

2009

## High speed radar signal collection and characterization on a custom reconfigurable platform

Francis D. Fradette  
*University of Dayton*

Follow this and additional works at: [https://ecommons.udayton.edu/graduate\\_theses](https://ecommons.udayton.edu/graduate_theses)

---

### Recommended Citation

Fradette, Francis D., "High speed radar signal collection and characterization on a custom reconfigurable platform" (2009). *Graduate Theses and Dissertations*. 2733.  
[https://ecommons.udayton.edu/graduate\\_theses/2733](https://ecommons.udayton.edu/graduate_theses/2733)

This Thesis is brought to you for free and open access by the Theses and Dissertations at eCommons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of eCommons. For more information, please contact [mschlangen1@udayton.edu](mailto:mschlangen1@udayton.edu), [ecommons@udayton.edu](mailto:ecommons@udayton.edu).

HIGH SPEED RADAR SIGNAL COLLECTION AND  
CHARACTERIZATION  
ON A CUSTOM RECONFIGURABLE PLATFORM

A Thesis

Submitted to  
the Engineering School of the  
UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for  
The Degree  
Master of Science in Electrical Engineering

by

Francis D. Fradette, B.E.E  
UNIVERSITY OF DAYTON

Dayton, Ohio


December 2009

HIGH SPEED RADAR SIGNAL COLLECTION AND  
CHARACTERIZATION ON A CUSTOM RECONFIGURABLE  
PLATFORM

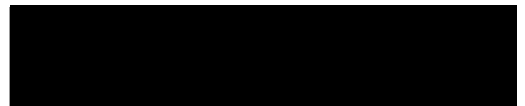
APPROVED BY:



Eric Balster, Ph.D.  
Adviser Committee Chairman  
Electrical & Computer Engineering



Frank A. Scarpino, Ph.D.  
Committee Member  
Electrical & Computer Engineering



Ralph Barrera, D.E.  
Committee Member  
Electrical & Computer Engineering



Malcolm Daniels, Ph.D.  
Associate Dean, School of Engineering



Tony E. Saliba, Ph.D.  
Dean, School of Engineering

## ABSTRACT

### HIGH SPEED RADAR SIGNAL COLLECTION AND CHARACTERIZATION ON A CUSTOM RECONFIGURABLE PLATFORM

Name: Francis D. Fradette  
University of Dayton, 2009

Advisor: Eric Balster

In this thesis, a high speed parallel data collection, processing and communications systems is developed. The input into the system consists of eight channels and is sampled at 50MHz. The system collects information from the eight channels of data and packages the data for transmission through a gigabit ethernet connection to a standard desktop PC computer.

A Xilinx based design is used as the platform. It has been chosen for its reconfigurability and ease of design upgrades. The proposed platform contains a Xilinx VirtexII Pro FPGA. Specific design challenges of the proposed system include: High speed data path integrity, high-speed data processing, power plane integrity, and the utilization of gigabit ethernet.

Subsystems of the proposed platform are designed to accommodate the 50MHz rate of incoming sensor data. Noise filtering of the data path and the power planes is also addressed. A design to preprocess the information from the sensors is implemented in VHDL on the FPGA. This design utilizes the embedded PowerPC on the

VirtexII FPGA to interface to gigabit ethernet. Embedded software is developed to collect the sensor data as well as manage an ethernet connection, which connects the Xilinx processing core to a desktop computer

To My Friends and Loved Ones ..  
...who, after three years, can no longer ask how my thesis is going.  
What now?

## ACKNOWLEDGMENTS

I would like to thank all the understanding people who supported me during the writing of this thesis. Thank you all for putting up with me. I would like to especially thank the following people.

- **Dr. Frank Scarpino, PhD:**

I would like to thank you for advice in my work in my life throughout the years. Thank you for giving me the opportunity to advance my education.

- **Dr. Eric Balster, PhD:**

I would like to thank you for working with me week after week on this thesis.

- **James Hayes:**

I would like to thank you for laying the groundwork for this entire design.

- **Dave Mundy, Mike Flaherty, Dave Lucking, and Chris McGuinness:**

I would like to thank you for your editorial input on this work. Without your help, some of this thesis would not have made sense at all.

- **Kerry Hill and Al Scarpelli:**

I would like to thank you both and everyone at the Air Force Research Laboratories for your financial and resource support throughout the years at AFRL. Without your support, this project would have never been possible.

- **My Family:**

I would like to thank you for my foundation in life. It was your love and support that made me who I am today.

- **Ashleigh Algren:**

I would like to thank you for your unending patience and understanding. Your support and persistent optimism kept me moving forward. Even when I didn't believe in myself, you did.

# TABLE OF CONTENTS

	Page
Approval . . . . .	ii
Abstract . . . . .	iii
Dedication . . . . .	v
Acknowledgments . . . . .	vi
List of Tables . . . . .	x
List of Figures . . . . .	xi
Chapters:	
1. Introduction . . . . .	1
1.1 Target Use . . . . .	2
1.2 System Design . . . . .	2
1.2.1 Reconfigurable Hardware . . . . .	3
1.2.2 Hardware Platform . . . . .	5
1.2.3 Embedded Software . . . . .	6
1.2.4 External Software . . . . .	7
1.3 Innovative Contribution . . . . .	8
1.4 Thesis Organization . . . . .	8
2. Hardware Layout Descriptions and Considerations . . . . .	9
2.1 General Purpose Input and Output Ports . . . . .	9
2.1.1 Crosstalk . . . . .	9
2.1.2 Trace Spacing . . . . .	12
2.2 FPGA Switching Needs . . . . .	12
2.2.1 Design Requirements for Capacitance . . . . .	14
2.3 Frequency Distribution of Noise Suppression . . . . .	14



2.3.1	Bypass Capacitor . . . . .	16
2.3.2	External Influence of Bypass Capacitors Network . . . . .	19
2.3.3	Analysis of Bypass Network . . . . .	22
3.	Pulse Receptor Core . . . . .	25
3.1	Data Description . . . . .	25
3.1.1	Characteristics of Data to be Collected . . . . .	26
3.1.2	Global Time of Arrival Clock . . . . .	26
3.2	Amplitude Data Collection . . . . .	27
3.3	Characterization of Pulse . . . . .	28
3.3.1	Pulse Width . . . . .	28
3.3.2	Time of Arrival of the Pulse . . . . .	30
3.4	Channel Multiplexing . . . . .	31
3.5	Packet Forming . . . . .	34
3.6	Control Word Generation . . . . .	35
4.	Gigabit Ethernet Physical Layer Interface to the FPGA . . . . .	38
4.1	Layout Considerations . . . . .	39
4.2	Gigabit Data Path Interconnections . . . . .	40
4.2.1	Embedded System with GEMAC Core . . . . .	40
4.2.2	GMII to SGMII Bridge . . . . .	42
4.2.3	SGMII over RocketIO . . . . .	42
4.3	Gigabit System Timing Requirements . . . . .	43
4.4	Management Interface . . . . .	44
4.4.1	PHY Channel Configuration Register Settings . . . . .	45
4.4.2	SGMII Bridge Configuration Register Settings . . . . .	47
5.	Ethernet Software . . . . .	48
5.1	Ethernet Packet Structure . . . . .	48
5.1.1	Internet Protocol (IP) Packet Structure . . . . .	49
5.1.2	ARP . . . . .	50
5.1.3	ICMP . . . . .	51
5.2	Embedded Software . . . . .	52
5.2.1	Communication Path Initialization . . . . .	52
5.2.2	Ping Exchange . . . . .	53
5.2.3	Data Exchange . . . . .	54
5.3	Third-Party Software . . . . .	54
5.3.1	Windows XP Network Interface Software . . . . .	54
5.3.2	WireShark . . . . .	55

5.3.3	Ethernet Traffic . . . . .	55
5.4	General Purpose Computer Software . . . . .	55
5.4.1	Data Transfer . . . . .	56
6.	Results . . . . .	58
6.1	Pulse Receptor Core Usage . . . . .	58
6.1.1	Global Time of Arrival . . . . .	58
6.1.2	DREAM Machine Test Set . . . . .	59
6.1.3	Reception of Pulse . . . . .	60
6.1.4	Generation of Control Word . . . . .	62
6.2	Ethernet System Demonstration . . . . .	64
6.2.1	Ethernet Packet Handling . . . . .	65
6.2.2	ARP Packet Exchange . . . . .	66
6.2.3	Ping Exchange . . . . .	67
6.2.4	Ethernet Data Transfer from the DREAM board . . . . .	68
7.	Conclusions & Future Work . . . . .	73
7.1	Future Work . . . . .	74
Appendices:		
A.	. . . . .	76
B.	. . . . .	81
C.	. . . . .	103
D.	. . . . .	127
Bibliography . . . . .		130

## LIST OF TABLES

Table	Page
2.1 Comparison of Capacitor Usage in Reference Design and DREAM Design	15
2.2 Selection of Bypass Capacitors based on Xilinx Suggested Values[11]	15
2.3 Selection of Capacitor Values . . . . .	17
2.4 Parasitic Values of Bypass Capacitors . . . . .	18
3.1 Comparison of Capacitor Usage in Reference Design and DREAM Design	35
4.1 Default Representation of LED Indications . . . . .	46

## LIST OF FIGURES

Figure	Page
1.1 Overview of the Intended Use of the Design . . . . .	3
1.2 Dynamically Reconfigurable Experimental Application Module (DREAM) Outline . . . . .	4
1.3 Dynamically Reconfigurable Experimental Application Module (DREAM)	5
2.1 Radiation Intensity of Signal Conductors . . . . .	11
2.2 Frequency Analysis of a Step Function . . . . .	16
2.3 Equivalent Schematic of a Real Capacitor . . . . .	17
2.4 Frequency Response of $2.2\mu\text{F}$ Capacitor . . . . .	19
2.5 Frequency Response of $470\mu\text{F}$ Capacitor . . . . .	20
2.6 Frequency Response of $0.1\mu\text{F}$ Capacitor . . . . .	21
2.7 Frequency Response of $0.01\mu\text{F}$ Capacitor . . . . .	22
2.8 Bypass Capacitor Configuration . . . . .	23
2.9 Composite Frequency Impedance of Bypass Capacitors . . . . .	24
3.1 PRC Channel Block Diagram . . . . .	27
3.2 Global Time of Arrival Generator Block Diagram . . . . .	27
3.3 PRC Channel State Diagram . . . . .	30
3.4 PRC Channel Data Multiplexing Block Diagram . . . . .	32
3.5 PRC Channel Data Collection Simulation . . . . .	33
3.6 PRC Channel Data Multiplexing Simulation . . . . .	34
3.7 Control Word Generation Block Diagram . . . . .	36
3.8 Control Word Generation State Diagram . . . . .	37
4.1 Block Diagram of Gigabit Ethernet . . . . .	39
4.2 Management Control Register . . . . .	45
4.3 Management Control Register . . . . .	45
5.1 Structure of Ethernet Packet . . . . .	49
5.2 Structure of Internet Protocol Packet . . . . .	50
5.3 Structure of ARP Packet . . . . .	51
5.4 Structure of ICMP Echo Packet . . . . .	52
5.5 WireShark View of Ethernet Controller Traffic . . . . .	56
5.6 DREAM Board TestBed . . . . .	57
6.1 Global Time of Arrival Generator Experimental Measurement . . . . .	59

6.2	DREAM Machine Test Set . . . . .	60
6.3	DREAM Test Set with Channel 1 Set to a Pulse Width of AC . . . . .	61
6.4	HyperTerminal Display of DREAM Board Receiving a Pulse on Channel 1 with a Width of AC . . . . .	62
6.5	DREAM Test Set with Channel 1 Set to a Pulse Width of 78 . . . . .	63
6.6	HyperTerminal Display of DREAM Board Receiving a Pulse on Channel 1 with a Width of 78 . . . . .	64
6.7	HyperTerminal Interface for PRC Control Word Generation . . . . .	65
6.8	PRC Channel Data Collection Experimental Measurement . . . . .	66
6.9	HyperTerminal Display of DREAM Board Initializing . . . . .	67
6.10	HyperTerminal Display of DREAM Board Printing the Next Ethernet Packet Received . . . . .	68
6.11	HyperTerminal Display of DREAM Board Parsing an Ethernet Packet . . . . .	69
6.12	WireShark Display of General Purpose Computer and DREAM Board ARP Exchange . . . . .	70
6.13	Command Terminal Display of General Purpose Computer Executing a Ping Command . . . . .	70
6.14	WireShark Display of General Purpose Computer Executing a Ping Command . . . . .	71
6.15	HyperTerminal Display of DREAM Board Responding to Ping Packet . . . . .	71
6.16	DREAM TestBed After Receiving Data From DREAM Board . . . . .	72

# CHAPTER 1

## Introduction

Sensors provide a wealth of information in an array of different applications. However, information provided by sensors can generate large amounts of data. Depending on the application, much of this data may contain little or irrelevant information.

