is worth noting that other specifications in the 802.X family are used but 802.3 is by far the most common type for wired networks. Our approach to summarizing the standard will be bottom-up and will focus on the type that was used to create the delay device (mostly 802.3 section 3 clauses 32 through 34 and Annex 36A through Annex 43C).

2.2.1 CSMA/CD and background

Carrier sense multiple access with collision detection is a method for two or more devices to share transmission medium in half duplex communication. Half duplex is when two or more devices share a medium and they can only send or receive during a given period, not both. To paraphrase the 802.3 standard, when communication is done in full duplex (simultaneous transmit and receive) between two devices on a medium that supports full duplex communication without interference, there is no need for CSMA/CD. The majority of modern-day neworking scenarios are generally full duplex, however support for half duplex communications does exist in the delay device to meet the requirements of the 802.3 standard. This gives the device a broader range of support that can include legacy devices.

2.2.2 Physical Layer, Attachment and Auto Negotiation

The Physical layer is essentially the collection of hardware that handle the interpretation of data from the data link layer into real signals transmitted and received on the medium. Figure 2.2 shows the relation of the 802.3 standard to the OSI model. Clause 36 through 39 discuss many physical layer implementations. To maintain interoperability, the Medium Dependent Interface (MDI) working through a physical layer implementation has a Medium Independent Interface (MII) that can connect to the reconciliation layer. The delay device will only use the Gigabit Medium Independent Interface (GMII), specified in Clause 22, so only the GMII will be included in further discussion.

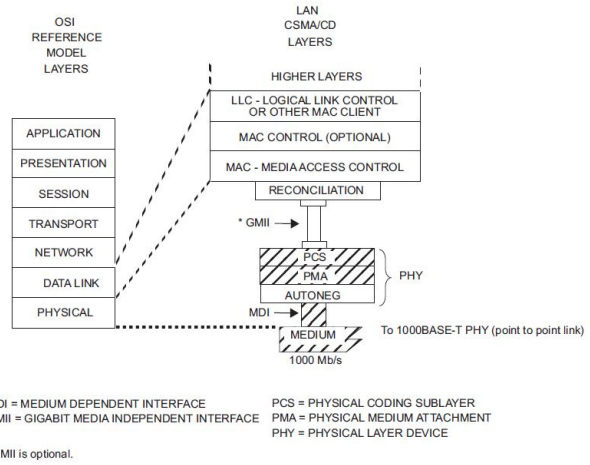

Figure 2.2: *GMII Relation to OSI/IEC reference model and IEE 802.3 CASMA/CD LAN model [6]*

The physical layer can be broken down into three basic parts: Medium Dependent Interface(MDI), Physical Medium Attachment (PMA) and Physical Coding Sublayer (PCS), these are defined by clauses 36-40. There are four versions of the physical layer for Gigabit communication, two cover copper media (1000BASE-CX clause 39 and 10000BASE-T clause 39) and the other two cover fiber optic media. Only copper will be discussed to limit the scope to relevant topics.

The functionality of the physical layer can be extended by adding support for auto-negotiation. Once media is attached to both devices in a network, auto-negotiation allows the devices to advertise their type of communications support and decide on which type of communications should be used, such as half/full duplex, master/slave relation. Clause 40.5.1.1 defines a set of registers that all 1000BASE-T auto-negotiation physical layers use. This allows the two link partners a mechanism through which to communicate and correctly negotiate link parameters without a dependence on a higher level layer. There are additional standards for the communication mechanism to setup the auto-negotiation registers on a physical layer. These registers also allow a user to enable and disable specific features,

such a master/slave, as well as store data for the link partners' abilities so that appropriate features can be exploited in user level applications. The auto-negotiation feature works with the physical layer's PMA sublayer by sending it a specific set of messages, such as enable/disable and scan for carrier. The PMA handles transmission, reception, link status, physical layer control, and clock recovery for the physical layer. The PMA can work with the auto-negotiation logic to bridge the physical layer logic to the MDI.

The final sublayer of the physical layer to discuss is the PCS. The PCS simply bridges the PMA service interface to the GMII/MII. It also provides the logic for enabling/disabling data transmission/reception and is responsible for carrier sensing and collision detection logic.

The GMII is an eight-bit wide interface to connect the PMA to the reconciliation sublayer of the MAC layer.

2.2.3 Media Access and Logical Link Control

The data link layer of the OSI reference model is generally comprised of two pieces the Logical Link Control (LLC) and Media Access Control (MAC) sub layers. In a general sense, this layer handles the point-to-point addressing and provides the ability to check the information at every point for accuracy. IEEE 802.2 Clause 3 stats the required format of the MAC frame. First, on transmission, there are seven octets of preamble that allow the devices to synchronize. Next, a start frame delimiter sequence of "10101011" is sent. Then the destination and source addresses are sent which are 48 bits long each. Then the length/type field is sent. This field can very depending on the frame type. Next, the payload of the frame is transmitted, which can contain a zero pad if the amount of data is less then 46 bytes. Finally the frame check sequence is appended. The frame check sequence is cyclic redundancy check that is 32 bits in length. Figure 2.3 illustrates the frame format.

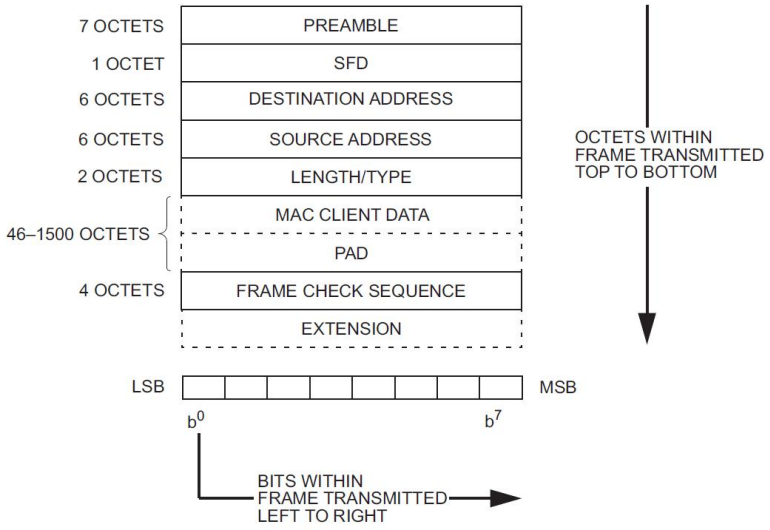

Figure 2.3: *MAC Frame Format [6]*

Signal Type	Required	I/O	Description
address	Yes	O	byte address to slave
waitrequest	Yes	I	forces master to wait for transfer
read	No	O	read request signal to slave dependency-readdata
write	No	O	write request signal to slave dependency-writedata
writedata	No	O	data to write to slave dependency-write
readdata	No	I	data to be read from slave dependency-read

Table 2.1: *Avalon Master Interface [1]*

2.3 Avalon Interface

The Avalon interface was developed to provide a easy-to-use communications standard between components in a FPGA project. [1] The Avalon bus standard is used in the delay device as the primary interface between components. There are six different types of Avalon interfaces (Memory-Mapped, Streaming, Interrupt, Memory-Mapped Tristate, Clock and Conduit), but the delay device only uses two of the six. Only the two types that are used in the project will be reviewed and they will be broken into four sections: Memory-Mapped Master, Memory-Mapped Slave, Streaming Sink and Streaming Source.

2.3.1 Avalon Memory Mapped Master Interface

The Avalon memory-mapped master interface was used by the Nios II processor in the system. Future work to the packet handler component will also include this interface type to communicate with different types of memory and memory controllers. The signals that a common component would use to master the memory controller or other memory-mapped slave device are shown in Table 2.1. This table also summarizes the required signals of the specification from section 3.8 of the Avalon manual [1].

The master interface timing specification is described in Figure 2.4. System on a Programmable Chip Builder (SOPC) automatically inserts timing latency logic between components with different latency specifications. However, it is the master component that must adhere to the wait signal. This allows for the implimentation of variable read and write latency as well as the creation of slave side arbitration.

2.3.2 Avalon Memory Mapped Slave Interface

Because the slave Avalon interface used for the packet hander custom peripheral was a memory mapped read/write interface, that set of signals are described in Table 2.2. In an Avalon memory mapped slave interface the user can choose if they want read, write or both read and write abilities. Only the set of signals for the intended interface version must be present, i.e. only write signals and no read signals for a read-only interface. These data signals can vary in width. Adapter logic between components of varing width can be

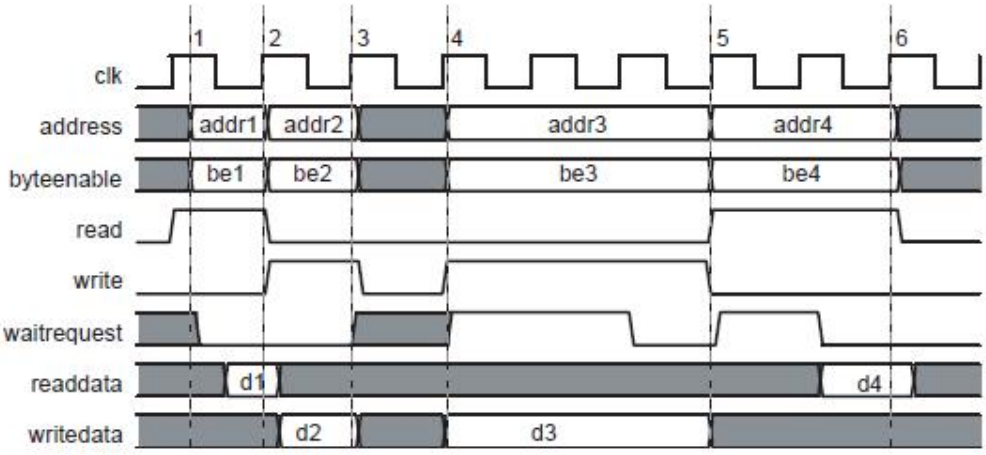

Figure 2.4: *Avalon MM Master Timing Diagram [1]*

Signal Type	Required Y/N	Description
address	Yes	byte address to slave
read	Yes	read request signal to slave dependency-readdata
write	Yes	write request signal to slave dependency-writedata
writedata	Yes	data to write to slave dependency-write
readdata	Yes	data to be read from slave dependency-read

Table 2.2: *Avalon Slave Interface [1]*

inserted in SOPC builder if a connection between to different data width components is desired. Each component of the delay device's system was implemented with 32-bit wide interfaces so data adapters would not have to be used. Memory mapped slave interfaces also can automatically generate slave-side arbitration if more then one master is connected to the device.

Other useful signals can also be implemented, however this was not necessary for the scope of this project and only added to design complexity. Useful signals to add to this interface in the future to make the interface more robust would be readdatavalid and writedatavalid. These signals are used for pipelined transfers that can have variable latency, which could be vary useful in systems where clock-domain crossing must occur due to timing issues or if burst transfers are desired. The packet handler custom component implements a read/write slave with fixed wait states as described in figure 2.5.

Because input and output data can be updated in the custom component each clock cycle and the address can be decoded with combinational logic, the master will have a read and write latency to the component of zero. This means that the data on the write data bus is latched on the same cycle that the write strobe is asserted. In a similar manner the data on the readdata bus becomes valid on the same cycle that the readdata strobe is asserted.