Specialized hardware can be designed to preprocess sensor data in real-time applications to reduce the volume of data. However, designing and fabricating customized hardware is very time consuming and expensive. Also, specialized circuitry that is developed is typically application specific.

An alternative is to use a general purpose computer. Standard desktop PC computers are often chosen as a platform for sensor data processing. The cost for an individual computer is relatively low and the uniformity of PC computers eases the overhead of learning a new system for data processing. However, since a standard desktop computer is a general purpose computer, it often is unfit for real-time data processing.

Field Programmable Gate Arrays(FPGA) are a very useful tool for sensor data processing. The reconfigurable nature of FPGAs enable experimentation with multiple hardware processing implementations which can be developed quickly without the need to fabricate multiple integrated circuits and circuit boards. An example of this

is seen in [7]. The flexibility of the FPGA can be utilized to continually update or change the processing implementation.

## **1.1 Target Use**

The design, developed in this thesis, is used to capture data from a system which generates eight channels of 8 bit data. The 8 bits are the output of an Analog to Digital Converter (ADC) representing the amplitude of a sensor detecting an RF pulse. The proposed design provides the following information about each channel of data: the amplitude of an RF pulse, the duration of the pulse, and the time the pulse occurred. A block diagram of the target use is seen in Figure 1.1. On the right of the Figure are the eight channels of RF sensor data.

The system then collects and characterizes the incoming sensor data. This data consists of a three bit channel identification, an eight bit amplitude, a sixteen bit pulse width, and a thirty-eight bit time of arrival. These signals are described in detail in Chapter 3. The RF pulse data is then formed into packets and transmitted via gigabit ethernet to a general purpose computer. This process is described in Chapter 4 and Chapter 5.

## **1.2 System Design**

The proposed system utilizes a board design known as the Dynamically Reconfigurable Experimental Application Module (DREAM) board, a specialized printed circuit board consisting of processing and communication elements. Figure 1.3 is an image of the DREAM board and the major components of the PC board layout

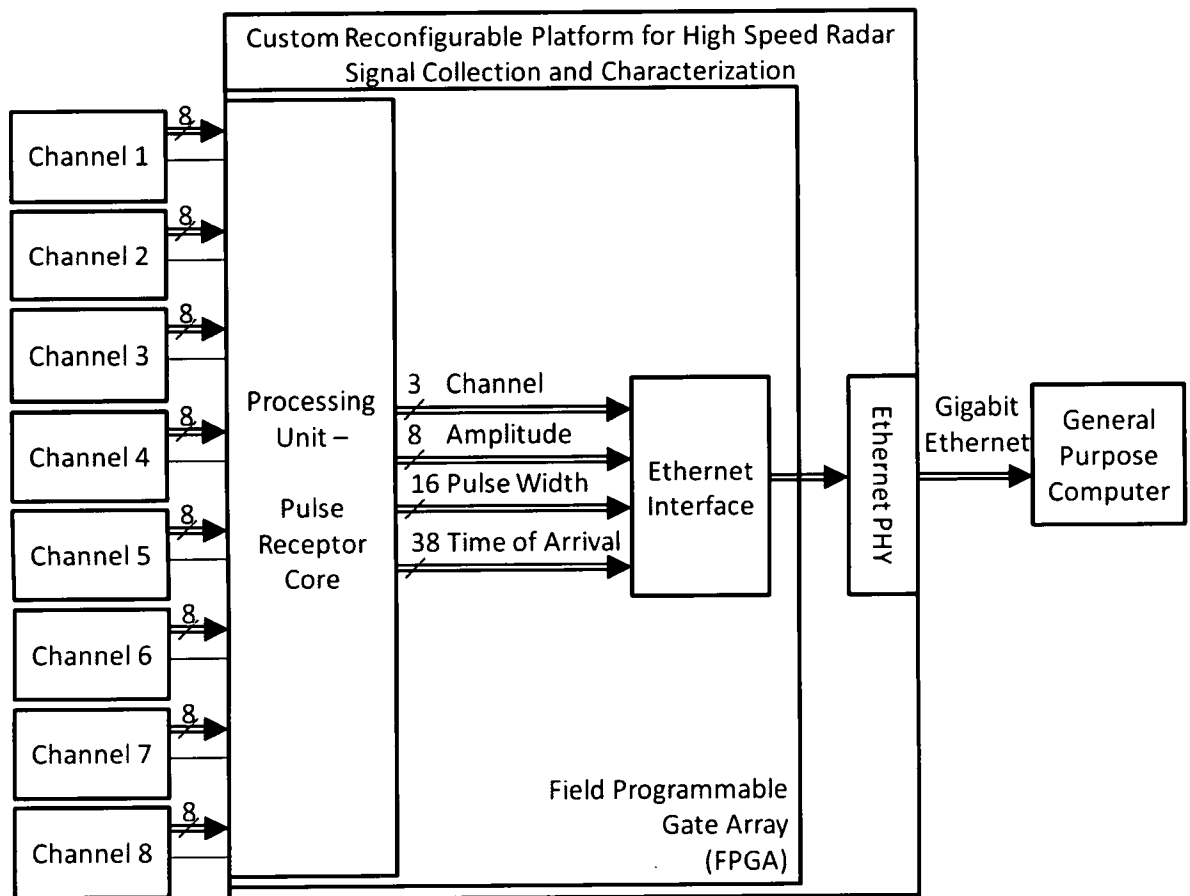


Figure 1.1: Overview of the Intended Use of the Design

are seen in Figure 1.2. The capabilities of the DREAM board are described in the following sections.

### 1.2.1 Reconfigurable Hardware

The FPGA on the DREAM Board is a VirtexII Pro made by Xilinx, seen on the left in Figure 1.2. The specific FPGA device, XCVP30, has eight RocketIO transceivers[12]. Four of these RocketIO connections are connected to an InfiniBand connection and the other four are connected to a gigabit PHY device, VSC8234,



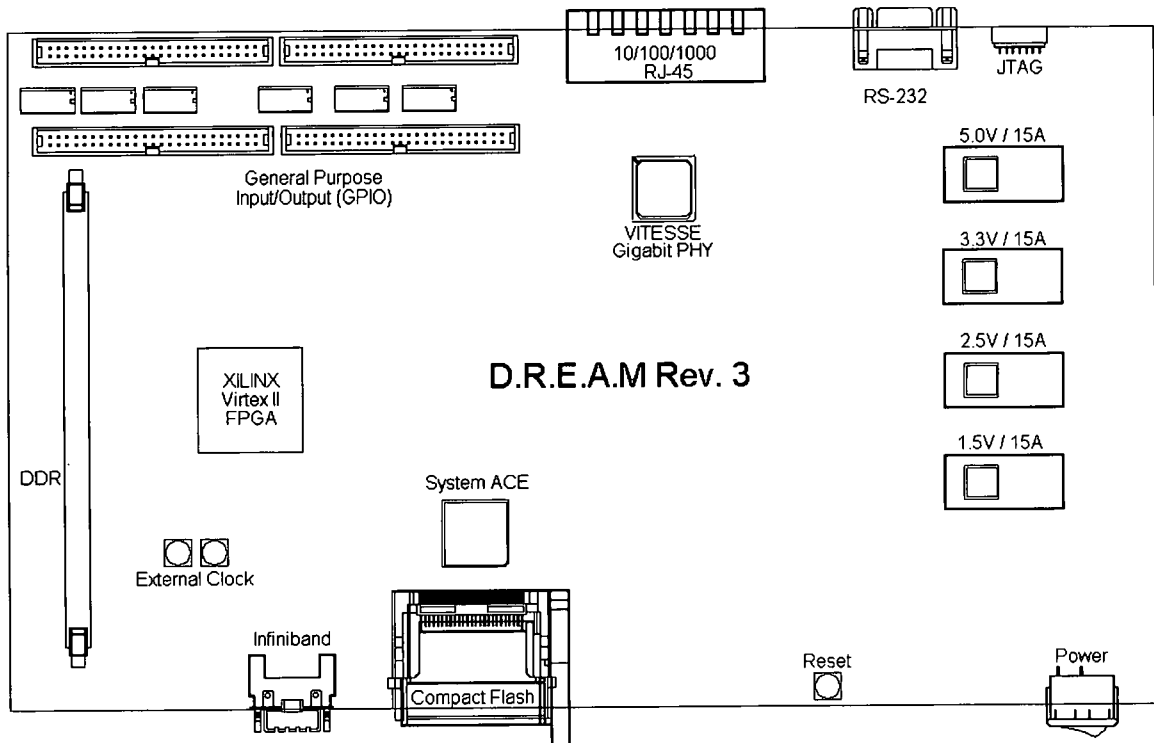


Figure 1.2: Dynamically Reconfigurable Experimental Application Module (DREAM) Outline

manufactured by Vitesse[10]. Both of these connections are described in Section 1.2.2. There are more than 30,000 logic cells available on the XCVP30, as well as two PowerPC blocks. One PowerPC block is used as the processor for embedded software communicating via ethernet to the desktop PC, which is described in section 1.2.3. This Virtex chip has 896 pins, 644 of which are user-definable input/output(I/O) connections[12].

A subset of the available I/O pins connect the large amounts of RF sensor data to the FPGA. It is in the reconfigurable logic cells that preprocessing of the sensor data occurs. The raw RF sensor data is captured and characterized and the information

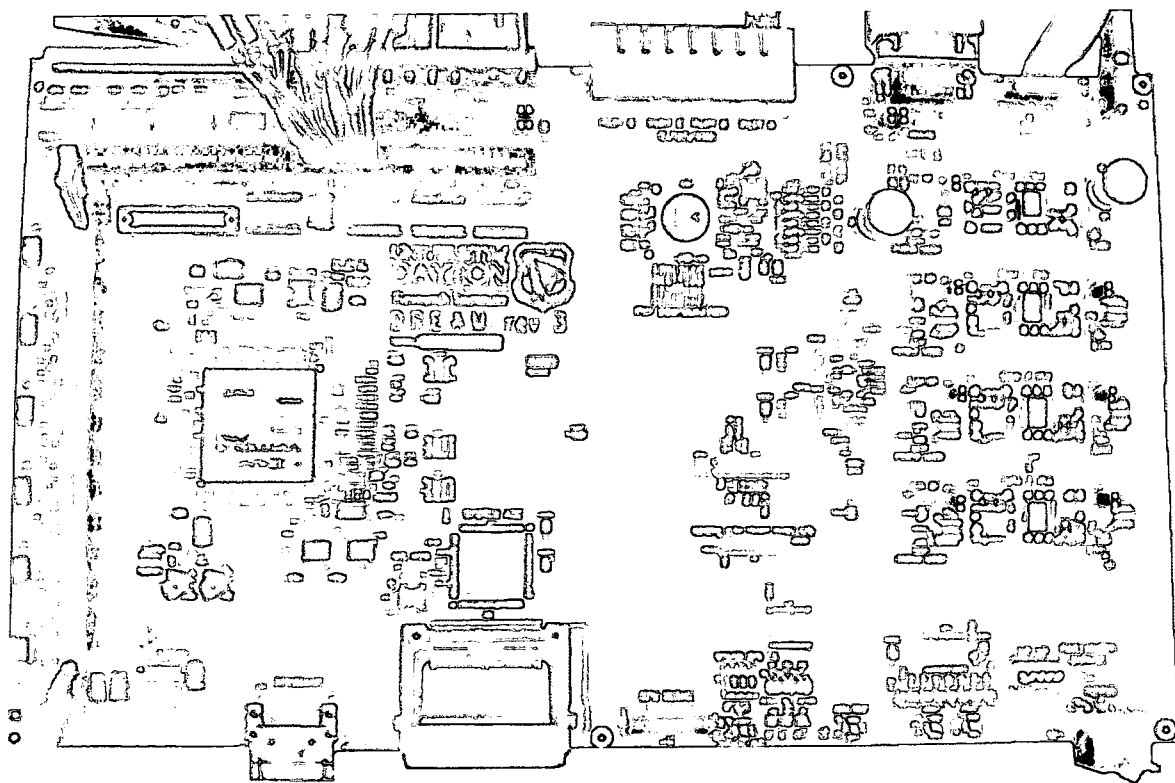


Figure 1.3: Dynamically Reconfigurable Experimental Application Module (DREAM)

is forwarded through the FPGA fabric to the gigabit ethernet interface portion of the reconfigurable hardware. The data is managed by one of the two available PowerPCs.

### 1.2.2 Hardware Platform

Of the available I/O connections to the FPGA, 160 connect to four 50 pin General Purpose Input/Output (GPIO) sockets. The remaining 40 GPIO socket connections are connected to ground. The GPIO can be selected to be either 3.3V or 5.0V logic,

by means of Texas Instruments 16-Bit Level Shifting Transceiver. These connections are located in the upper left portion of the PC board, as seen in Figure 1.2.

On the left side of the PC board is a 160 pin DDR2 RAM connection. The RAM is not utilized in this design, but may be utilized as in [7]. On the lower portion of the PC board is a sixteen channel InfiniBand connection. The InfiniBand is connected to the RocketIO connections on the top side of the FPGA. The InfiniBand connection is not used in this design, but an example of Infiniband interconnection can be seen in [8]. These extra connections allow utilization of the DREAM board in a wide range of applications.

On the DREAM board is a Vitesse Quad 10/100/1000Base-T Gigabit Ethernet PHY, which is located on the upper side of the PC board, as seen in Figure 1.2. This piece of silicon has four gigabit ethernet channels which connect to the RocketIO connections[10] on the FPGA. Directly above the ethernet PHY is four standard ethernet, RJ-45, connections.

### **1.2.3 Embedded Software**

Embedded software is developed in C to utilize the PowerPC on the FPGA. The advantage of the PowerPC in this embedded system is the ability to utilize complex data transfer protocols without the need for dedicated silicon or the utilization of a large portion of the reconfigurable fabric. This design uses the embedded PowerPC for transferring data between hardware cores, as in [7], and transferring data outside the embedded system, as in [1].

As shown in Chapter 3, the embedded software identifies when data is ready to be received from the preprocessing hardware. The data is then transferred to a user

interface hardware core. The RS-232 serial connection on the DREAM board, seen in Figure 1.2, is used for basic user input and output.

Chapter 5 discusses embedded software that manages the way packets, which are received by the gigabit ethernet, are treated. The response to the gigabit ethernet is formed by the embedded software, depending on the configuration which has been selected through the user interface.

### **1.2.4 External Software**

Given that there is no display or keyboard on the DREAM board, user interface with the board must occur through a general purpose computer connection. There are three possible connections on the DREAM board. The least complicated connection is through the JTAG connector which is used to program the FPGA. This connection is used primarily for configuration, initial programming for the PowerPC, and some troubleshooting.

The RS-232 is connected to the embedded processor on the FPGA and simple interface software can be used on the General Purpose computer to interact with the DREAM board, through the use of Hyperterminal. This connection is described in Chapter 3 to control the data collection and to show the data which is being collected.

The serial connection is also described in Chapter 5 for basic ethernet control and troubleshooting. A separate C# program is developed to control communication between the general purpose computer and the gigabit ethernet connection on the DREAM board.

### **1.3 Innovative Contribution**

In this thesis, a custom hardware system is designed to provide a reconfigurable platform for stand-alone data processing. An approach to designing PC board power distribution is described in Chapter 2. Also, custom logic is designed in VHDL to characterize a digitized RF signal in Chapter 3. This is accomplished through generating multiple parallel logic blocks. Finally, a Multi-Gigabit Transceiver (MGT) on a Xilinx VirtexII Pro is connected to a gigabit ethernet chip using an SGMII connection, which is described in Chapter 4.

### **1.4 Thesis Organization**

This thesis documents the design of a custom reconfigurable platform for characterizing and serializing multiple channels of RF sensor data for transmission to a general purpose computer. The presence of an RF signal and the amplitude of the signal are what is needed for the application.

In the following chapters, the system is described in detail. The design of the hardware is discussed in Chapter 2. This chapter discusses the physical connections on the PC board. Chapter 3 explains the design of the data preprocessing section of the design which occurs on the FPGA. The interface between the embedded processor, the FPGA, and the gigabit ethernet PHY is examined in Chapter 4. The packaging of the data for transmission on gigabit ethernet and the collection of the data on a standard desktop computer is described in Chapter 5. The results are seen in Chapter 6 followed by a conclusion.

## **CHAPTER 2**

### **Hardware Layout Descriptions and Considerations**

For an eight channel RF data receiver to be feasible, layout considerations of the processing circuit are first explored. The characteristics of parallel data are discussed in Section 2.1, the power plane placement needs of the FPGA are discussed in Section 2.2 and Section 2.3, and the hardware layout considerations of the ethernet portion of the design are discussed in Chapter 4.

#### **2.1 General Purpose Input and Output Ports**

The General Purpose Input Output (GPIO) ports are used to communicate data from the RF sensors to the FPGA. Therefore the characteristics of the data must be considered when placing copper traces on the board to connect the external signal to the FPGA. One problem that can manifest itself is crosstalk [9], which is described, along with the solution, in the following section. The content of the data is described in Chapter 3.

##### **2.1.1 Crosstalk**

Crosstalk is a noise phenomenon caused by electromagnetic coupling of digital signal paths. Because of this coupling, crosstalk is also referred to simply as noise coupling[9]. This phenomenon can occur between any two digital signal conductors that are acting, in an undesirable way, as transmitting and receiving antennas. The

traces discussed in this section run parallel to each other. The method of coupling experienced in this situation is magnetic coupling[9].

Digital data that is changing value on a conductor cause a corresponding current change. The voltage and current change on the conductor generate an electromagnetic field that radiates from the conductor; this conductor becomes an unintentional transmitting antenna.

Other conductors in the vicinity of the transmitting conductor may act as unintentional receiving antennas. These receiving conductors can be on the same signal layer or on an adjacent signal layer as the transmitting conductor. Figure 2.1 depicts the relationship between adjacent conductors on the same signal plane. The rate of current change within the conductors determines the transmission characteristics. The power intensity that reaches a point on the receiving conductor from a point on the transmitting conductor is characterized by Equation 2.1, also known as the radiation intensity of a point source equation [4]. Where  $I$  is the intensity observed at some point which is at a distance  $r$  from an emitting source with power  $P_s$ .

$$I = \frac{P_s}{4\pi r^2} \quad (2.1)$$

Using this equation, the relationship between signal traces is graphically depicted in Figure 2.1. It is clearly seen by the equation and the diagram that the power intensity being emitted by one conductor decreases by the square as the distance to the receiving conductor increases.

The effect of a digital signal radiated to a nearby conductor is complex. The intensity of the received noise is a function of data levels and data change rates and the physical materials involved. Equation 2.2 and Equation 2.3 are the constants of

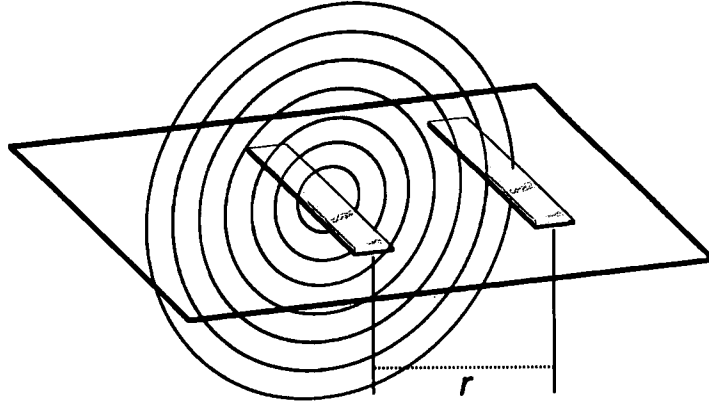


Figure 2.1: Radiation Intensity of Signal Conductors

expected crosstalk [5].  $C_M$  is the mutual capacitance and  $L_M$  is the mutual inductance of the adjacent traces.  $C$  and  $L$  are the capacitance per unit length and inductance per unit length of the trace, respectively.  $V$  is the amplitude of the original signal with a rise time of  $a$  and length  $l$ .

$$\gamma(k+1)V \quad (2.2)$$

$$\text{where } \gamma = \frac{C_M}{C} \text{ and } k = \frac{L_M}{L} \times \frac{C}{C_M}$$

$$\gamma(k-1)\frac{lV}{\mu a} \quad (2.3)$$

$$\text{where } \mu^2 = \frac{1}{LC}$$



These Equations are used in [9] to show the relationship between the amount of crosstalk experienced on a trace and the physical dimensions of the traces and trace spacing. This is discussed further in the following section.

### **2.1.2 Trace Spacing**

In this design, the traces which connect the GPIO to the FPGA are spaced such that changes in data do not cause noise to couple from one data signal trace to another. As seen in the previous section, the amount of power which is received by one conductor from another is directly related to the square of the distance between the conductors.

The spacing between conductors is referred to by the width of an individual trace. In this design, the traces are 5mil wide, therefore, placing the conductors apart by 5mil is referred to as 1X spacing[5]. According to [9], there exists nearly 16% coupling between traces of this width at 1X separation. Increasing the conductor spacing to 3X, or 15mil, spacing decreases coupling between adjacent conductors by approximately a factor of four.

## **2.2 FPGA Switching Needs**

The power requirement of an integrated circuit is, in general, not constant. In integrated circuits, the current required by the circuit is dependant on the number of switching transitions that are occurring at any given time. The source and drain of integrated circuits are typically connected to both a power plane and a ground plane, respectively.

When the current required by a circuit from the power plane changes, the voltage, starting at the point of the current change, adjusts accordingly. Parameters which

contribute to the characteristics of the voltage change are the physical size and shape of the power plane, rate of the current change (Eq. 2.4) and the amount of current change (Eq. 2.5). Where  $V_C$  is the voltage across a capacitance  $C$  due to the integral of the current  $\int I dt$  and  $V_L$  is the voltage across an inductance  $L$  due to the change in current  $\frac{di}{dt}$ .

$$V_L = L \frac{di}{dt} \quad (2.4)$$

$$V_C = \frac{1}{C} \int I dt \quad (2.5)$$

Power supplies in general, cannot react fast enough to eliminate fluctuations in the power plane caused by the changing current requirements of the FPGA switching[2]. It is therefore necessary to provide a mechanism by which these fluctuations can bypass a given circuit. This is primarily accomplished through the use of *Bypass capacitors*.