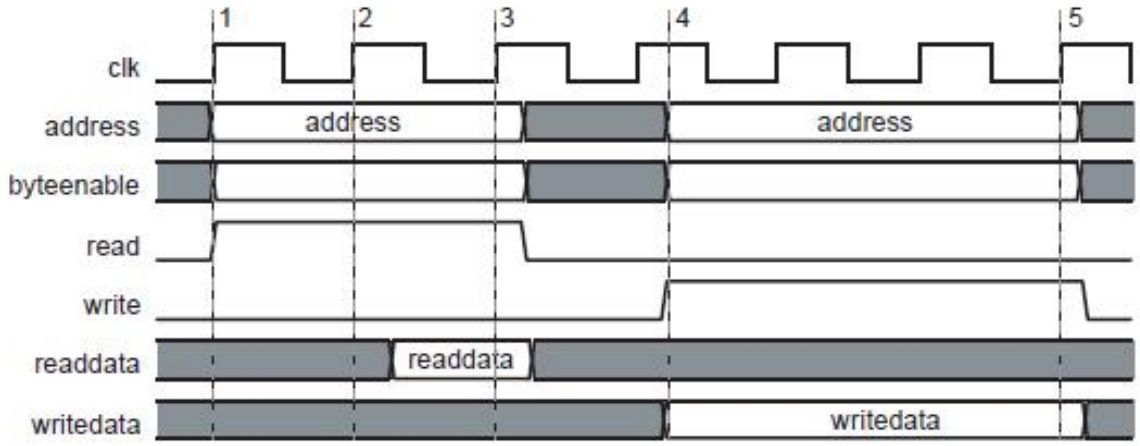

Figure 2.5: *Avalon MM Slave with Fixed Wait States-Timing Diagram [1]*

Signal Type	I/O	Description
data	I	data from source
empty	I	bytes in last transfer that were empty
endofpacket	I	last word of a packet
startofpacket	I	first word of a packet
error	I	if an error occurs it is encoded on this signal
ready	O	ready to recieve data signal
valid	I	data is valid signal

Table 2.3: *Avalon Streaming Sink Interface [1]*

In the packet handler's interface, data that is written to a read/write memory location will be updated on the following clock cycle. For example, if a user writes to a memory address at clock time 0 and simulatenously reads the data from that location it will not be valid. However, if the user writes data to a memory address at clock time 0 and then reads from the same address at clock time 1 the data will be valid.

2.3.3 Avalon Streaming Sink Interface

This section identifies the signals required at the sink for an Avalon sink-to-source transfer. Although none of the signals are strictly required by the standard, the signals described in 2.3 had to be implimented in order to correctly connect the Packet Handler to the Ethernet MAC.

The interface for both the source and sink implement a back pressure signal to/from the Rx and Tx FIFO of the Ethernet MAC. These interfaces were created with a read wait time of zero clock cycles, which means the data is valid on the interface once the source asserts

the valid bit and that data will remain valid until the clock cycle after ready bit is asserted. If the ready and valid bits both remain asserted, another transfer will take place on the next clock cycle. Implementing the interface in this manner is the best method because it offers the lowest complexity and the lowest latency. If an Avalon sink and source are connected together, and they have different read and write latency, SOPC builder automatically inserts timing adapter logic between the modules.

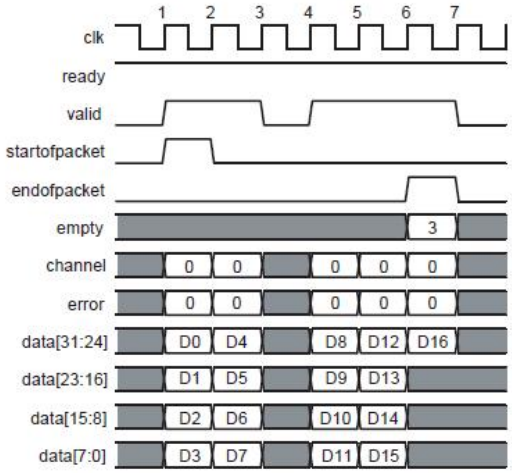

Figure 2.6: *Avalon Streaming Source/Sink Timing For Packet Transfer [1]*

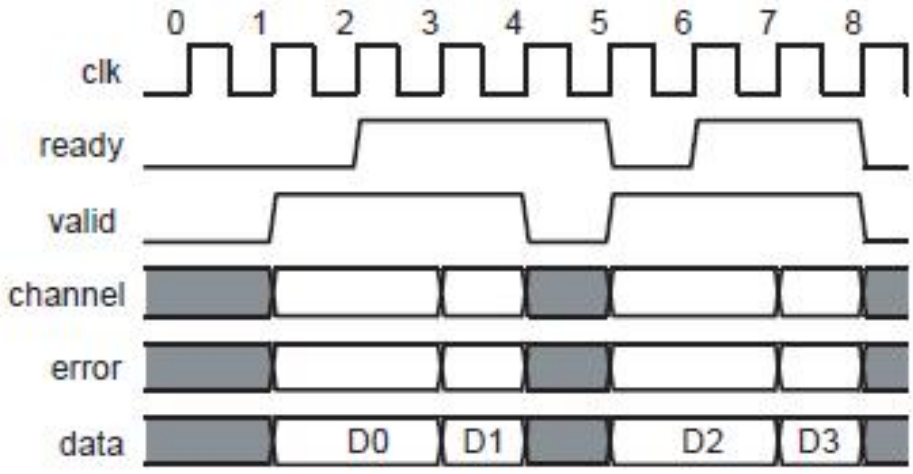

Figure 2.7: *Avalon Streaming Source/Sink Transfer with Back pressure [1]*

Signal Type	I/O	Description
data	O	data to sink
empty	O	fifo from source is empty
endofpacket	O	last word of a packet
startofpacket	O	first word of a packet
error	O	if an error occurs it is encoded on this signal
ready	I	input to allow a wait if sink is not ready
valid	I	data is valid signal

Table 2.4: *Avalon Streaming Source Interface [1]*

2.3.4 Avalon Streaming Source Interface

The Avalon streaming source interface is used to provide Altera Ethernet MAC sink interface with outgoing data. The timing diagrams shown in Figure 2.6 and Figure 2.7 also describe the streaming source's signal timing for a typical Avalon source interface.

2.4 Summary

In this chapter, we gave a summarized description of the OSI reference model and its ability to categorize network communications into different layers. Generally, a computer user only interacts with the application layer. However, the dependence of the application layer on other lower layers, specifically the data link layer, shows that simulated delay can and does occur in lower network layers. Next, we summarized the applicable parts of the 802.3 standard to bring the reader up-to-speed on the requirements for hardware and logic implimentation in Ethernet communications, and their relation to the OSI model. The 802.3 standard describes the necessary fields for transmitt and receive frames to to be exchanged. It also demonstrated the physical layer's auto-negotiation mechanism is not controlled directly by the data link layer, so additional setup is needed as the physical layer is not necessarily autonomous. Finally, we summarized the important portions of the Avalon bus communication standard, which is used in our device to communicate between on-chip components. The Avalon standard allows for easy connection between components in current and future systems.

Chapter 3

Delay Mechanics

In this chapter we discuss the cause and effects delay in a network. We also discuss the effect of delay on reliable vs. unreliable protocols to establish what types of results we should see when we test our system.

3.1 Delay

Generally there are four types of delay to worry about in a network:

1. Transmission Delay
2. Propagation Delay
3. Queuing Delay
4. Processing Delay

Transmission delay is dependent upon the amount of data and the rate at which the data was sent. This can be seen as the time that each device along the path takes to absorb the frame coming in off the medium. Although the frame size can be changed in some networks, it is generally a fixed maximum length.

The second, propagation delay, is a function of the length of the line and the speed of the medium on which the transmission is taking place. In networks that have nodes spaced extremely far apart this type of delay can be devastating to some protocols. Simulation of this type of delay is the simplest and would be the most beneficial for educational environments because many universities have networks that are connected for research.

The third type, queuing delay, is caused by the time it takes for a device to queue, service and transmitt a packet. It can be modified depending on what type of equipment is being used. Much work has been done in the computing community to speed up this stage of communications. One method is to use a cut-through mechanic which only reads the first and most necessary part of an incoming packet before forwarding it to the correct port where it is transmitted. Other methods include the ability to check traffic as it comes

through a switch or router and give certain protocols higher priority so they are transmitted before others. Due to wide variations in equipment types, this type of delay can be difficult or even impossible to simulate due to the unknown nature of multiple chains of equipment. Other difficulties of simulating at this level include the need to decrement TTL fields in certain protocols. Since the number of hops might be unknown in a multi path routing environment, it is not practical to try to simulate router delay specifically.

The fourth type of delay, processing delay, is assocated with the computation time of devices that use software to process packets. This type of delay can be a function of many variables including processing load, hardware speed and hardware service scheduling. It is worth mentioning though, because this type of delay can effect round trip times in protocols that use acknolodgement mechanisms such as TCP. This can also contribute to variations in the jitter and skew measurements during testing.

3.2 Throughput

Network throughput is a function of the link speed along the path from one node to another and is also a function of any link throttling and protocol throughput reduction mechanisms. Other factors, such as traffic load, can effect the throughput of a network. We will limit the discussion of throughput to the examination of a single Ethernet link.

3.2.1 Link Level

When sending frames on an Ethernet link, they are not transmitted back-to-back but rather, the 802.3 standard states that there must be 96 bit times worth of separation between frames. This is referred to as the Inter Packet Gap or IGP. This reduces the effective throughput of a medium running at a given frequency to less then simply simultaneous frame transmission at the link rate. The manual for Altera's Triple Speed Ethernet MAC cores, used in the delay device, states,The IEEE Standard specifies that frames must be separated by an inter packet gap (IPG) of at least 96 bit times. The MAC function, however, accepts frames that are separated by only 48 and 64 bit times in GMII (1000 Mbps operation) and MII (10/100 Mbps operation) respectively. The MAC function removes all preamble and SFD bytes from accepted frames. [7]"

This provides some flexibility to use hardware which does not strictly adhere to the 802.3 standard with the delay device. Lowering IGP on a link is done in an attempt to lower the transmission overhead and get more usable data through a link.

The IEEE 802.3 standard also provides a mechanism for flow control called pause frames. Pause frames allow a receiving node to notify the sender when the revive buffer is full or the reviver is backlogged. This provides a link-level throttling mechanism. When the system is generated link level flow control logic can be included as an option in the MAC layers. If the option is enabled when the hardware is generated the ability will then exist to turn on pause frame generation and acceptance in software. As stated in the 802.3 standard, pause frames are control frames and have an opcode of 0x0001. Pause frames also have a fixed

MAC address of 01-80-C2-00 which is reserved exclusively for pause frames. Pause frames carry data to tell the sender how long to wait before transmitting again. This time is called a quanta that varies from 0x0000 to 0xffff and each 0x1 quanta corresponds to 512 bit times of the currently connected link. The remainder of the frame's data section is padded with 42 bytes worth of zeros.

3.2.2 Protocol Level

Protocols can have a large effect on the throughput of the data that that they carry. Generally, they can be broken in to two categories, reliable and unreliable. Situations exist for both, which largely depend on the type of data being carried by the protocol. It is obvious that there are situations where an unreliable protocol is unacceptable, but for applications like streaming media unreliable protocols work much faster and offer better application-level quality. We will discuss the effects of delay on both types of protocols to establish an understanding for the expected results.

Reliable

In a common reliable system, such as TCP, the receiving device will send an acknowledgement back to the transmitting device once it has successfully recieved an incoming packet. In the simplest reliable protocol for every sent packet the sender (A) would wait for acknowledgement from the receiving device (B) [8]. In this system we will make the assumption that the line could function at the rate expressed in Equation 3.1. However, because the the device must wait for acknowledgement to send its next frame we must consider not only the line rate but the entire Round Trip Time (RTT). The RTT is expressed in Equation 3.3, this equation makes the assumption of zero processing time but displays the general concept needed to establish the underlying mechanic. Because the sender the must wait on the acknowledgement, the throughput of the connection will be effectively reduced to a rate expressed in Equation 3.4.

$$rate_{line} = \frac{1}{T_{frame}} \tag{3.1}$$

$$t_{AtoB} = T_{frame} + T_{prop} \tag{3.2}$$

$$t_{AtoBtoA} = T_{frame} + 2 \times T_{prop} + T_{ack} \tag{3.3}$$

$$rate_{reduced} = \frac{1}{T_{frame} + 2 \times T_{prop} + T_{ack}} \tag{3.4}$$

Reliable protocols usually operate at a reduced rate, which is the trade off for its checking mechanism. This shows that the throughput of a link is directly correlated with RTT in reliable wait-for-acknowledgement protocols. The expected result of increasing line delay via a delay device on this type of protocol is a reduction in throughput directly corresponding to that increase in line delay.

Unreliable

Unreliable protocols such as User Datagram Protocol (UDP), do not receive acknowledgement for sent data. If a packet is dropped during transmission, it is not accounted for and is not resent. Multimedia such as streaming video and web radio, are excellent uses for this type of protocol. UDP can offer higher throughput on a link because data can be continuously transmitted. However, the effects of this type of traffic can quickly find the limitations of systems because the protocol is not aware of congestion. If a new packet arrives once the recieve buffers are full, the device must drop the incomming packet to avoid buffer overflow. This mechanic will be useful to test the finite buffer limitations of the delay device by testing the available throughput without dropping packets at a certain delay.

3.3 Summary

To establish the expected results for use of the delay device, this chapter has identified the cause of network delays and discussed throughput at both link and protocol levels. We have established that the throughput of a reliable protocol, such as TCP, will be directly correlated with RTT. We have also established that we can test the available buffer capacity for a given delay in the device using UDP. That packet should get dropped in a UDP streamed transfer when the systems buffers are too full to handle the throughput. We also discussed the use of pause frames and their ability to control congestion at a link level.

This discussion nleads us to two important conclusions:

1. Wire transmission delays are the easiest and accurate types of delays to simulate. For this reason it is best to create delays at the link level.

2. Pause frames must be turned off during testing to establish the link's maximum throughput and buffer capacity

Chapter 4

Delay Device

The delay device was implemented on a flexible FPGA platform so other components could be added or removed as necessary. SOPC builder was chosen as the assembly method because of its ease-of-use for future delay system builders and modifiers. The heart of the delay device is the packet handler custom component. This custom component was created to delay traffic between any Avalon streaming sink and streaming source. This provides flexibility for future designs, allowing changes in MAC cores, and, even off-FPGA sources such as microcontrollers with FPGA hardware interfaces or non-Ethernet to Ethernet adaptation to be easily implemented and delayed.

4.1 System Overview

The system was assembled and generated in Altera's System On a Programmable Chip (SOPC) builder IDE. Each component was attached using the Avalon interface specification and all required hardware signals were exported to the top level so that pin outs could be assigned to the system via a block diagram file in Quartus II.

4.1.1 Hardware

The system was implemented on an Altera Stratix II GX PCI Express development board. This platform was chosen, because of its high speed grade and high density EP2SGX90FF1508C3NES 1508-pin FBGA package FPGA, as well as its numerous peripherals.

The board has two Small Form Pluggable (SFP) cages on it to accommodate two SFP transceiver modules. This allows the board to easily be adapted to either copper SFP modules operating at up to one 1.0Gb/s or Synchronous Optical NETworking (SONET) modules operating at typical speeds of 2.488Gb/s. Maximum transceiver speed is capable of up to 5Gb/s if specialized modules were available [5].

The development board also has several memory interfaces available. The 256 Megabyte DDR2 memory was used in the design. The DDR2 RAM was used by the processor for bulk storage. The DDR2 memory offers a large amount of storage and a very high bandwidth.

Error correction and detection logic is available, however it was not implemented to allow future versions of the packet handler custom peripheral greater adaptability to use DDR2 memory. Onchip memory was also included in the design as additional space for the soft processor's operating system. A minimal amount of onchip memory was created to keep resource utilization as low as possible.

Programming and debugging interfaces are implemented via a JTAG.

The design could easily be migrated to other FPGA devices as long as the FPGA and peripheral had enough performance and space (logical elements and memory blocks ect.) for target design.

The current hardware also provides room for expansion to a 10Gb/s format through High Speed Mezzanine Connectors.

Additionaly, useful interfaces exist for future expansion, such as a PCI Express bus for traffic sniffing and as a general debugging interface.

Two D-Link DGS-712 1000BASE-T Copper SFP transcievers are used to implement the physical layer of the device. They have the ability to operate in 10/100/1000BASE-T modes and have a Serial Gigabit Medium Independent Interface (SGMII), which fit the requirement for the development board. They are advertised as,"...compatible with the Gigabit Ethernet and 1000BASE-T standards as specified in IEEE 802.3z and 802.3ab. [9]" The the transceivers also support auto Medium Dependent Interface (MDI/MDIX) so that cross over and regular cables are handled interchangeably. This allows the system to be easily used between two computers or a computer and a hub or switch without changing cabling.

Figure 4.1: *D-Link DGS-712 [9]*

4.1.2 FPGA Core

The system consists of several components that each do a specialized job. There is a Nios II process that handles user input/output from the JTAG interface. The OS can change the packet handler delay time over the Avalon interface, and provides a mechanism to retrieve MAC statistics and debug utility. The MAC interfaces communicate with Altera ALT2GBX transceiver cores to send data out to the small form pluggable modules. Figure 4.2 shows the basic high-level connections between the components in SOPC Builder. They will be discussed in descending order as they appear in Figure 4.2.

Figure 4.2: *System High Level Connection*

- **sysid**
  Tells Nios II EDS what system it is connecting to on the FPGA to avoid attempting to run code that is intended for a different target system then the one that is currently programmed to the FPGA. The ID updates each time the system is regenerated so that there are not any version issues when programming two variations with of the same system name.

- **cpu**
  The Nios II is a scalable, 32 bit processor. It was added to the system to handle setup of the SFP modules, to set the necessary registers on the MAC cores and set the delay time of the packet handers. Because of its software base and JTAG communication ability, it allows an easy method for command line entry from the user to update the time delay without restarting the system. This mechanic also creates the ability to easily implement delay profiles by using one of the included timers and updating the delay on a periodic interval.

- **timer/timer1**
  The timers allow the proecssor delay an activity for a fixed imt interval and be interrupted when that activity completes. This can be useful for creating delay profiles.

- **ddr2mem**
  This is an instance of the DDR2 SDRAM High Performance Controller. This provides the logic, pin outs, and timing adjustment logic to communicate with the offchip, onboard DDR2 memory. This memory is used for bulk storage in the design that is shown in Figure 4.2 but could be mastered by other on chip components if they require bulk storage implementations. One such example is an additional variation of the Packet Handler custom component that uses offchip memory.

- **jtag_uart**
  The JTAG UART allows for the onchip system to communicate with the onboard JTAG control chip.

- **pll**
  Allows the clock rate to be reduced on the system.

- **PacketHandlerOnChip_inst and PacketHandlerOnChip_inst_1**
  Onchip version of the packet handler custom peripheral connects the Avalon source and sink ports of the two Ethernet Mac cores together. This component uses on chip memory to store the information while it is being delayed. This component is discussed further in the "'Packet Handler In Depth" section.

- **altera_ethernet and altera_ethernet_1**
  Altera tripple speed Ethernet cores provide the registers, fifo, and logic to implement the media access control layer of the system according to the 802.3 standard.

- **pio/pio_1/pio_2/pio_3**
  These are general purpose I/O pins that are exported to the top level block diagram. They are used to communicate with the SFP modules to set up and communicate with the their PCS. The communication is implemented as a two-wire serial bus for both SFP PCSs.

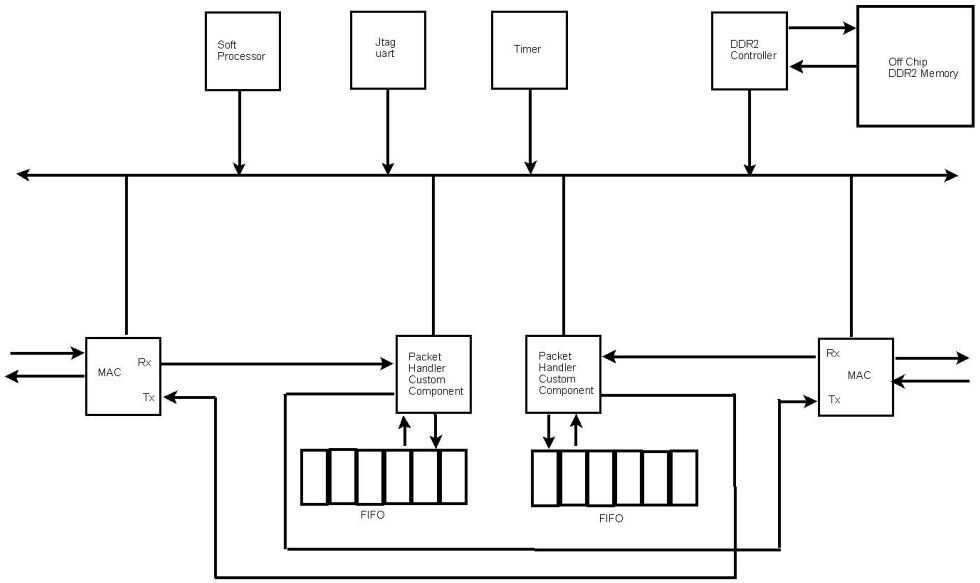

Figure 4.3: *System Block Diagram*

4.1.3 Software

The application is loaded into memory by the JTAG after the FPGA's hardware is programmed. This can be done by using a Nios II command window or by using the Nios II EDS.

Once the software is loaded and executed, the processor does the following series of operations:

1. Sets-up necessary registers in the SFP modules. Including auto-negotiation, link partner ablity etc.
2. Determines the link partner ability if link connection is active.
3. Resets MAC cores.
4. Programs the necessary MAC core parameters.
5. Finishes making the Ethernet interfaces active by enabling transmit and receive abilities in the MAC cores.

Module	CPU Instruction Master Address	CPU Data Maste Address
cpu.jtag_debug_module	0x20080800 - 0x20080fff	0x20080800 - 0x20080fff
onchiip_mem.s1	0x20040000 - 0x2005ffff	0x20040000 - 0x2005ffff
jtag_uart.avalon_jtag_slave	not connected	0x20081920 - 0x20081827
pll.s1	not connected	0x20081880 - 0x2008189f
pio.s1	not connected	0x200818e0 - 0x200818ef
pio_1.s1	not connected	0x200818f0 - 0x200818ff
pio2_.s1	not connected	0x20081800 - 0x2008190f
pio3_.s1	not connected	0x20081910 - 0x2008191f
altera_ethernet.control_port	not connected	0x20081000 - 0x200813ff
altera_ethernet_1.control_port	not connected	0x20081400 - 0x200817ff
timer.s1	not connected	0x200818a0 - 0x200818bf
ddr2_mem.s1	0x00000000 - 0x0fffffff	0x00000000 - 0x0fffffff
sysid.control_slave	not connected	0x20081928 - 0x2008192f
PacketHandlerOnChip_inst.slave	not connected	0x10000000 - 0x1000003f
PacketHandlerOnChip_inst_.slave	not connected	0x10000040- 0x1000007f

Table 4.1: *System Core Memory Map*

6. Resets packet handler component via soft reset.
7. Programs the necessary packet handler registers, such as initial delay time.
8. Waits for user input on the command line, to change the delay time.