Bypass capacitors serve the purpose of directing noise that is present on the power plane, from any source, directly to ground, bypassing the switching components on the board. These capacitors are sometimes referred to as decoupling capacitors, as they couple the noise on the power plane to ground and decouple it from the components connected to the power plane. Characteristics of capacitors, in general, and bypass capacitors specifically are discussed in Section 2.3.1. The quantity of bypass capacitance necessary for this design is discussed in the following section.

### 2.2.1 Design Requirements for Capacitance

The circuitry of the FPGA is split into several banks. Each bank consists of a set of power connections, a set of ground connections, and a set of I/O connections. There are eight general purpose banks on the FPGA. Different banks of the FPGA require different amounts of capacitance to protect against the noise due to switching.

The FPGA core power connections provide power to the internal logic of the FPGA and not the I/O functions. Also, there are auxiliary power connections on the FPGA as well. Xilinx recommends that one bypass capacitor be used for each pin connected to the power plane. Therefore, there are thirty 1.5V FPGA Core capacitors and sixteen 3.3V FPGA auxiliary capacitors.

When considering the general purpose banks, the number of capacitors needed is based on the number of I/O connections that are used as outputs[11]. Bypass capacitors are only necessary when pins are used as outputs because output drive current is supplied through the FPGA from the power plane.

The Xilinx reference design assumes that each I/O pin is used as an output. That assumption is revisited in this design process. The significance of using fewer capacitors is discussed in Section 2.3. After analyzing each bank of the FPGA and counting the I/O pins which are connected as outputs, twenty-one of the eighty capacitors on the eight FPGA banks could be removed from the reference design. Table 2.2.1 shows the new capacitor requirements.

## 2.3 Frequency Distribution of Noise Suppression

As seen in the previous section, switching causes noise on the power planes. The speed at which this noise occurs is closely related to the speed at which the switching

Table 2.1: Comparison of Capacitor Usage in Reference Design and DREAM Design

<i>Bank</i>	<i>Reference</i>	<i>DREAM</i>
0	10	10
1	10	10
2	10	10
3	10	10
4	10	5
5	10	3
6	10	7
7	10	4
Total	80	59

Table 2.2: Selection of Bypass Capacitors based on Xilinx Suggested Values[11]

<i>SuggestedRange</i>	<i>SelectedValue</i>	<i>PercentageofTotal</i>
470 $\mu F$ 1000 $\mu F$	470 $\mu F$	4%
1.0 $\mu F$ 4.7 $\mu F$	2.2 $\mu F$	14%
0.1 $\mu F$ 0.47 $\mu F$	0.1 $\mu F$	27%
0.01 $\mu F$ 0.047 $\mu F$	0.01 $\mu F$	55%

event occurs. In other words, a sharper transition of an output signal will cause a sharp voltage transient on the power plane. A simple frequency analysis of a step, seen in Figure 2.2, shows that the generated noise from the transition will have high frequency components.

This suggests the bypass capacitor network must be able to couple a wide band of noise transients to ground. In order to accomplish this, bypass capacitors of different values are chosen for the bypass network. Table 2.3 shows the Xilinx suggested implementation and the capacitors used in this design.

The selection of these capacitors differs from the reference design in that, all of the bypass capacitors in the reference design are 0.1 $\mu F$ . The data in Table 2.2.1 is used

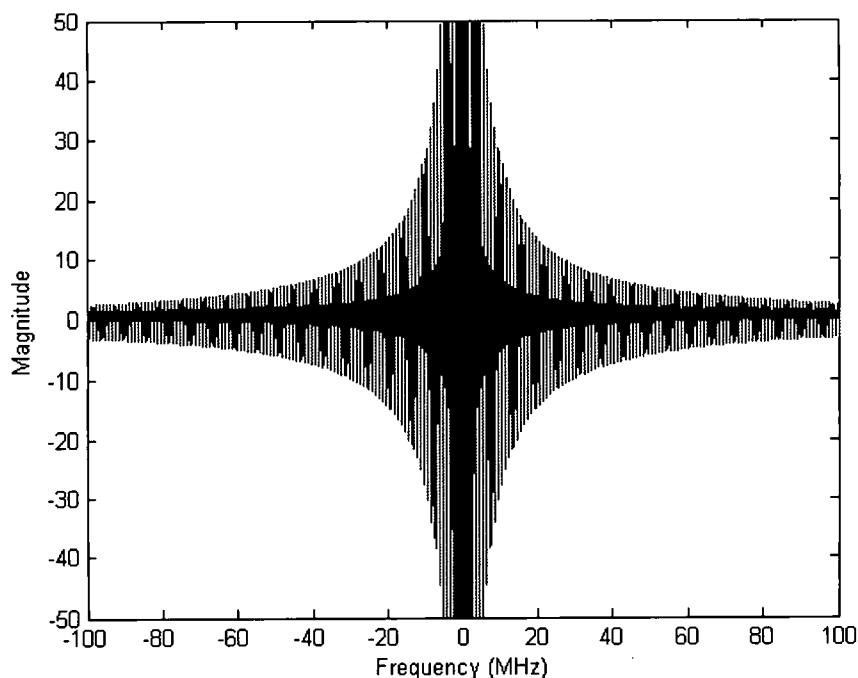


Figure 2.2: Frequency Analysis of a Step Function

along with the data in Table 2.3 to determine the correct number of which capacitor to use. The capacitor selections are shown in Table 2.3. The totals include the general purpose bank capacitors, Table 2.2.1 along with the FPGA Core capacitors and the FPGA Auxiliary capacitors mentioned in the previous section.

### 2.3.1 Bypass Capacitor

The Ideal capacitor model is an open circuit at low frequencies and zero impedance at high frequency. A real, or non-ideal, capacitor has parasitic properties which affect its behavior in the presence of high frequencies. The effect of this is unexpected behavior of the bypass capacitor network.

Table 2.3: Selection of Capacitor Values

<i>Capacitor Value</i>	<i>Percentage Use</i>	3.3V		2.5V		1.5V	
		<i>Exact</i>	<i>Actual</i>	<i>Exact</i>	<i>Actual</i>	<i>Exact</i>	<i>Actual</i>
470 $\mu F$	4%	1.2	1	1.8	2	1.28	1
2.2 $\mu F$	14%	4.2	4	6.3	6	4.48	4
0.1 $\mu F$	27%	8.1	8	12.15	12	8.64	9
0.01 $\mu F$	55%	16.5	17	24.75	25	17.6	18
Total			30		45		32

A capacitor can be modeled as an ideal capacitor in series with both an resistor and an inductor. The equivalent schematic drawing of a real capacitor is seen in Figure 2.3. The resistor represents a constant impedance experienced throughout the frequency spectrum. This resistance is also referred to as the Equivalent Series Resistance (ESR) and is the lower limit of the impedance of a capacitor at all frequencies. One source of this is the resistance found in the metal contacts and plates of the capacitor. Also in the real capacitor model is an inductor, in series. This parasitic inductance increases the impedance of the capacitor as frequency increases. This is known as the Equivalent Series Inductance (ESL)[9]. The primary cause of this parasitic inductance is the capacitor package. The shape and size of the capacitor plates and the connections of the capacitor contribute to the capacitor ESL.

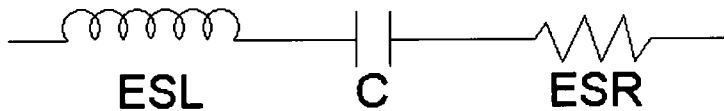


Figure 2.3: Equivalent Schematic of a Real Capacitor

Table 2.4: Parasitic Values of Bypass Capacitors

<i>Capacitor Value</i>	<i>Package</i>	<i>ESL</i>	<i>ESR</i>
470 $\mu F$	<i>Tantalum</i>	3200pH	120m $\Omega$
2.2 $\mu F$	0805	2300pH	5.1m $\Omega$
0.1 $\mu F$	0603	1990pH	101m $\Omega$
0.01 $\mu F$	0402	990pH	634m $\Omega$

The ESL and ESR are used in conjunction with the rated capacitance of a capacitor to determine the real response of the device. Equation 2.6 shows the effective impedance of real capacitor. Where  $Z$  is the effective impedance of a capacitor  $C$ . It is clear in the equation that both the inductance and capacitance are frequency dependant.

$$Z = \sqrt{(ESR)^2 + \left(2\pi f \cdot ESL - \frac{1}{2\pi f \cdot C}\right)^2} \quad (2.6)$$

Equation 2.6 is used to show the frequency response of the capacitors in this design. The ESL and ESR are values which are provided by the manufacturer of the capacitor. For example, according to the the 2.2 $\mu F$  capacitor datasheet, the ESR of the capacitor is the minimum of the frequency response of the 2.2 $\mu F$  capacitor as shown in Figure 2.4. It can be seen in the figure that the frequency at which the capacitor has the lowest impedance is near 2Mhz. The impedance at this point is equal to the ESR of the capacitor. It is then seen that the impedance begins to increase with frequency. This is a consequence of the ESL of the capacitor.

Table 2.3.1 lists the ESR and ESL for each type of bypass capacitor used in this design. The frequency responses of the 470 $\mu F$ , 0.1 $\mu F$ , and 0.01 $\mu F$  capacitors are seen in Figure 2.5, Figure 2.6, and Figure 2.7, respectively.

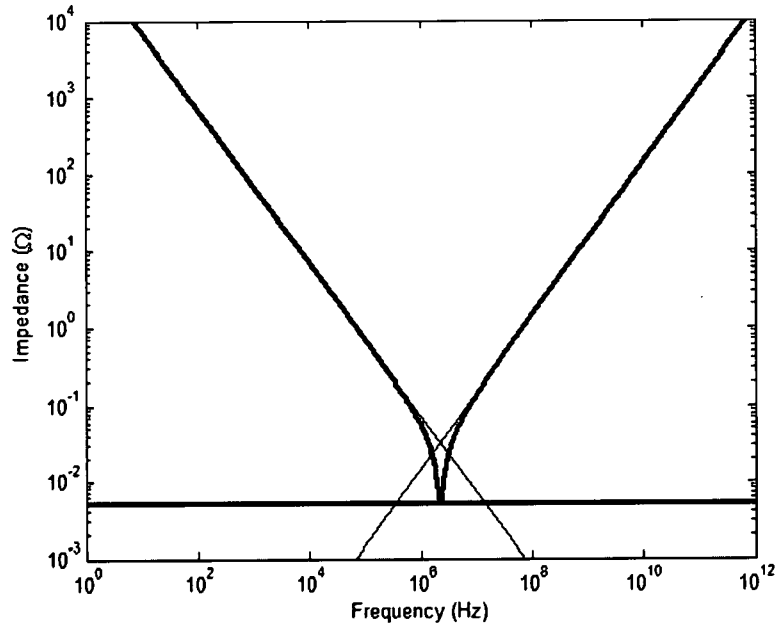


Figure 2.4: Frequency Response of  $2.2\mu\text{F}$  Capacitor

It is noted that lowest impedance of the  $0.01\mu\text{F}$  capacitor, seen in Figure 2.7, is at a higher frequency than the capacitors of higher values. These capacitors will react the fastest to power fluctuations.