The system has a memory map shown in Table 4.1

4.2 FPGA Resource Use

Resource use for logic elements is low for the available resources on the current FPGA. The resource use can change dramatically depending on the features the user decides to impliment. Namely, the size of the internal packet FIFO and the internal descriptor FIFO heavily determine the number of memory blocks that the design uses. This also determines the system's capacity and, in turn, its performance. Currently, on the Stratix II GX FPGA memory block usage is at approximately 94 percent for M4K blocks. Table 4.2 details the system's resource use. This system currently utilizes as much on-chip memory as possible to yield the best system performance.

With DDR2 in the system and not using the ECC bits on the controller the design uses 139 I/O pins. In order to communicate with the SGMII interface on the SFP modules two GXB receivers and two GXB transmitters were used.

Item	LC Combinational	LC Registers	Block Memory Bits
Slave Arbiter 0	1	0	0
Slave Arbiter 1	3	0	0
Altera Ethernet 0	2745	3148	154896
Altera Ethernet 0	2693	3126	154896
Clock 0	17	93	0
Clock 1	270	1152	0
Clock 2	18	1158	0
CPU	1070	900	46336
CPU Data Maser Arbiter	401	33	0
CPU Instruction Master Arbiter	137	2	0
CPU JTAG Debug Module Arbiter	27	3	0
DDR2 Memory Controller	1793	2227	52636
DDR2 Memory Arbiter	135	114	0
JTAG UART	110	107	1024
JTAG Avalon Slave Arbiter	2	0	0
OnChip Memory	1	0	1048576
OnChip Memory Arbiter	34	3	0
Packet Handler 0	557	298	1072640
Packet Handler 1	471	292	1072640
PIO 0	4	1	0
PIO 1	4	3	0
PIO 2	3	1	0
PIO 3	3	3	0
PLL	11	24	0
PLL Slave Arbiter	0	1	0
Timer	103	120	0
Timer1	109	120	0
Timer 1 Arbiter	5	0	0
Timer Slave 1 Arbiter	3	0	0
SLD Hub	78	82	0

Table 4.2: *Core Resource Use By Component*

4.3 Packet Handler

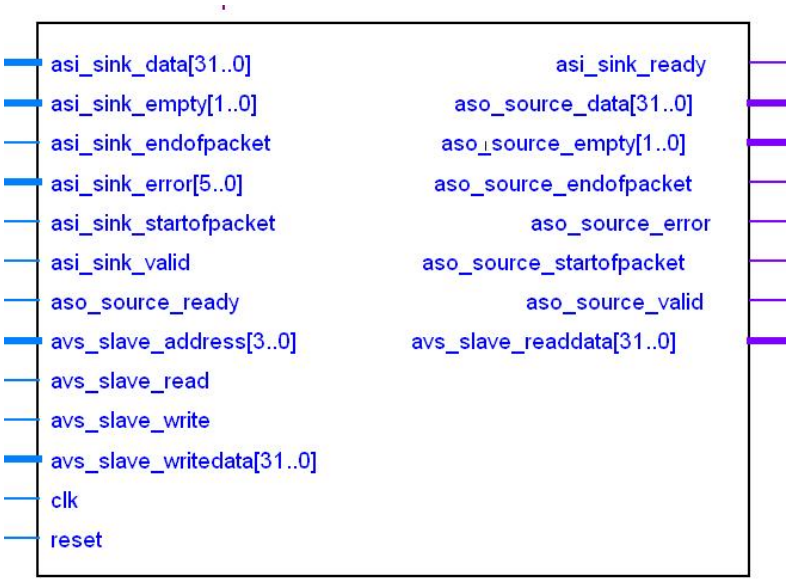

Figure 4.4: *Packet Handler Block Diagram*

There are two FIFOs in the system; one holds descriptors and the other holds the frame information. A descriptor consists of a time stamp, corresponding to the time that frame initially arrived, and the number of bytes in the frame. The descriptor is forty-eight bits wide, which allows for a thirty-two bit time stamp and sixteen bits to count the number of bytes in any given frame. This sixteen bit size allows for frames to be received up to 65,535 bytes. The largest frame received should only be 1,518 bytes for common frames without Q tags. It is also worth noting that the MAC cores in the current build of the system can support jumbo frames (9000 bytes), and this portion of the descriptor field would not have to be expanded to provide error free jumbo frame transport.

The component handles information in the following flow (see 2.7) for additional reference): The packet handler asserts the asi_sink_ready bit to tell the MAC core that is ready to start accepting data. Incoming information shows up on the sink data bus and the MAC core asserts asi_sink_startofpacket and asi_sink_vaild to start the transfer. Because the read wait latency of the packet handler is zero clock cycles, the data is latched and stored in the FIFO on the current clock cycle. A time stamp with added delay time is also latched off of the free running clock inside the packet handler instance and stored until the end of the packet arrives. The packet handler continues to take information as long as the asi_sink_vaild signal is asserted and continues to count the number of incoming bytes, which is four per clock cycle (32 bit wide interface) until the end of packet condition is reached. When asi_sink_valid and asi_sink_endofpacket are both asserted the final data for the packet is latched and the correct number of bytes for that cycle are added to the total number of bytes for the packet. This is calculated as four bytes minus the value on asi_sink_empty

during the end of packet clock cycle. The descriptor includes both the timestamp created at the start of the packet and the number of bytes that were in the transfer. Please note that the use of the word packet, in this instance, does not refer to a network packet, but rather the packet of data transfered over the Avalon bus between the source and sink. For this particular connection in the system, the information transfered in one Avalon packet could, and normally does, hold all the frame data, including destination address, source address and CRC32. The data that gets forwarded to the packet handler instance through this interface is a result of how the user decides to set the MAC registers.

Because the packet handler instance can delay any Avalon packet information system variations can be made to delay data between any Avalon sink and source. The system's MAC cores are currently set to forward the MAC source and destination addresses to the packet handler but not the CRC32. The MAC cores also currently remove the zero padding in the frame. This is done because the MAC cores to transmit frames are always zero padded and the MAC can not remove the zero pad on the input side and simultaneously keep the CRC32; there is an inherent dependency in the MAC core. The result of this does not modify the operation of the packet handler instance it only changes how software must set the MAC cores transmit registers. This also reduces the total amount of data stored in the packet handler FIFO's.

Packets continue to be received, given the condition that the packet data FIFO does not get within one packet (1,518 bytes) of full. If the packet data FIFO reaches this level the reception of data stops after any transfers that were in progress end. This ensures that no data is lost and the FIFO is not overflowed. Once this condition occurs the asi_sink_ready bit will stay low until space becomes available. This effectively back-pressures the MAC core's receive FIFO. If a frame was in the process of being received and there is enough room in the receive FIFO then the transmission continues. However, if there was not sufficient space, the frame is dropped.

Logic continually checks the empty flag of the descriptor FIFO. If the FIFO is not empty then the current FIFO is checked continuously unill a valid transmitt condition is found. The event of a clock wrap around is handled by making the transmission condition valid for all time in the time range, excluding the period from when the packet arrived to when the packet arrived plus its time delay. This works well because at a system clock rate of 83.3 MHz the 32 bit clock can have a total delay of about 51.7 seconds and delay times for the system are generally in the milliseconds. This also allows for the time delay for each packet to overlap in the system if necessary e.g. a packet arrives and has a time delay of 5ms, the user changes the time delay to 1ms delay and a second packet arrives 1ms later, the packet will have overlapping valid transmission times. Because the system is implemented as a FIFO, in the event that the second packet has a shorter required time delay than the first packet, the first packet will finish its time in the queue and be transmitted, then the second packet will be transmitted as soon as possible. The system was implemented in the fashion first, because it most closely mirrors the store forward delay found in most common switching equipment, which do not implement priority queuing and second, for simplicity.

Once a descriptor is found to be valid, the following series of events occur: First, the number of bytes to be transmitted is latched in a counter. Second, if the asi_source_ready

bit is asserted then transmission can begin to the MAC core, otherwise the packet handler is back pressured and must wait until the MAC core becomes ready. Once the asi_source_ready bit is asserted asi_source_startofpacket and asi_source_valid are asserted the data is latched by the MAC core and the byte count is decremented. The MAC can pause transfer at any time by deasserting the asi_source_ready bit. This process occurs until four bytes or less are left in the byte count. At that point, the asi_source_endofpacket bit is asserted and the asi_source_empty bits are calculated and set. The empty bits are simply calculated as four minus the number of bytes left on the last cycle.

The component also has an Avalon slave port so that the processor, or any Avalon memory mapped read/write master, can read and write information to the component. Write transactions are zero clocks in write latency and are handled by checking on each cycle to see if the avs_slave_address decodes to the register's address. If it does, the avs_slave_write signal is asserted the value shown on avs_slave_writedata is latched. Read transactions are decoded in a similar manner. If avs_slave_read is asserted then then the address of the correct register is decoded and presented on the avs_slave_readdata data bus. Table 4.3 describes the register map of of the packet handler component and the individual registers functions. A soft reset can be issued by writing a one to the zero bit of the control register. This reset condition is cleared on the next clock cycle. The address shown in table 4.3 is the decoded address as seen by the packet handler instance. The address that would be written to, in software, would be multiplied by four due to word-only addressing support by the packet handler component e.g. to write/read to addresses 0x0 one would write/read to the packet handlers base address plus zero and to write/read to address 0x1 one would write/read to the packet handlers base address plus four.

Figure 4.5 demonstrates the Packet Handler components Avalon interface signals and shows that they adhere to the Avalon specification. First, the simulation shows incoming packets being received from the Avalon Source. Next, the system waits for a delay of 50 clock cycles. Finally, the two packets are presented correctly to the Avalon Sink interface. The packets distinguished by the start-of-packet and end-of-packet signals. The simulation also demonstrates a pause during the first packet's transfer by driving the asi_sink_valid bit low for a period of time.

Figure 4.6 shows the state diagram for checking the output descriptors and sending a packet. The state machine is a Moore type state machine. The state machine sits in the idle state checks for a descriptor to become valid. Once the descriptor is found valid the state transitions to state_latch_desc, where it is removed from the queue and the number of bytes to be transmitted is latched into a register. The state machine then transition to the start-of-packet state where the start of packet sequence is generated. Next, the state machine transitions to the regular transfer state where the transfer continues in four byte increments. Finally, when less then four bytes are left the state machine transitions to the end-of-packet state which generates the correct sequence of signals finish the packet transfer. Table4.6 describes the state machine's transition conditions.