### 2.3.2 External Influence of Bypass Capacitors Network

The geometry of the PC board, along with the values of the capacitors, contribute to the effectiveness of the bypass network. There are two significant ways in which this occurs. The first of which, component location, is the less significant, but still important. The lowest value capacitors, the  $0.01\mu\text{F}$  capacitors, are the fastest responders to switching noise on the power plane. The reason for this is the ESL of the small package size, as discussed in the previous section. It is for this reason that



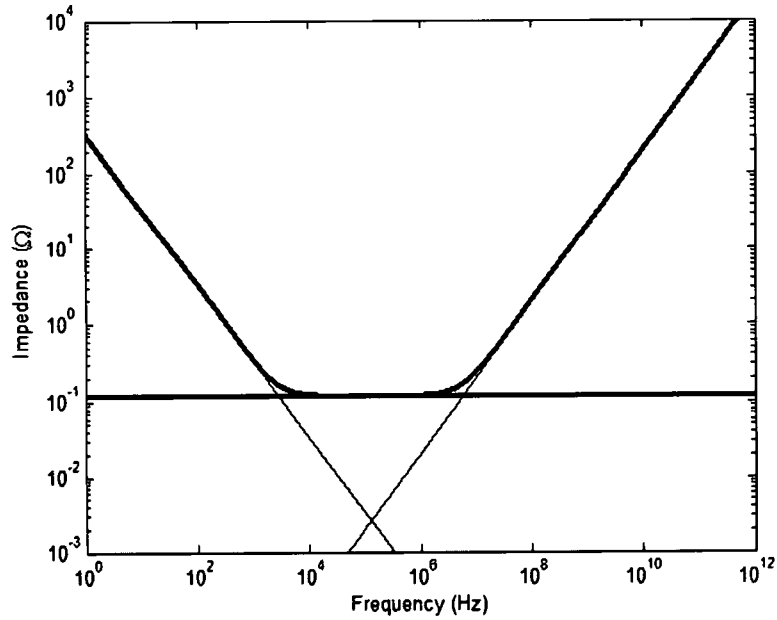


Figure 2.5: Frequency Response of  $470\mu\text{F}$  Capacitor

they should be the closest to the power connections of the FPGA. Xilinx suggests that the lowest value capacitors are to be placed within one inch of their respective FPGA power plane connection [11].

Table 2.3 shows that there are sixty  $0.01\mu\text{F}$  capacitors which must be located within one inch of their respective FPGA pin, in order for the capacitors to be effective. It is also for this reason that the need for capacitors is examined in Section 2.2.1. Analysis of the output usage of the I/O banks, leads to approximately twelve fewer  $0.01\mu\text{F}$  capacitors in the design versus using a capacitor for every power plane connection of the FPGA.

The design, shown in Figure 2.8, utilizes the PC board space directly under the FPGA. There are thirty capacitors under the FPGA. Also, the capacitor placements

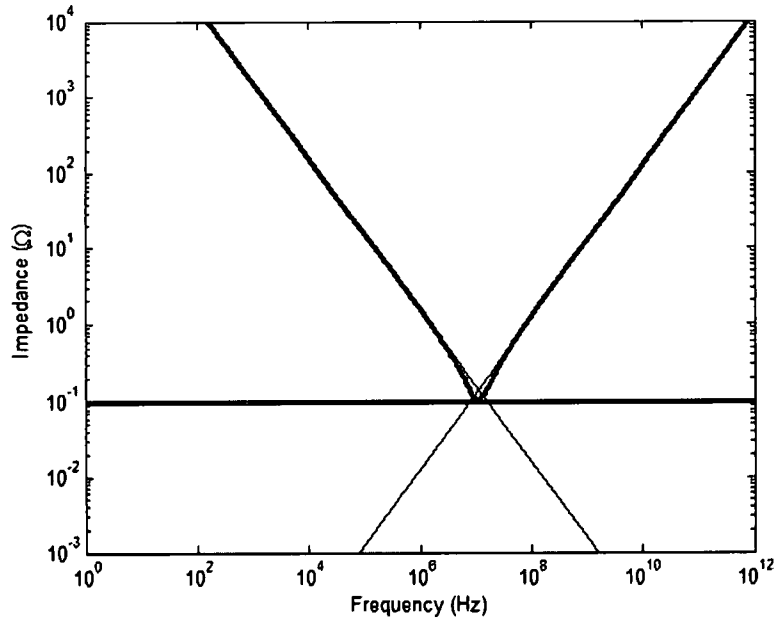


Figure 2.6: Frequency Response of  $0.1\mu\text{F}$  Capacitor

are prioritized. FPGA core capacitors have a higher priority than the capacitors on general purpose banks.

The second effect that board geometry has on the effectiveness of the bypass network is connection between the capacitor and the FPGA. The physical size of the individual capacitors, the connection to the board, and the connection to the FPGA affect the characteristics of the bypass network. In addition to physical capacitors, the copper of the power plane and the copper of an adjacent ground plane also form a capacitor. Because the insulation between board layers is not intended specifically as a dielectric, this is a relatively low capacitance.

The current bypass network consists only of  $0.1\mu\text{F}$  capacitors. Without taking into account the effect of the capacitance formed by the PC board layers, the spectrum of

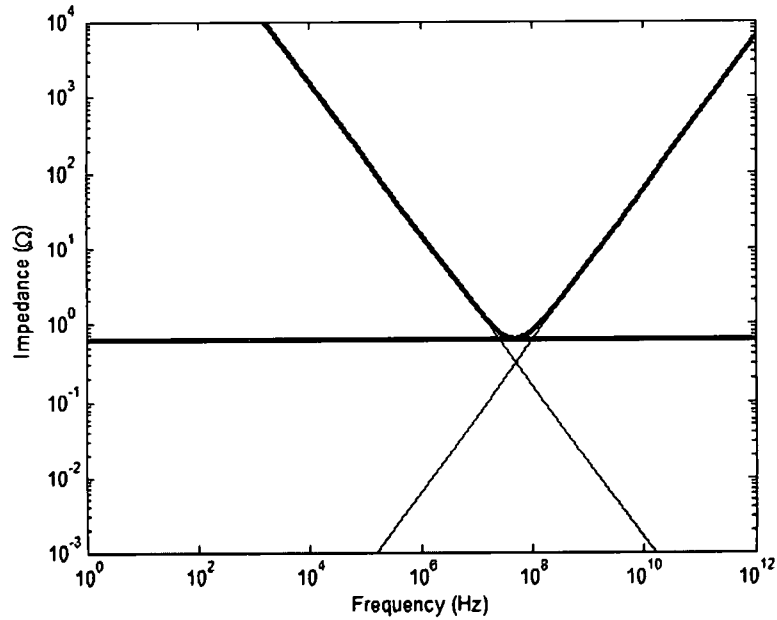


Figure 2.7: Frequency Response of  $0.01\mu\text{F}$  Capacitor

frequency that will be bypassed around the FPGA will closely resemble the frequency response of an individual  $0.1\mu\text{F}$  capacitor, seen in Figure 2.6

### 2.3.3 Analysis of Bypass Network

Considering the different influences on the bypass network, a composite model is developed to ensure that performance will be acceptable. A MATLAB script is written for this analysis. The distances, package sizes, and connection geometry are all taken into consideration. Figure 2.9 shows the results of this analysis. It can be seen that each set of capacitors contribute to bypassing a significant band of transient noise to ground.

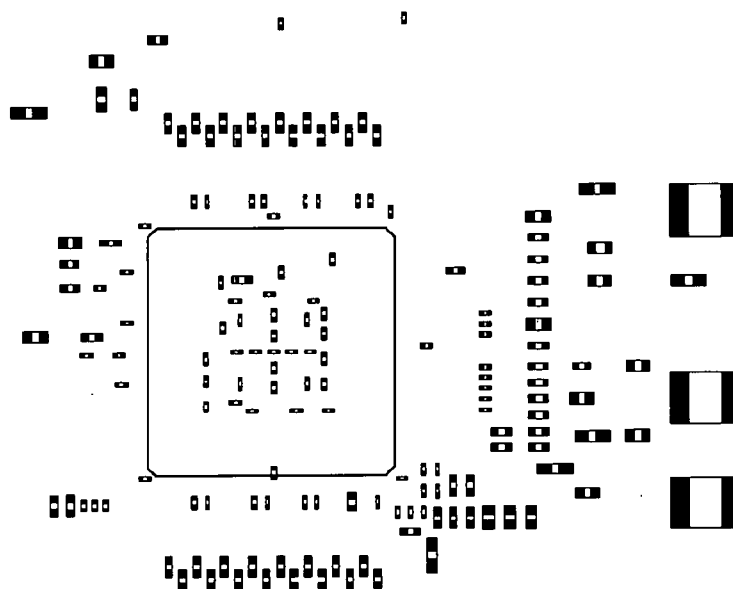


Figure 2.8: Bypass Capacitor Configuration

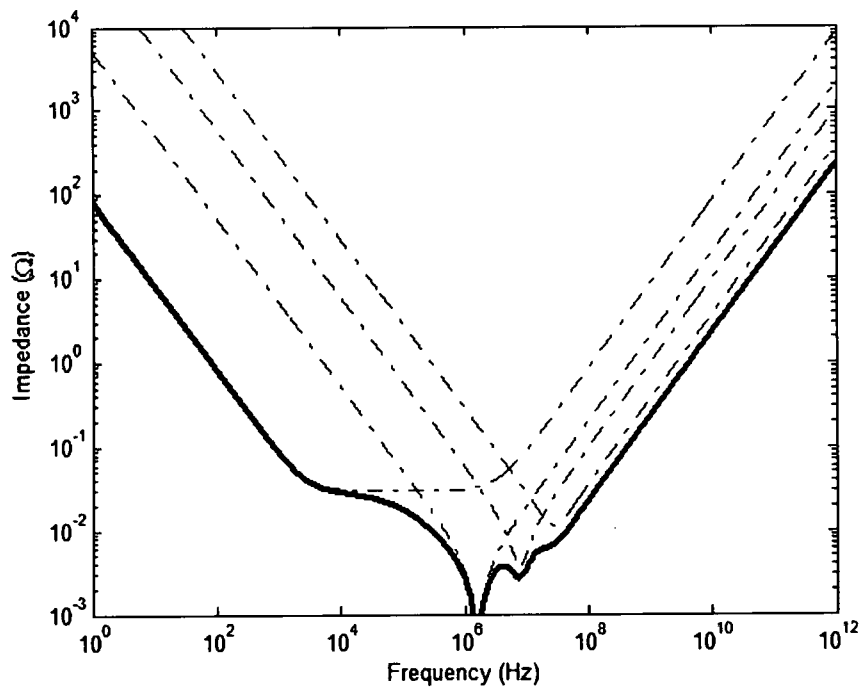


Figure 2.9: Composite Frequency Impedance of Bypass Capacitors

## CHAPTER 3

### Pulse Receptor Core

The Pulse Receptor Core (PRC), which is located on the FPGA, is required to collect radar pulse amplitude data and characterize it. The eight channels of input data are, for the sake of this design, identical. Therefore, the design for a single channel is described followed by the multiplexing of data.

The first requirement is to identify when the channel is active, i.e. when a pulse is detected. When a channel is active, the amplitude of the signal is captured. Finally, when the channel is no longer active, the PRC provides the time that the pulse arrived and duration, or width, of the pulse.

Another requirement for the design is to generate a thirty-two bit parallel control word and a signal to indicate that the control word is valid. The control word must be set for one microsecond. At which time, the data valid signal is asserted for 150 nanoseconds. This is discussed in Section 3.6

#### 3.1 Data Description

There are three global signals which are made available to the design. The first is a highly stable 10MHz clock. From this clock is derived a 50MHz clock. The 20ns period of the derived clock is the resolution required when measuring both the width of the pulse and the Time of Arrival (TOA) of the pulse. This will be discussed in

more detail in Section 3.1.2. The second global signal is a global reset and the third global signal is a start signal. The start signal momentarily goes high, which defines time zero.

### **3.1.1 Characteristics of Data to be Collected**

Each of the eight channels of incoming data consist of an eight bit amplitude and data line which is high when a pulse is detected. In Figure 3.1, these signals are shown entering the channel on the left. The eight bits are the output of an Analog to Digital Converter (ADC). Data from an ADC will not be valid as soon as the *pulse detected* signal becomes active. The time required for an ADC to settle is referred to as the pipeline delay and is specific to the manufacturer of the individual ADC.

Characterization of the pulse includes the width of the pulse and the time at which the pulse arrived. These outputs are also seen in Figure 3.1. Both of these measurements are in seconds, measured as a quantity of clock cycles. The 20ns period of the 50MHz clock is used to measure the duration of the data. The design of this portion of the PRC is discussed in Section 3.3.

### **3.1.2 Global Time of Arrival Clock**

From a highly stable 10MHz clock seen in Figure 3.2, a 50MHz clock is derived using a Xilinx Digital Clock Manager (DCM) module. The 50MHz clock is used to drive a thirty-eight bit counter. The least significant bit of this thirty eight bit counter will toggle every 20ns. The most significant bit of the counter will toggle approximately every 2250 seconds, as seen in Equation 3.1. This means that the clock counter will have unique values for approximately 90 minutes. The output of this counter is used to define the Global Time of Arrival (TOA) for each channel. The

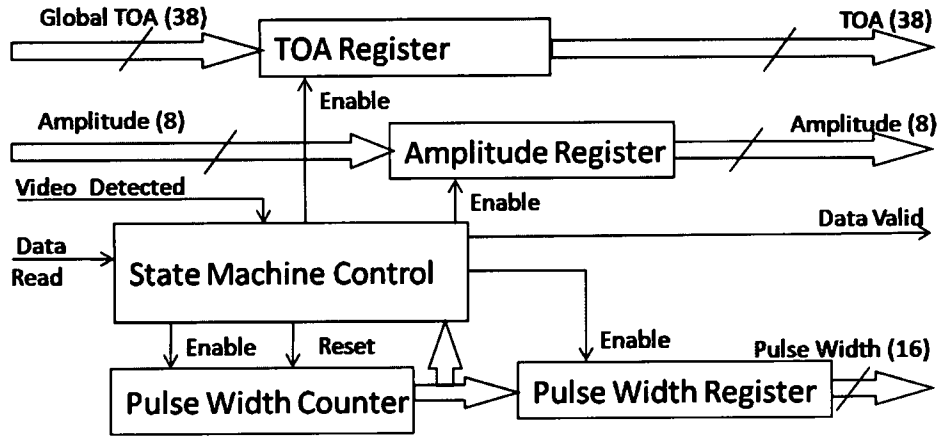


Figure 3.1: PRC Channel Block Diagram

experimental examination of the Global TOA are found in Chapter 6. The source code for this module is found in Appendix B.

$$Time_{Maximum} = \frac{1}{50MHz} * (2^{38}) = 2250sec \quad (3.1)$$

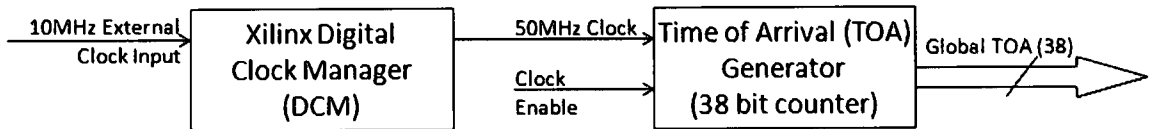


Figure 3.2: Global Time of Arrival Generator Block Diagram

## 3.2 Amplitude Data Collection

When a pulse is detected (detect signal goes high), the input signal amplitude is captured. This is accomplished by latching the eight bits of amplitude, on the



GPIO, in the FPGA, synchronous to the derived 50Mhz clock. As stated before, the amplitude will not be valid immediately. A parameter, specific to the ADC used, can be set to delay the latching of the amplitude. An equation to determine how many clock cycles to wait before latching the ADC data is seen in Eq. 3.2.

$$ADC_{wait} = \frac{1}{20ns} * settlingtime \quad (3.2)$$

For example, if the ADC has a 100ns settling time then the ADC data will be valid 5 clock cycles after the pulse has been detected. This is seen in Eq.3.3.

$$ADC_{wait} = \frac{1}{20ns} * 100ns = 5clockcycles \quad (3.3)$$

The method used to address the issue of settling time is discussed in the following section.

### 3.3 Characterization of Pulse

As previously stated, a requirement of the PRC is to characterize the pulse which has been detected. The time that the pulse arrived and the length of the pulse are required by the application.

#### 3.3.1 Pulse Width

When the pulse arrives, the system begins measuring the pulse width with a 16 bit counter. The counter is clocked by the 50MHz clock, thus the resolution of the pulse width counter is 20ns.

Furthermore, to prevent erroneous pulse detection due to glitches in the external circuitry, a detection parameter is added. The parameter, called the *pulse width discriminator*, is the minimum time of a valid pulse.

There are three measurements starting at the same moment. When the pulse is detected: the starting point for the ADC pipeline delay, the pulse width discriminator, and the pulse width. This design identifies each of these measures with the sixteen bit pulse width counter and a state machine.

The inputs to the state machine, seen in Figure 3.3, are the clock, the pulse detected signal, and the value of the pulse width counter. When there is no pulse detected, the state machine is in a waiting state; the value in the pulse width counter is set to zero. Only a pulse detection causes the state to change to a pre-threshold state. In this state, the pulse width counter begins running. Every following positive clock edge causes, as long as the pulse detected signal is still active, a comparison between the value of the pulse width counter and both the pulse width discriminator and the ADC pipeline delay.

When the value of the pulse width counter is greater than the pulse width discriminator, the state machine enters the post-threshold state. If the pulse detected signal becomes inactive before the threshold is reached, then the state machine clears the pulse width counter and returns to the wait state. When the value of the pulse width counter becomes greater than the ADC pipeline delay, the state machine enters the measuring state. A signal is asserted during the transition into this state which causes the eight bit value of the ADC to be latched into a register and allows the pulse width counter to continue to count.

It is assumed that any momentary fluctuation or glitch, which could possibly cause a false signal detection, will be shorter than the time required for the ADC to settle.

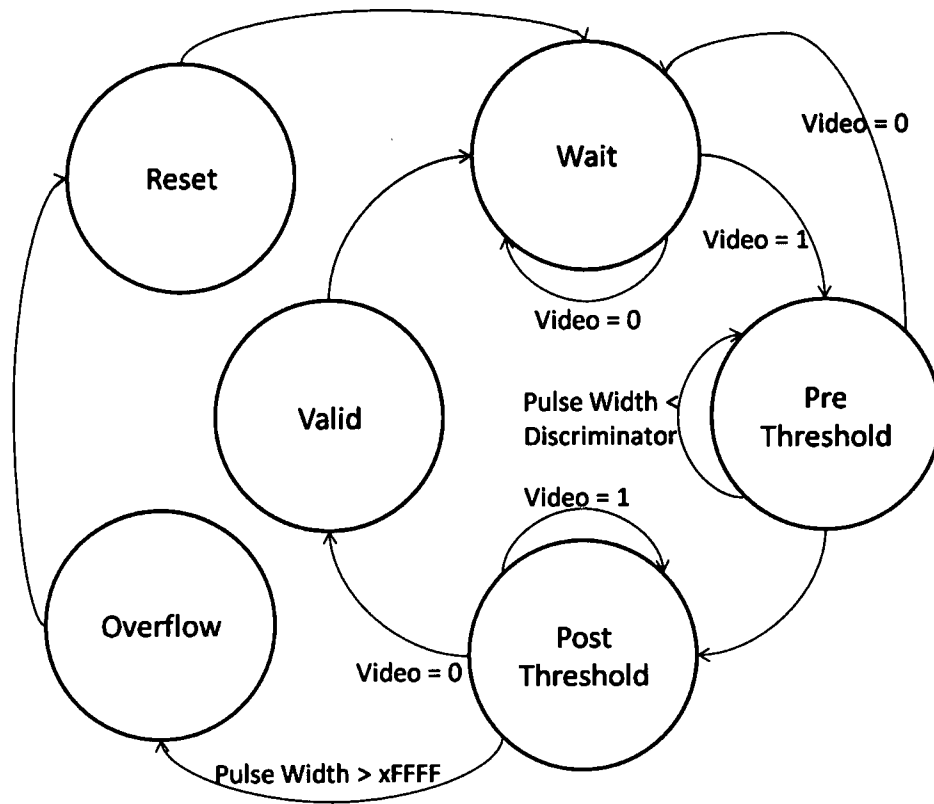


Figure 3.3: PRC Channel State Diagram

During the clock cycle that the *pulse detected* signal becomes inactive, the value in the pulse width counter is captured in a register. The amplitude of the pulse and the duration of the pulse are, also, temporarily stored.

### 3.3.2 Time of Arrival of the Pulse

During the wait state, a Time of Arrival (TOA) register is left cleared. During the clock cycle that transitions the state machine into the pre-threshold state, the thirty eight bit value of the Global Time of Arrival Clock is registered into a Time of Arrival (TOA) register. If the state machine transitions back to the wait state before

entering the post threshold state, the signal which resets the pulse width counter also clears the value of the TOA register. When the state machine leaves the measuring state, the value of the TOA register along with the other characterized and collected data waits in registers to be collected.

### 3.4 Channel Multiplexing

Each of the eight channels provides 62 bits of data to define a pulse. There are 38 bits used to identify the Time of Arrival. There are 16 bits to identify the Pulse Width. And, there are 8 bits to identify the Amplitude of the pulse. Additionally, three bits are used to represent from which channel data had originated. The use of this data is shown in more detail in the next section. Figure 3.4 shows a block diagram of the eight PRC channels and the multiplexer.

Simple multiplexing of the collected data using the *valid* signal from the individual channel corresponds to data being collected in reference to the 50MHz clock. This is seen in Figure 3.5. Along the bottom of the figure, the *pulse detected* (referred to in the Figure as Video) signals become active for arbitrary periods of time. When the detection signal is de-asserted, the amplitude, width, and time of arrival of each pulse along with the channel address is seen on each of the *Test* busses in the Figure. Since the data from each channel is held in a register until it is collected by the PowerPC, the case of two channels becoming inactive at the same time is addressed through priority encoding of the data valid signals.