Register Name	Address	Read/Write	Function
Command	0x0	R/W	soft reset by writing to bit zero
NumInPFIFO	0x1	R/W	number of entries in the packet FIFO
TimeBase	0x2	Time delay before packet transmitt	
Unused	0x3	R/W	N/A
Unused	0x4	R/W	N/A
Unused	0x5	R/W	N/A
Unused	0x6	R/W	N/A
Unused	0x7	R/W	N/A
Unused	0x8	R/W	N/A
Unused	0x9	R/W	N/A
Unused	0xA	R/W	N/A
Unused	0xB	R/W	N/A
Unused	0xC	R/W	N/A
Unused	0xD	R/W	N/A
Unused	0xE	R/W	N/A
Unused	0xF	R/W	N/A

Table 4.3: *Packet Handler Control and Status Registers*

Item	LC Combinational	LC Registers	Block Memory Bits
Packet Data FIFO	204	74	1048576
Descriptor FIFO	34	54	24064
Packet Handler Total	557	298	1072640

Table 4.4: *Packet Handler Resource Use*

Item	Type	Port Depth	Port Width	Total Size
Descriptor FIFO	M4K	512	48	24576
Packet Data FIFO	Auto	32768	32	1048576

Table 4.5: *Packet Handler RAM Use*

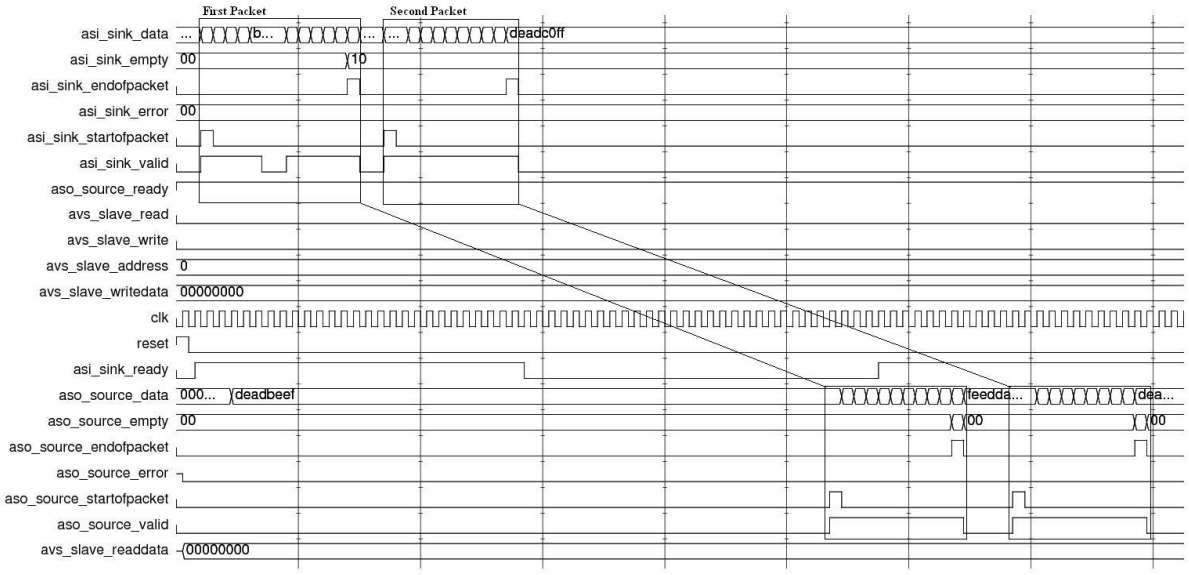

Figure 4.5: *Packet Handler Simulation*

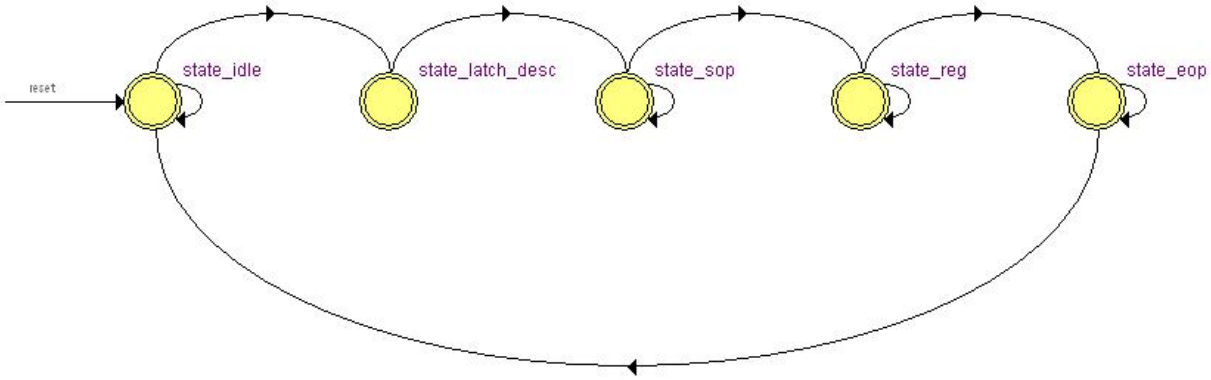

Figure 4.6: *Packet Handler Transmit FSM State Diagram*

Source State	Destination State	Transition Logic Summary
state_idle	state_idle	No descriptor or invalid descriptor time
state_idle	state_latch_desc	Valid descriptor is ready to be transmitted
state_latch_desc	state_sop	Always
state_sop	state_sop	Source not ready
state_sop	state_reg	Source is ready
state_reg	state_reg	Less then four bytes left to transfer
state_reg	state_eop	More then four bytes left to transfer
state_eop	state_eop	Source not ready
state_eop	state_idle	Source is ready

Table 4.6: *Transmit FSM Transition Table*

Chapter 5

Results

5.1 Theoretical Throughput

The theoretical throughput of the packet handler custom component is based on the clock and memory speed that was used to implement the component FIFOs. Currently, the system clock runs at 83.3Mhz. The component has a memory interface 32 bits wide, has a read/write latency of zero clocks and can accept data every clock cycle as long as the FIFOs are not full. This allows the component to offer a maximum theoretical throughput of up to approximately 2.65Gb/s at the currently implemented clock rate, ignoring any back pressure on the Avalon source output side. Throughput this high depends on setting a delay time low enough to keep the the FIFOs from filling up.

The entire system is limited to a theoretical throughput of 1Gb/s because that is the maximum rate at which the current SFP modules can send and receive data.

5.2 Device Connectivity and Use

The device can placed in any Ethernet 10/100/1000BASE-T network where there is a MTU of 1500 bytes or less.

5.2.1 Hardware

Figures 5.1 5.2 illustrate some typical scenarios for the device. If pause frames are required in the link a user must ensure that that the flow control option is included when compiling the design hardware and that pause frames are turned on in the command configuration register of the approperate MAC core/cores. Additionaly the user can also set a fixed pause frame quanta if desired. The default design does not have flow control enabled.

Figure 5.1: *Connection Scenario 1*

Figure 5.2: *Connection Scenario 2*

5.2.2 Software

The user can update the time delay of the system by typing in an integer number on the command line for the intended delay in nanoseconds. The console will then update the delay in both packet handler components and echo back the value read from both components to verify that they were correctly set. Since the delay is set symetrically, the total RTT will simply be twice the delay time entered. Figure 5.3 shows the console in action.

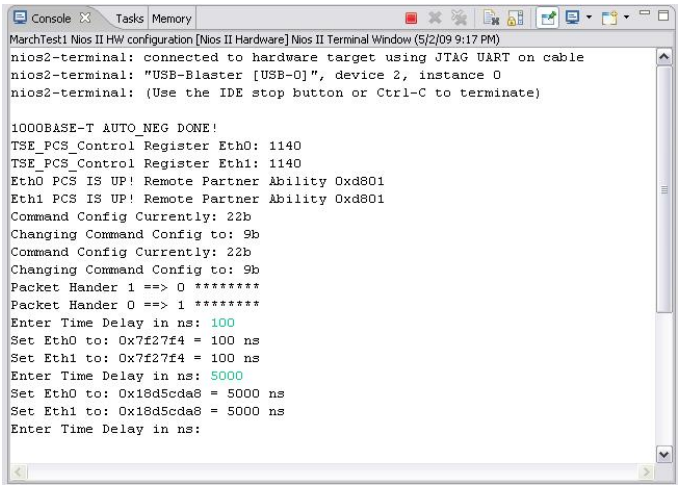

Figure 5.3: *Delay Device Console Application*

Additional functionality can be added by changing and recompiling the software using the set of API functions described in Table 5.1.

Function	Description
resetPacketHandler()	Soft Reset of Packet handler
setPacketHandler()	Set Packet Hander delay time
getPacketHandler()	Get Packet Hander delay time
linearRamp()	Linear time ramp

Table 5.1: *API Functions*

5.3 Delay Performance

The delay device can theoretically delay packets for over 50 seconds at the current clock rate, however, using delays that high would be very impractical in any setting. Most TCP traffic would time out at delays that high, despite valid data still being in transmission. Programming the device to have asymmetric delays along the transmit and receive paths is also possible by simply setting the delay time different for the two packet handler components.

A test bed was constructed to test the performace of the device. This test bed consisted of two Windows XP machines, with 1000BASE-T network cards, and a Netgear GS608-V2 10/100/1000 switch. Iperf was used to generate and recieve test traffic. Both common scenarios were tested and similar results were recieved during both configurations. Since both sets of results were similar only the results for scenario two (Figure 5.2 will be shown.

Figure 5.4 shows a screen shot of two separate pings between the computers to verify connection. The first ping was sent with the delay device set at zero delay and the second was with the device set a 25ms delay (50ms RTT).

Figure 5.5 and Figure 5.6 show the iperf TCP throughput graphs for 0ms and 50ms delay respectivly. As expected the throughput during high round trip times falls off significantly. In very low round trip times the throughput is very similar to the throughput achieved using only a switch between the computers as a benchmark (260Mb/s). With better testing equipment higher throughput may be achievable, however the test equipment that was used is very common to most computer labs and provides an excellent normal use scenario.

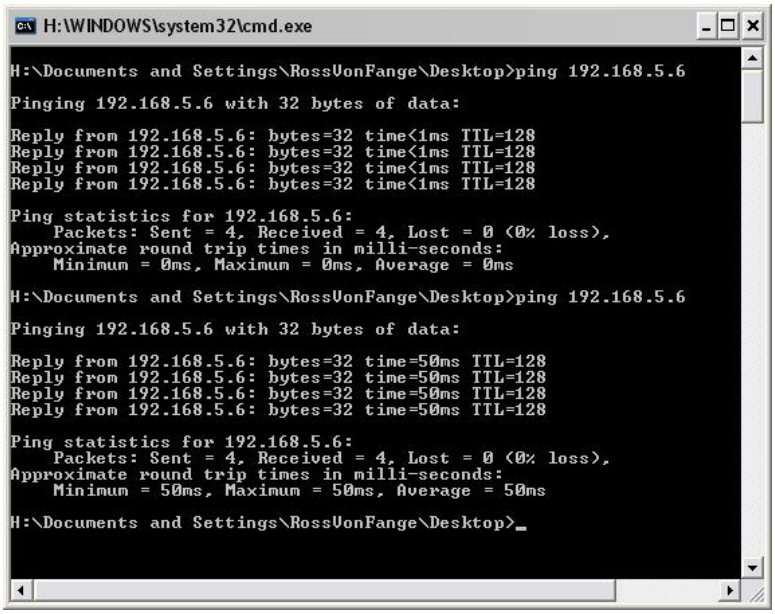

Figure 5.4: *Connection Scenario 2*

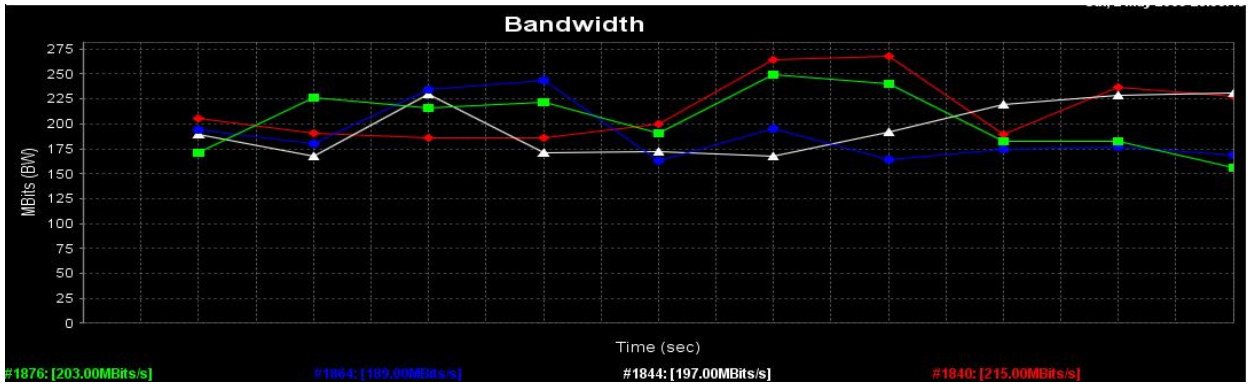

Figure 5.5: *Connection Scenario 2: Iperf TCP Throughput 0ms Delay*

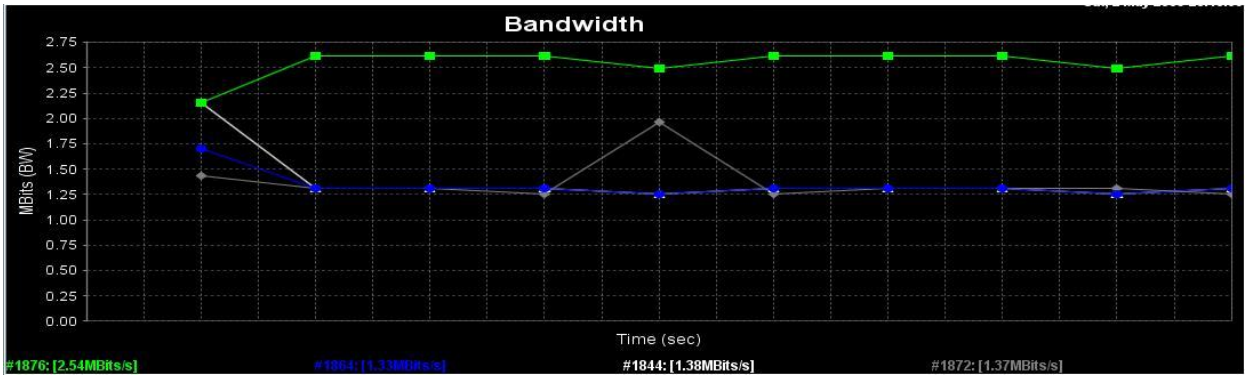

Figure 5.6: *Connection Scenario 2: Iperf TCP Throughput 25ms Delay*

Figure 5.7 and Figure 5.7 show the UDP bandwidth and jitter for the benchmark setup and scenario 2. UDP throughput for both cases was found nearly identical over several runs. Jitter measurements for both cases were also found to be very comparable. Iperf has an inherent UDP traffic generation limit of 50Mb/s so at moderate delays no buffer effects were experienced.

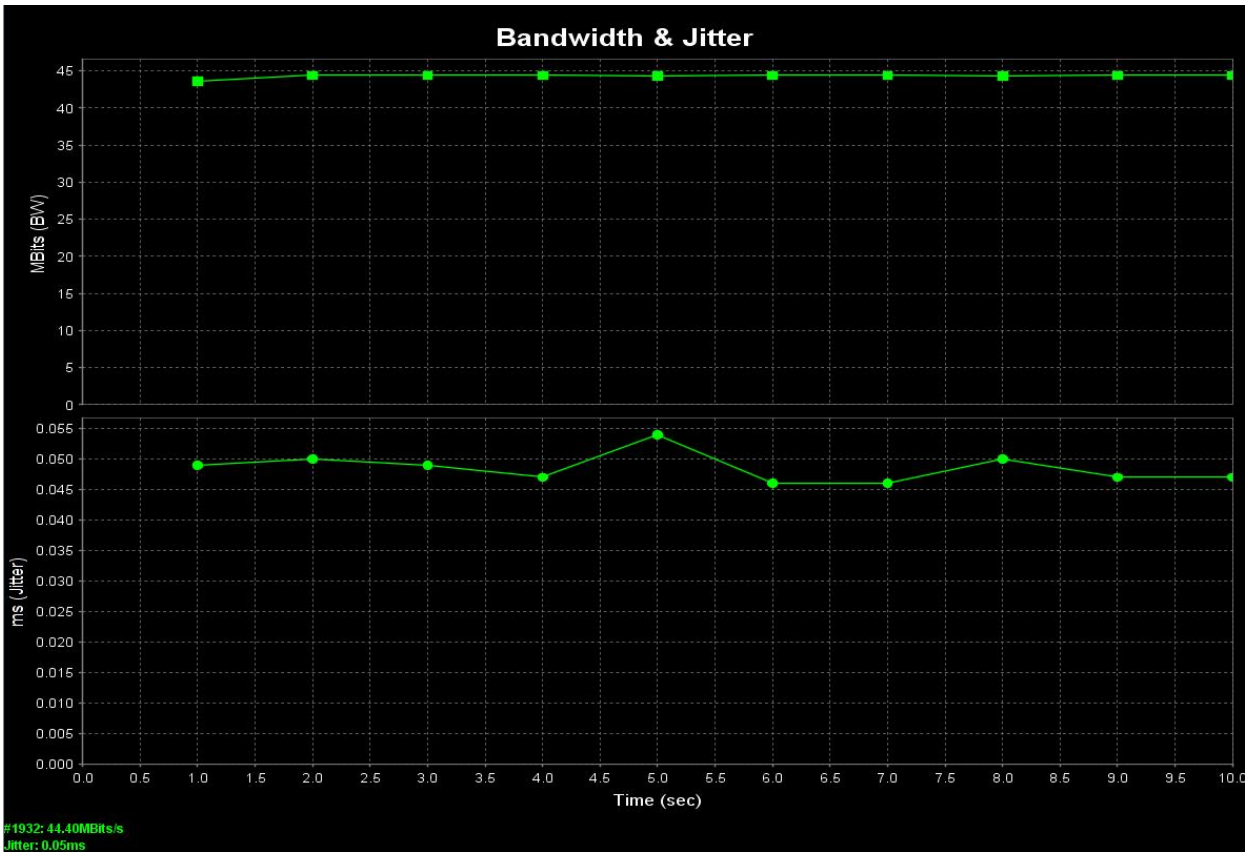

Figure 5.7: *Connection Benchmark: Iperf UDP Throughput with Only Switch*

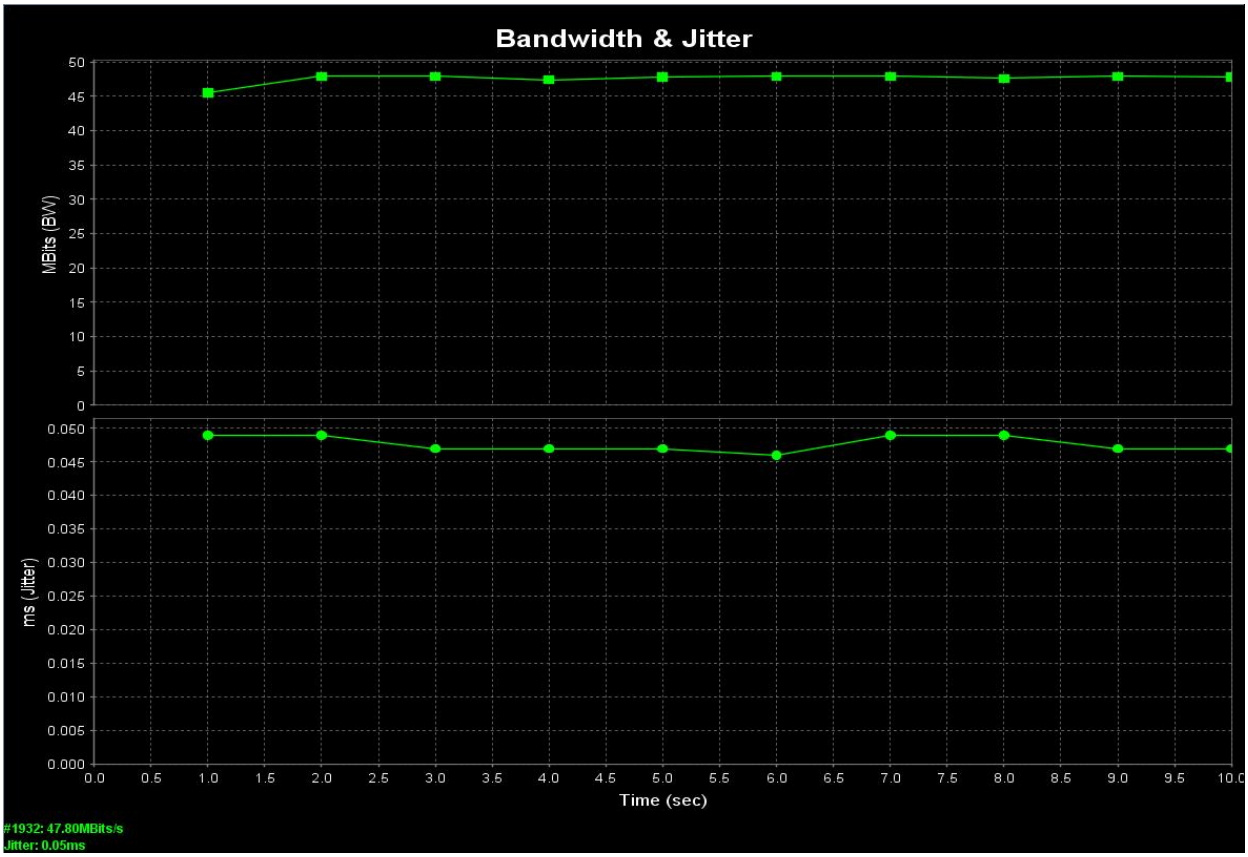

Figure 5.8: *Connection Scenario 2: Iperf UDP Throughput 20ms RTT*

Figure 5.9 shows RTT vs. Sequence number for a TCP flood generated by iperf on a Windows XP platform. The device was set to delay frames by 25ms, which generates a total RTT of 50ms plus computer processing time for the ack generation.

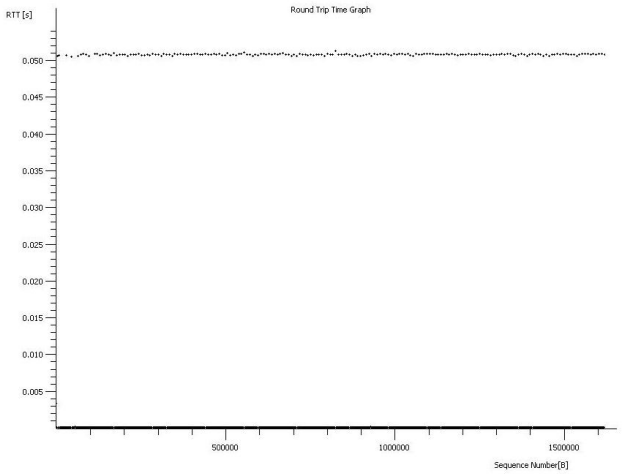

Figure 5.9: *50ms Delay RTT vs. Sequence Wireshark Capture*

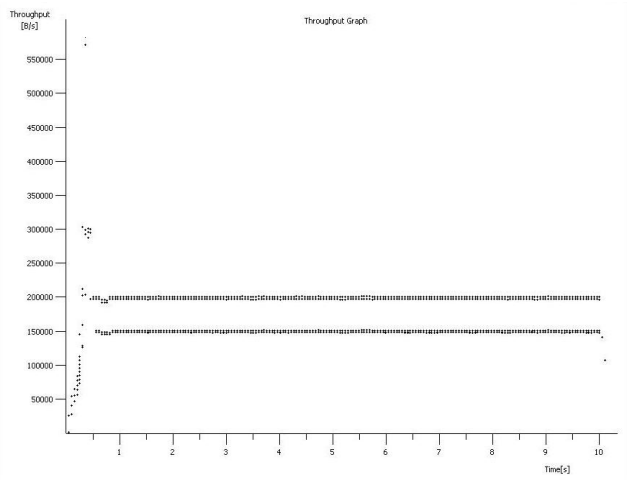

Figure 5.10: *50ms Delay Throughput vs. Time Wireshark Capture*

Figures 5.11 and figure 5.12 show the results from a simple delay profile that increases the delay by 300 clock cycles (clocked at 83MHz) every 255 processor clock cycles.