To insure that the data from all eight channels is captured, a faster clock is used for multiplexing. The 100MHz clock which is used for the PowerPC is readily available. This design operates the multiplexer using this 100MHz clock. Using the PowerPC

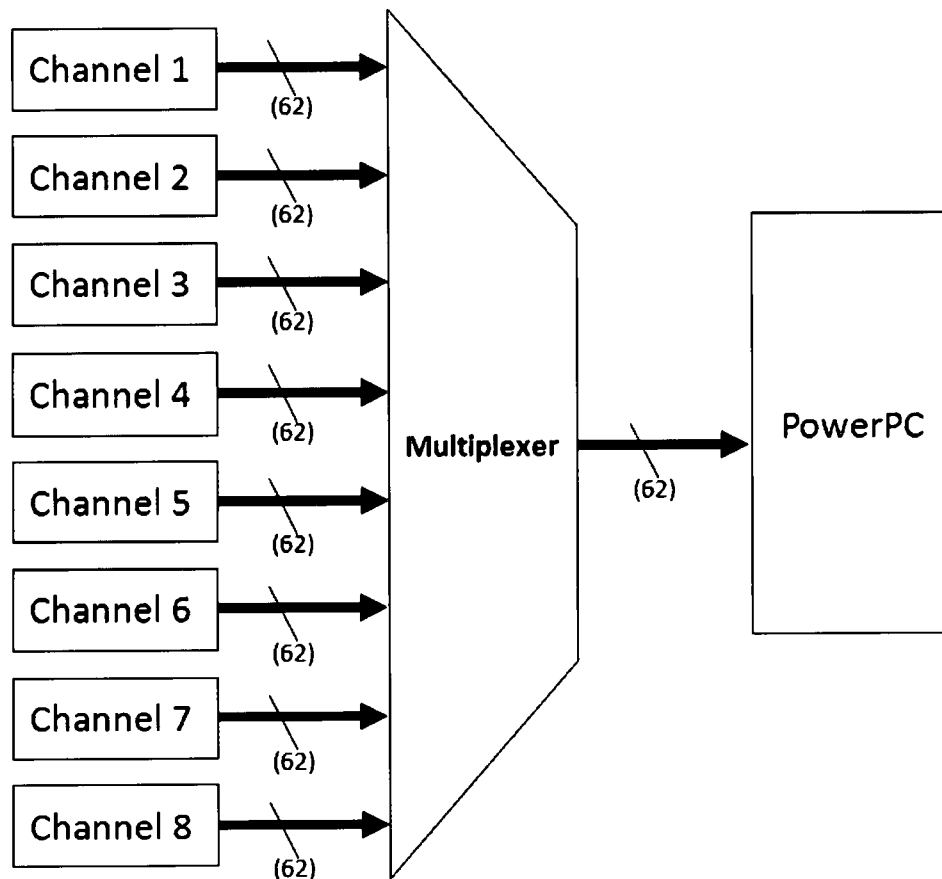


Figure 3.4: PRC Channel Data Multiplexing Block Diagram

clock also reduces potential signal registration issues. While the data waiting to enter the multiplexer is subject to change with the 50MHz clock, it can be collected at twice the rate at which it changes. This provides a theoretical minimum for the *pulse width discriminator* which is shown in Equation 3.4. If every channel has data waiting, as long as a pulse is greater than eight 100MHz clock cycles, the data from each channel can be collected without any data being over written. The eight 100MHz clock cycles corresponds to 80ns or a *pulse width discriminator* of 4. This situation is simulated in Figure 3.6. In the simulation, all *pulse detected* signals, labeled Video in

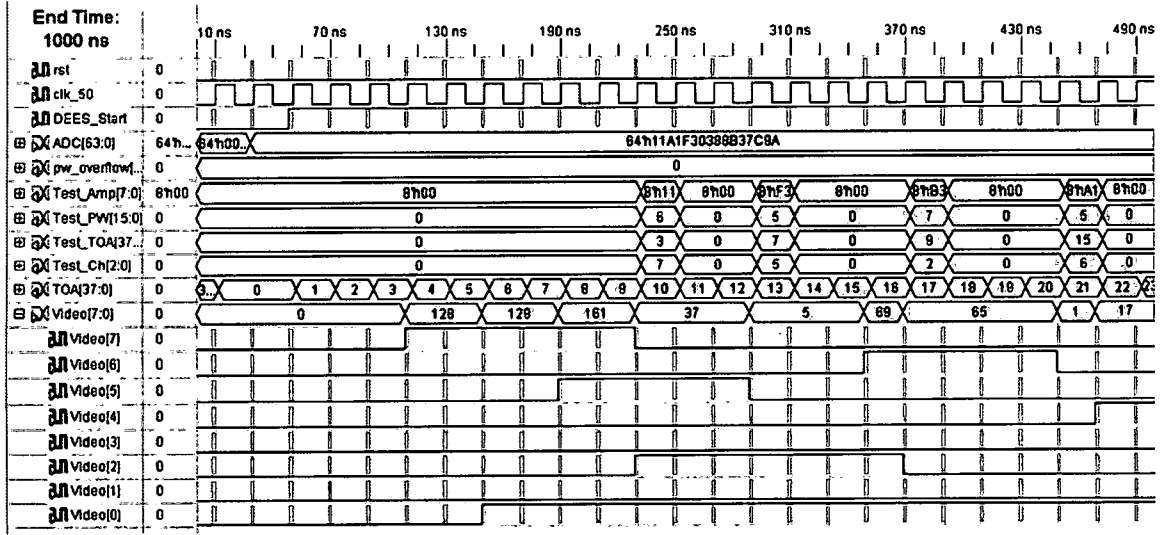


Figure 3.5: PRC Channel Data Collection Simulation

the simulation, are simultaneously asserted for five 20ns clock cycles. After 100ns, the signals are simultaneously de-asserted. It is seen in the simulation that the amplitude, the channel, the pulse width, and the time of arrival of each channel is present on its corresponding bus for one clock period. The data generated from all channels is collected in 80ns.

$$Discriminator_{PW} * \frac{1}{clk_1} = Channels * \frac{1}{clk_2} \quad (3.4)$$

$$Discriminator_{PW} * \frac{1}{50MHz} = 8 * \frac{1}{100MHz}$$

$$Discriminator_{PW} = 4$$

VHDL code which instantiates this design is found in Appendix B. In Figure 3.6 is the simulated behavior of this module.

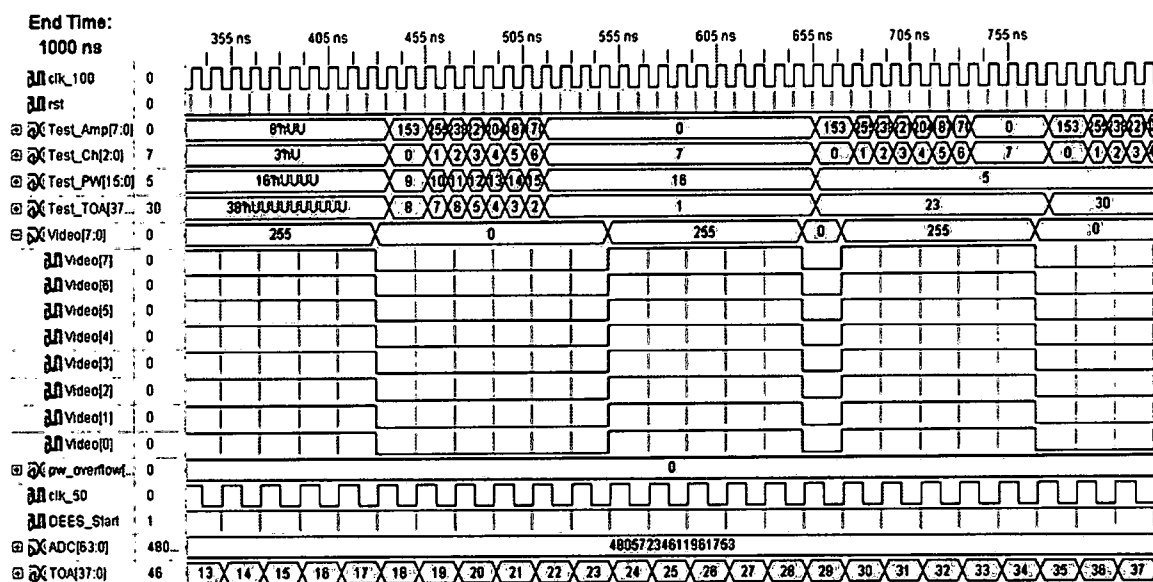


Figure 3.6: PRC Channel Data Multiplexing Simulation

### 3.5 Packet Forming

The pulse data is collected from the PRC data collection core by the PowerPC. The data collection core is embedded in a Xilinx provided Onboard Peripheral Bus (OPB) wrapper. This wrapper contains all the signalling overhead required by the to communicate with the Power PC through the OPB. Thirty-two 32 bit registers are available as the primary method of communication between data collection core and the PowerPC.

The lower 32 bits and the upper 30 bits of the 62 bit data word are placed into consecutive registers. Which two registers is dictated by the three bits discussed in the previous section that denote from which channel the data originated. The *data valid* signal asserted by an individual channel is used to inform the PowerPC that

Table 3.1: Comparison of Capacitor Usage in Reference Design and DREAM Design

<i>Register Address</i>	<i>Contents</i>
00000	Channel 1 bits 62-32
00001	Channel 1 bits 31-0
00010	Channel 2 bits 62-32
00011	Channel 2 bits 31-0
00100	Channel 3 bits 62-32
00101	Channel 3 bits 31-0
00110	Channel 4 bits 62-32
00111	Channel 4 bits 31-0
01000	Channel 5 bits 62-32
01001	Channel 5 bits 31-0
01010	Channel 6 bits 62-32
01011	Channel 6 bits 31-0
01100	Channel 7 bits 62-32
01101	Channel 7 bits 31-0
01110	Channel 8 bits 62-32
01111	Channel 8 bits 31-0

the 62 bit pulse data from the channel is valid and can be collected. Table 3.5 shows the register mapping.

### 3.6 Control Word Generation

Three functions are required for proper generation of control words. An overview of the control design is seen in Figure 3.7. The value of the control word, to be asserted by the design, originates from a general purpose computer and is communicated to the design through an ethernet connection to the PowerPC. Design of the ethernet communication system is described in Chapter 4. The command word in the PowerPC is then transmitted out through the GPIO connection.

A custom hardware core is designed for the FPGA and is connected to the PowerPC through the OPB bus. This interface uses shared registers between the PowerPC



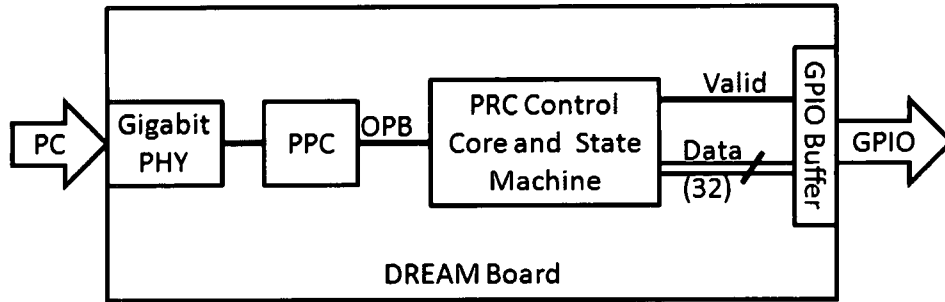


Figure 3.7: Control Word Generation Block Diagram

and the control word generation core. Using these registers, this core is required to set and hold the thirty-two bit command word and data valid signal according to the design specification. Designing a state machine is chosen to accomplish this task. Figure 3.8 is a state diagram of the simple controller state machine. During the *Wait for Data* state, all outputs of the control word generator are disabled. The state machine waits for a PowerPC shared register to change, indicating a control word is ready to be asserted. Once a register changes, the control word generation core registers the data from the register and state machine transitions into the *Data on Bus* state. During the this state, the control word generation core asserts the value, registered during state transition, on the GPIO connection.

During the *Data on Bus* state, the asserted value must be held for  $1\mu\text{s}$ . To accomplish the correct duration, a 7-bit counter is used with a 100MHz clock, a period of 10ns. The counter output counts to 99 and asserts a signal on the next clock to indicate  $1\mu\text{s}$  has passed. A different value must be calculated if another clock frequency is the input for the counter. The signal indicating that  $1\mu\text{s}$  has passed causes the state machine to transition into the *Data Valid* state.