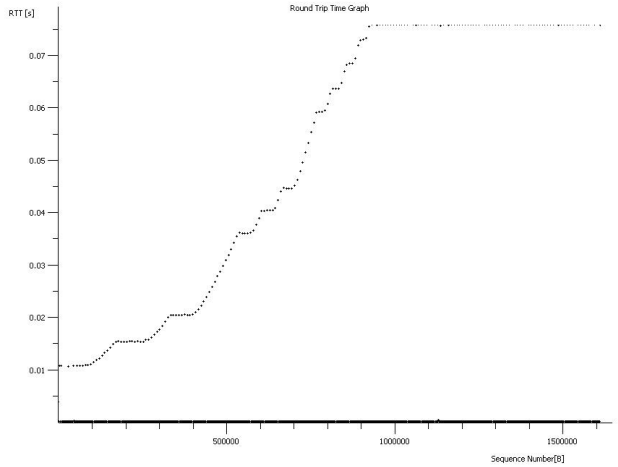

Figure 5.11: *Delay Profile RTT vs. Sequence Wireshark Capture*

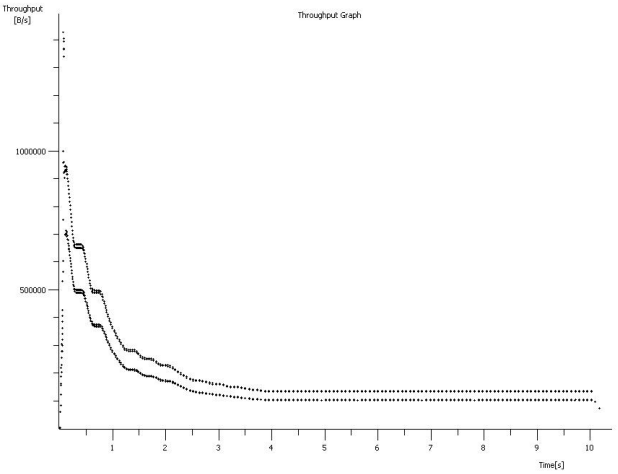

Figure 5.12: *50ms Delay Throughput vs. Time Wireshark Capture*

Figure 5.13 shows average throughput over ten second intervals using iperf with a TCP flood. It shows that as the RTT increase the throughput greatly decreases in the device. This is the expected result of delay on TCP traffic.

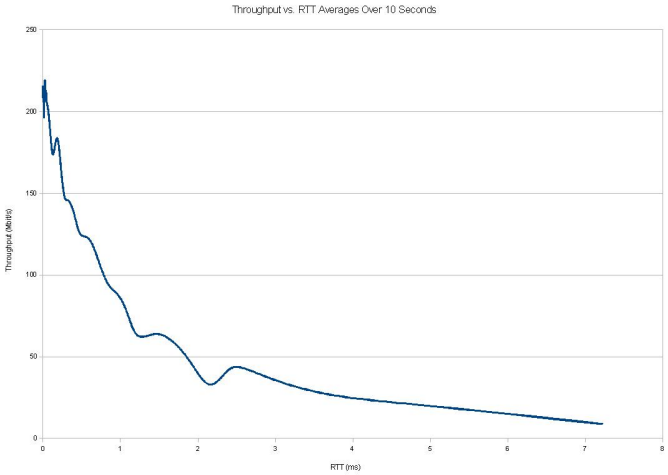

Figure 5.13: *50ms Delay Throughput vs. Time Wireshark Capture*

Chapter 6

Conclusion and Future Work

In conclusion, the delay device can successfully and accurately delay Ethernet frames with a range from as low as one packet store forward delay to theoretical maximum delay of 51 seconds. Trials showed that when using TCP transfers the performance suffers greatly at higher delays, which is expected because of TCP's mechanics. However, at reasonable delay sizes, similar to most delays users would try to replicate with this device, the throughput was high enough for most educational and research use and can be easily expanded by the addition of more memory.

We have contributed a design that is flexible, reproducable and inexpensive. The device also has a very high throughput and large delay time for its cost. Our design providess future users with an excellent starting point to base new variations from and provides a set of tools to ease the difficulty in creating new delay device variations. Future work should expand the portability and functionality of the design. To increase the device's performance dramatically, a master interface could be added to the packet handler component and logic could be adapted to make a FIFO of the DDR2 memory. Additionally the current development board has a third Ethernet physical layer that is a GMII interface which, coupled with the existing Nios II processor, could provide users a web based interface to change the time delay or activate delay profiles remotely. Migrating the current IP on the board toward open source cores, such as open source MAC cores would be beneficial in academic scenario for both cost and transparency. Finally, the addition of the ability to dynamically size the FIFOs in the packet handler component on-the-fly would allow users to easily test and assess the effects of buffer size in a store forward system.

Bibliography

[1] A. Corperation, *Avalon Interface Specifications*, 2008, Document Version: 1.1.

[2] B. A. Forouzan, *Data Communications and Networking 3rd Edition*, Elizabeth A. Jones, 2004.

[3] M. H. L. Kyung Chang Lee, Suk Lee, Worst case communication delay of real-time industrial switchd ethernet with multiple levels, in *Transactions On Industrial Electronics*, 3 Park Avenue, New York, NY 10016-5997,USA, 2006, IEEE, Vol. 53, No. 5 October 2006.

[4] S. N. Labs, Statment of work, FPGA Delay Device, 2007.

[5] A. Corperation, *Stratix II GX PCI Express Development Board Reference Manual*, 2007, Document Version: 1.0.1.

[6] T. I. of Electrical and I. Electronics Engineers, Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks - specific requirements: Part 3: Carrier sense multiple access with collision detection (csma/cd) access method and physical layer specifications, in *IEEE Std 802.3, 2005 Edition*, pages 1–1552, 3 Park Avenue, New York, NY 10016-5997,USA, 2005, IEEE, ISBN 0-7-381-2674-8 SS94892.

[7] A. Corperation, *Triple Speed Ethernet MegaCore Function User Guide*, 2007, MAC Core Users Guide.

[8] W. Stallings, *High-Speed Networks and Internets Performance and Quality of Service Second Edition*, Alan R. Apt, 2002.

[9] I. D-Link Coperation/D-Link Systems, *D-Link DGS-712*, 2006.

Appendix A

Packet Hander Verilog Code

```
module PacketHandlerOnChip (
input  wire [31:0] asi_sink_data,//Avalon Streaming Sinc  RX Interface
input  wire [1:0]  asi_sink_empty,
input  wire        asi_sink_endofpacket,
input  wire [5:0]  asi_sink_error,  //i dont use this
output reg         asi_sink_ready,
input  wire        asi_sink_startofpacket,
input  wire        asi_sink_valid,
output wire [31:0] aso_source_data,//Avalon Streaming Source  TX Interface
output reg  [1:0]  aso_source_empty,
output reg         aso_source_endofpacket,
output reg         aso_source_error,
input  wire        aso_source_ready,
output reg         aso_source_startofpacket,
output reg         aso_source_valid,
input  wire [3:0]  avs_slave_address,//Avalon Slave
input  wire        avs_slave_read,
output reg  [31:0] avs_slave_readdata,
input  wire        avs_slave_write,
input  wire [31:0] avs_slave_writedata,
input  wire        clk,
input  wire        reset

);

parameter state_idle=3'h0;
parameter state_latch_desc=3'h1;
parameter state_sop=3'h2;
parameter state_reg=3'h3;
parameter state_eop=3'h4;

//wire S_IDLE;
wire S_LATCH_DESC ;
wire S_SOP;
wire S_REG;
wire S_EOP;
reg  [2:0 ] sub_state;
reg  [2:0 ] state;

reg  [31:0] current_time;
reg  [31:0] descriptor_time;
wire [47:0] fifo_peek_data;
reg  push_desc;
reg   pop_desc;
wire  popable;
wire  descriptor_empty;

wire [47:0] descriptor_out;
wire       wr_packet_fifo;
reg  rd_packet_fifo;
wire [10:0] used_packet_fifo; //outputs of fifo are registered
wire [47:0]  descriptor_in;
wire  descriptor_full;

reg  [14:0]  byte_count;

reg  [14:0]     byte_rx_count;
reg  [31:0] timestamp;

reg  [31:0] time_base;
wire  sh_reset;
reg  [31:0]  ph_control;

     parameter in_idle=1'b0,in_running=1'b1;
reg  xfer_state;
wire in_progress;
wire  almost_full_sig;


// Soft Reset, Hard Reset Wire
assign sh_reset = (reset | ph_control[0]) ? 1'b1 : 1'b0;

// soft reset bit and control register
// ------------------------------------------------------
always @ (posedge clk or posedge reset)
 begin
if (reset)
ph_control <= 32'h0;
else if (avs_slave_write & avs_slave_address == 4'h0)
ph_control <= avs_slave_writedata;
else
ph_control <= ph_control & 32'hfffffffe; //clear out our reset bit ph_control[0]
 end

// Time Base Input
// ------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if (sh_reset)
time_base <= 50000;
else if ( avs_slave_write & avs_slave_address == 4'h2)
time_base <= avs_slave_writedata;
else
time_base <= time_base;
 end



// Output MUX of registers into readdata bus
// ------------------------------------------------------

always @ (avs_slave_read or avs_slave_address or ph_control or used_packet_fifo or time_base)
 begin
avs_slave_readdata = 32'h0;
if (avs_slave_read)
begin
case (avs_slave_address)
4'h0: avs_slave_readdata = ph_control;
4'h1: avs_slave_readdata = {21'h0,used_packet_fifo};  //used packet fifo = 11 bits
4'h2: avs_slave_readdata = time_base;
4'h3: avs_slave_readdata = 32'h0;
4'h4: avs_slave_readdata = 32'h0;
4'h5: avs_slave_readdata = 32'h0;
4'h6: avs_slave_readdata = 32'h0;
4'h7: avs_slave_readdata = 32'h0;
4'h8: avs_slave_readdata = 32'h0;
4'h9: avs_slave_readdata = 32'h0;
default: avs_slave_readdata = 32'h0;
endcase
end
 end

assign descriptor_in = {1'b0,byte_rx_count, timestamp};
//assign S_IDLE = ( sub_state == state_idle ) ? 1'b1 : 1'b0;
assign S_LATCH_DESC = ( sub_state == state_latch_desc ) ? 1'b1 : 1'b0;
assign S_SOP = ( sub_state == state_sop ) ? 1'b1 : 1'b0;
assign S_REG = ( sub_state == state_reg ) ? 1'b1 : 1'b0;
assign S_EOP = ( sub_state == state_eop ) ? 1'b1 : 1'b0;
```

```
// unused stuff
// ------------------------------------------------------------
always @ (posedge clk)
 begin
aso_source_error <= 0;
end

    // Local Time
// ------------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if ( sh_reset )
current_time <= 1;
else
current_time <= current_time + 1;
 end

    // State Machine over ride for later use also handles reset
// ------------------------------------------------------------
always @ (posedge clk or posedge sh_reset )
 begin
if (sh_reset)
state <= state_idle;
else
state <= sub_state;
 end

assign popable =  (~descriptor_empty && (current_time >= descriptor_out[31:0] ||
(current_time  <  (descriptor_out[31:0] - time_base)))) ? 1'b1 : 1'b0;

always @ ( state or aso_source_ready or byte_count or popable)
 begin
case(state)
state_idle:
 begin
if ( popable )
sub_state <= state_latch_desc;
else
sub_state <= state_idle;
 end
state_latch_desc:
sub_state <= state_sop;
state_sop:
 begin
if (aso_source_ready)
sub_state <= state_reg;
else
sub_state <= state_sop;
 end
state_reg:
 begin
if (aso_source_ready & (byte_count > 4) )
sub_state <= state_reg;
else
sub_state <= state_eop;
 end
state_eop:
if ( aso_source_ready)
sub_state <= state_idle;
else
sub_state <= state_eop;
default:
sub_state <= state_idle;
 endcase
end

// POP Descriptor
// ------------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if ( sh_reset )
pop_desc <= 1'b0 ;
else if ( S_LATCH_DESC )
pop_desc <= 1'b1; //we grab the bytes in the bytes counter
else
pop_desc <= 1'b0;
 end

// Start of Packet
// ------------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if ( sh_reset )
aso_source_startofpacket <= 1'b0;
else if ( S_SOP )
aso_source_startofpacket <= 1'b1;
else
aso_source_startofpacket <= 1'b0;
 end

// End of Packet
// ------------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if ( sh_reset )
aso_source_endofpacket <= 1'b0;
else if ( S_EOP )
aso_source_endofpacket <= 1'b1;
else
aso_source_endofpacket <= 1'b0;
 end

// Modulus
// ------------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if ( sh_reset )
aso_source_empty <= 2'b00;
else if ( S_EOP )
case ( byte_count[1:0] )
2'b00: aso_source_empty <= 2'b00;
2'b01: aso_source_empty <= 2'b11;
2'b10: aso_source_empty <= 2'b10;
2'b11: aso_source_empty <= 2'b01;
endcase
else
aso_source_empty <= 2'b00;
 end

// Byte Out Counter
// ------------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
begin
if (sh_reset)
 begin
byte_count <= 0;
rd_packet_fifo <= 1'b0;
 end
else if (S_LATCH_DESC)
```

```
 begin
byte_count <= descriptor_out[46:32];
rd_packet_fifo <= 1'b0;
 end
else if ( (S_REG | S_SOP) & aso_source_ready)
 begin
byte_count <= byte_count - 15'h4;
rd_packet_fifo <= 1'b1;
 end
else if ( S_EOP  &  aso_source_ready)
 begin
byte_count <= 15'h0;    // were done so remove the rest of the bytes
rd_packet_fifo <= 1'b1;  // read last value in fifo data wont be
 end
else
 begin
rd_packet_fifo <= 1'b0;
byte_count <= byte_count;
 end
end




// Vaild bit
// ------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if (sh_reset )
aso_source_valid <= 1'b0;
else if ( (S_SOP  | S_REG | S_EOP) & aso_source_ready )
aso_source_valid <= 1'b1;
else
aso_source_valid <= 1'b0;
 end


    DescriptorFIFO DescriptorFIFO_inst (
.aclr ( sh_reset ),
.data ( descriptor_in ),
.rdclk ( clk ),
.rdreq ( pop_desc ),
.wrclk ( clk ),
.wrreq ( push_desc ),
.q ( descriptor_out ),
.rdempty ( descriptor_empty ),
.wrfull (descriptor_full ),
.wrusedw ( descriptor_used )
);
wire descriptor_used;
wire descriptor_almostfull;

assign descriptor_almostfull = (511 - descriptor_used > 1) ? 1'b0 : 1'b1;



PacketFIFO PacketFIFO_inst (
.clock ( clk ),
.data ( asi_sink_data ),
.rdreq ( rd_packet_fifo ),
.sclr ( sh_reset ),
.wrreq ( wr_packet_fifo ),
.empty (),
.full ( ),
.q ( aso_source_data ),
.usedw ( used_packet_fifo ),
.almost_full (almost_full_sig )
);


always @ (posedge clk or posedge sh_reset)
 begin
if (sh_reset)
timestamp <= 0;
else if ( asi_sink_startofpacket )
timestamp <= current_time + time_base;
else
timestamp <= timestamp;
 end



// count the bytes and write them to the fifo as we go if we hit eop write the descriptor
// ------------------------------------------------------
always @ (posedge clk or posedge sh_reset)
 begin
if (sh_reset)
begin
byte_rx_count = 15'h4; // here if the fifo is too full rx_rdy go low and drop the packet so dont count it
push_desc = 0;
end
else if (asi_sink_startofpacket & asi_sink_valid  & asi_sink_ready)// & used_packet_fifo <= (65536 - 376 ))
//at he start of every packet we latch the first four bytes
begin
byte_rx_count = 15'h4;
push_desc = 0;
end
else if ( asi_sink_valid & asi_sink_endofpacket & asi_sink_empty == 2'b01  & asi_sink_ready)
begin
byte_rx_count = byte_rx_count + 15'h3;
push_desc = 1;
end
else if ( asi_sink_valid & asi_sink_endofpacket & asi_sink_empty == 2'b10  & asi_sink_ready)
begin
byte_rx_count = byte_rx_count + 15'h2;
push_desc = 1;
end
else if ( asi_sink_valid & asi_sink_endofpacket & asi_sink_empty == 2'b11  & asi_sink_ready)
begin
byte_rx_count = byte_rx_count + 15'h1;
push_desc = 1;
end
else if ( asi_sink_valid & asi_sink_endofpacket & asi_sink_empty == 2'b00  & asi_sink_ready)
begin
byte_rx_count = byte_rx_count + 15'h4;
push_desc = 1;
end
else if ( asi_sink_valid & asi_sink_ready & in_progress)
begin
byte_rx_count = byte_rx_count + 15'h4;
push_desc = 0;
end
else
begin
byte_rx_count = byte_rx_count;
push_desc = 0;
end
 end

assign wr_packet_fifo = (asi_sink_valid & asi_sink_ready & (asi_sink_startofpacket |
```

```
asi_sink_endofpacket | in_progress) ) ? 1'b1 : 1'b0;

always @ (posedge clk or posedge sh_reset)
 begin
if (sh_reset)
asi_sink_ready <= 1'b0;
else if (~in_progress & (almost_full_sig | descriptor_almostfull )) //if were not running and the fifo is full then
asi_sink_ready <= 1'b0;    // we cant hold another whole packet here so were going to have to start dropping
else
asi_sink_ready <= 1'b1;

 end

 assign in_progress = ( xfer_state==in_running ) ? 1'b1 : 1'b0;


 always @(posedge clk or posedge sh_reset)
  begin
if (sh_reset)
xfer_state <= in_idle;
else
case (xfer_state)
in_idle:
if (asi_sink_startofpacket & asi_sink_ready)
xfer_state <= in_running;
else
xfer_state <= in_idle;
in_running:
if (asi_sink_endofpacket & asi_sink_ready)
xfer_state <= in_idle;
else
xfer_state <= in_running;
default:
xfer_state <= in_idle;
endcase
 end


//Function for calculating the log2() for the input parameter memory depth
function log2;
input [31:0] value;
begin
for (log2=0; value>0; log2=log2+1)
begin
value = value>>1;
end
end
endfunction

endmodule
```

Appendix B

Nios II Soft Processor C Code

```
#include "system.h"
#include "hardware_def.h"
#include "sys/alt_stdio.h"
#include "stdio.h"
#include "tse_control.h"
#include "io.h"
#include "phapi.h"



#define DEBUGON 1

#ifdef DEBUGON
 #define DEBUG(args)(printf("Debug: "), printf args)
#endif

//void dumpPacketHandlerRegisters(unsigned int base_address)
//{
//    unsigned int watch;
//    watch = IORD_32DIRECT(base_address,PACKET_HANDLER_CONTROL);
//    printf("Control: 0x%x\n",watch);
//    watch = IORD_32DIRECT(base_address,PACKET_HANDLER_STATUS);
//    printf("Status: 0x%x\n",watch);
//    watch = IORD_32DIRECT(base_address,PACKET_HANDLER_INFO);
//    printf("Info: 0x%x\n",watch);
//    watch = IORD_32DIRECT(base_address,PACKET_HANDLER_BEEF);
//    printf("BEEF: 0x%x\n",watch);
//}

void dumpEthernetRegisters(unsigned int base_address)
{
    unsigned int watch;
    if(IORD_32DIRECT(base_address, TSE_CMD_CONFIG) & 0x8)
    {
        printf("Promiscious Mode: On\n");
        printf("Command Config Value %x\n",IORD_32DIRECT(base_address, TSE_CMD_CONFIG));
    }
    else
    {
        printf("Promiscious Mode: Off\n");
        printf("Command Config Value %x\n",IORD_32DIRECT(base_address, TSE_CMD_CONFIG));

    }
}

void dumpPacketHandlerRegisters(unsigned int base_address) {
    unsigned int temp;
    temp = IORD_32DIRECT(base_address, PACKET_HANDLEROC_USEDPACKETFIFO );
    printf("Used Packets: %x \n", temp);
}

void packetHandlerReset(unsigned int base_address)
{
    unsigned int temp;
    temp = IORD_32DIRECT(base_address,PACKET_HANDLEROC_CONTROL);
    temp = temp | 0x00000001;
    IOWR_32DIRECT(base_address, PACKET_HANDLEROC_CONTROL,temp);
}

void setTimeBase( unsigned int base_address, unsigned int timebase )
{
    unsigned int temp;
 //   printf("%x",base_address);
    temp = IORD_32DIRECT(base_address,PACKET_HANDLEROC_TIMEBASE );
 //   printf("Reseting Time Base - Was: %x changing to %x\n",temp,timebase);
    IOWR_32DIRECT(base_address, PACKET_HANDLEROC_TIMEBASE ,timebase);
}
unsigned int getTimeBase( unsigned int base_address)
{
    unsigned int temp;
 //   printf("%x",base_address);
    temp = IORD_32DIRECT(base_address,PACKET_HANDLEROC_TIMEBASE );
    return temp;
}

void init(void)
{
    unsigned int ETH0_macaddra_low  = 0x07541300;   // custom
    unsigned int ETH0_macaddra_high = 0x0000A4D5;
    unsigned int ETH1_macaddra_low  = 0x22334455;   //custom
    unsigned int ETH1_macaddra_high = 0x0000ee15;
    packetHandlerReset( PACKETHANDLERONCHIP_INST_1_BASE );
    packetHandlerReset( PACKETHANDLERONCHIP_INST_BASE );
    tseMacReset(TSE0_BASE);
    tseMacReset(TSE1_BASE);
    tseMacConfigure(TSE0_BASE,ETH0_macaddra_high, ETH0_macaddra_low);
    tseMacConfigure(TSE1_BASE,ETH1_macaddra_high, ETH1_macaddra_low);
    tseSfpConfigureLink((unsigned int)1000, 1 );
    tseMacPromisEnable(TSE0_BASE);
    tseMacPromisEnable(TSE1_BASE);

#ifdef DEBUGON
    printf("Packet Hander 1 ==> 0 ******** \n");
    printf("Packet Hander 0 ==> 1 ******** \n");
#endif
} //end init

int main(void)
{

    unsigned int i;
    unsigned int tb,temptb1, temptb2;
    init();
    setTimeBase(PACKETHANDLERONCHIP_INST_BASE,0x1fffff); //20ms delay
    setTimeBase(PACKETHANDLERONCHIP_INST_1_BASE,0x001fffff);
    tb = 0;
    while(1) {
        for( i=0; i < 0x0000FFFF; i++ ) {}  //delay a few clocks
        printf("Enter Time Delay in ns: ");
        scanf("%d",&tb);
        tb = tb*83;
        setTimeBase(PACKETHANDLERONCHIP_INST_BASE,tb);
        setTimeBase(PACKETHANDLERONCHIP_INST_1_BASE,tb);
        temptb1 = getTimeBase(PACKETHANDLERONCHIP_INST_BASE);
        temptb2 = getTimeBase(PACKETHANDLERONCHIP_INST_1_BASE);
        printf("Set Eth0 to: 0x%x = %d ns \n",temptb1,temptb1/83);
        printf("Set Eth1 to: 0x%x = %d ns \n",temptb2,temptb2/83);
    }
    return 0;
} //end main
```