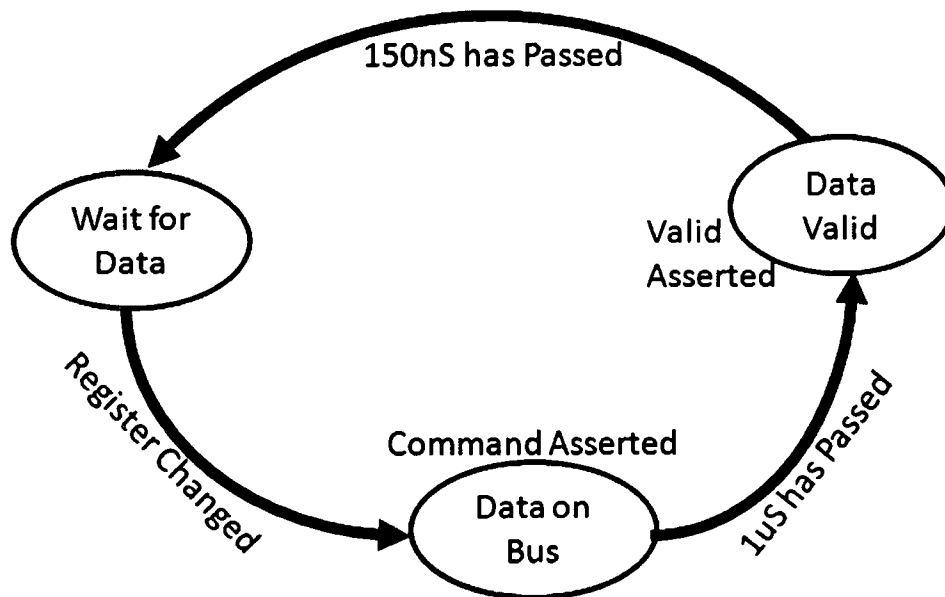


Figure 3.8: Control Word Generation State Diagram

During transition into the *Data Valid* state, the Data Valid output of the control word generation core is asserted. This signal is to be asserted for 150ns. To accomplish the correct duration, the output of the previous counter, which indicated that  $1\mu\text{s}$  had passed, is used to indicate and additional fourteen 10ns clock cycles have elapsed. On the next clock cycle, a signal indicating that 150ns has passed is asserted, which causes the state machine to transition back to the *wait* state, resetting the clock counter.

The VHDL code for this module can be found in Appendix B. The experimental results are found in Chapter 6.

## CHAPTER 4

### Gigabit Ethernet Physical Layer Interface to the FPGA

The DREAM board design includes a *Quad 10/100/1000Base-T PHY with an SGMII/SerDes MAC interface* manufactured by Vitesse[10]. The PHY is used to interface the signalling and timing of gigabit ethernet with a given design. In this design, the PHY uses the Serial Gigabit Media Independent Interface(SGMII) to communicate with the FPGA. This is seen in Figure 4.1. The SGMII connection in the FPGA is through the Multi Gigabit Transceivers (MGT) on the VirtexII. Inside the FPGA, the SGMII connection to the MGTs are managed by the use of a RocketIO core[14].

A bridge core exists which can connect a SGMII device with a Gigabit Media Independent Interface(GMII) device. This is labeled as the "GMII to SGMII Bridge" in Figure 4.1. The GMII connection can be used by a Xilinx provided gigabit ethernet core. A Gigabit Ethernet Media Access Controller (GEMAC) core is used as the interface to the embedded PowerPC system. The GEMAC, in turn, connects to the PowerPC using the CoreConnect<sup>TM</sup> architecture[6]. Specifically, the Processor Local Bus (PLB) of the CoreConnect<sup>TM</sup> standard is used to connect the GEMAC to the embedded system.

There are also clocking and management considerations in the design seen in Figure 4.1. These will be discussed, in detail, in Section 4.3 and Section 4.4, respectively

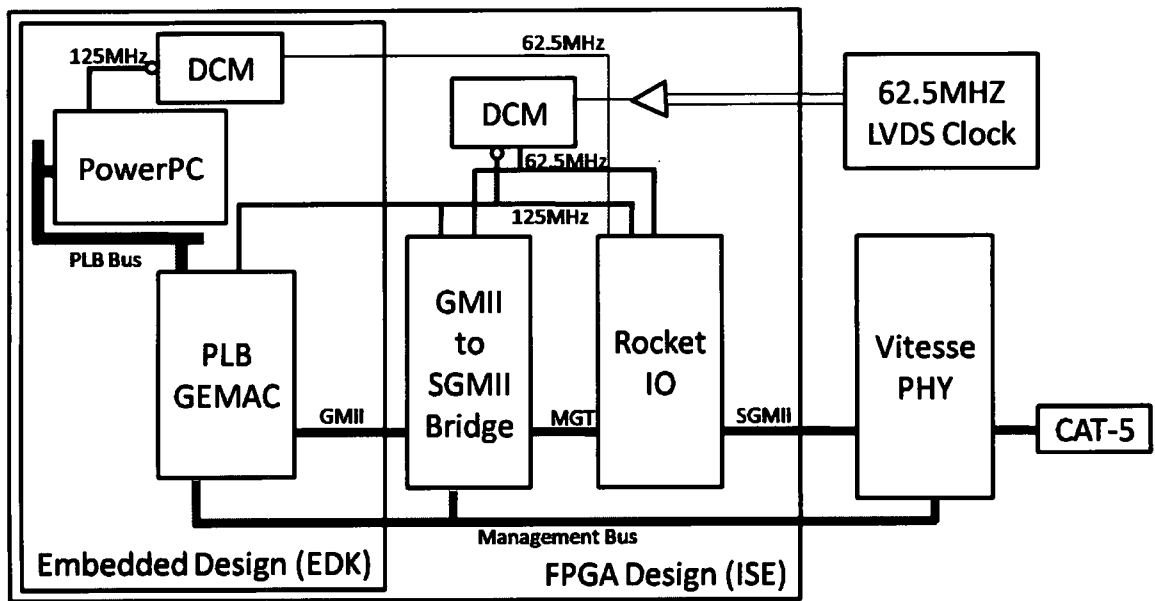


Figure 4.1: Block Diagram of Gigabit Ethernet

## 4.1 Layout Considerations

The Vitesse PHY is an external component chosen for the relatively few physical connections that are required for operation. The major connections of the PHY are the connections to the FPGA through SGMII and the connection to the ethernet magnetics. Both of these connections have relatively the same data width. However, the PHY generates ethernet frame data for transmission and removes ethernet frame data from received incoming ethernet frame data. This extra framing data is only present on the ethernet side of the PHY, which guided the decision to place the Vitesse PHY closer to the ethernet connection, making this a shorter path.

## 4.2 Gigabit Data Path Interconnections

With the exception of the PHY, the majority of the gigabit ethernet component of the design resides in the FPGA. The endpoints for the FPGA design are the PLB of the PowerPC and the RocketIO pin connection. The FPGA design occurs in two environments. Xilinx Integrated Software Environment(ISE) is used to develop the overall FPGA design. In the ISE design is the SGMII to GMII bridge core and the RocketIO core and the embedded system core. The embedded system core is developed using Xilinx Embedded Development Kit(EDK). In the EDK project is the PowerPC, the processor interconnection busses, the RS-232 UART, and the GEMAC. The following sections describe the components in the FPGA.

### 4.2.1 Embedded System with GEMAC Core

The GEMAC core, however, is capable of connecting through either a Ten Bit Interface (TBI) or a Gigabit Media Independent Interface(GMII). For this design, the GMII interface is chosen because there is not an apparent connection made between the GEMAC core and the PHY using the TBI interface. The GEMAC core is connected to the RocketIO by means of a GMII to SGMII bridge. This bridge is a LogiCORE core, which is available from Xilinx[14].

The embedded portion of the design is developed with the Xilinx Embedded Development Kit (EDK). The embedded system interconnects the on-chip FPGA with system peripherals. These peripherals include the RS-232 UART through the Onboard Peripheral Bus (OPB) core and the GEMAC. The GEMAC is implemented using a standard Intellectual Property (IP) core provided by Xilinx. The specific core is the *PLB 1-Gigabit Ethernet Media Access Controller (MAC) with DMA (v1.01a)*[15].

The PLB GEMAC has many different parameters which can be changed to suit the need of the designer. In this design, most of the default settings are acceptable. First, the GEMAC core is set to utilize a *Simple DMA* interface. The other options are a simple FIFO or a scatter-gather DMA. The simple DMA allows direct access to the memory of the embedded system. This allows the data generated in the other hardware (i.e. the PRC) to be collected without being passed through the PowerPC. This configuration, versus the scatter-gather DMA, was selected because of the simple nature of the data which it would be collecting.

The Management bus is, by default, disabled. The management bus is very important to this design, as seen in Section 4.4. The GEMAC core is the master of the management bus, on which the SGMII to GMII bridge and the PHY connects. Once the management interface is configured to be instantiated in hardware, the management bus clock must be set. The management bus clock is derived from the primary GEMAC clock. Also, the management bus clock must be less than 2.5MHz. A divider parameter is set in the GEMAC core to generate this clock. Equation 4.1 shows the derivation of the requirement for the management clock to be greater than the binary value 0b11000. The value 0b11010 satisfies this criterion and was selected. Equation 4.2 shows that the actual clock frequency of the management interface is approximately 2.4MHz.

$$2.5MHz > \frac{125MHz}{(ClkDivider + 1) * 2} \quad (4.1)$$

$$(ClkDivider + 1) * 2 > 50$$

$$ClkDivider + 1 > 25$$

$$ClkDivider > 24$$

$$ClkDivider > 0b11000$$

$$Clock_{MIIM} = \frac{125MHz}{(0b11010 + 1) * 2} \quad (4.2)$$

$$Clock_{MIIM} = \frac{125MHz}{52}$$

$$Clock_{MIIM} \approx 2.4MHz$$

### 4.2.2 GMII to SGMII Bridge

Between each PHY channel and each GEMAC core, there must be a bridge to connect SGMII to GMII. This is accomplished using the *Ethernet 1000BASE-X PCS/PMA or SGMII v8.0* core from Xilinx.

The Xilinx core is implemented through the use of ISE. As expected by the use of the word *or* in the title of the core, a parameter must be set to enable the SGMII communication bus. This also disables the *1000BASE-X PCS/PMA* components of the core. Also, *clock tolerance compliant with ethernet specification* was selected.

### 4.2.3 SGMII over RocketIO

The *RocketIO v8.0* core provided by Xilinx is used to allow the SGMII connection to be external to the FPGA. The primary use of this core is ensure that the SGMII data leaving the FPGA is correct to the standard. There are several parameters which must be set when configuring the RocketIO core.

The voltage swing of the differential data is set to be 600mV. This is parameter is set to be equal to the voltage swing expected by the Ethernet PHY. The PHY setting will be discussed in Section 4.4. A data width of one byte is selected; two bytes is the default value. The data width is dictated by the SGMII to GMII bridge.

The encoding for the RocketIO core is set to *Static encoding control: Enable 8B/10B encoding* with a pre-emphasis of 20%. Also, CRC is disabled for both the transmitter and the receiver.

### 4.3 Gigabit System Timing Requirements

On the FPGA, the PowerPC, the PLB GEMAC, the SGMII to GMII Bridge and the RocketIO core all have different clock requirements. These clock requirements are satisfied through the use of a single 62.5MHz clock on the PC board and two Digital Clock Managers (DCM). A DCM is a core which is provided by Xilinx to derive clock signals from existing clock signals. The DCM is discussed in Chapter 3. Two clock managers are used because of the location, in silicon, of the PowerPC and the RocketIO and the nearest location of a DCM.

The 62.5MHz clock signal is the input to each DCM. The PowerPC DCM is used only for the PowerPC. The DCM is configured such that the output drives the PowerPC at 125MHz. The other DCM is used to provide the clock signals to the remainder of the cores in the gigabit ethernet design. The doubled output of this DCM, 125MHZ, is provided to each of the gigabit ethernet cores in the FPGA. This output is also inverted. This is at the recommendation of the Xilinx design recommendation concerning the RocketIO[13].

The output of the DCM that is a stable representation of the input signal is connected to both the GMII to SGMII Bridge core and the RocketIO core. Also, the original 62.5MHz clock is also connected directly to the RocketIO core. This is at the recommendation of Xilinx[13].



## 4.4 Management Interface

There is a management interface which exists in the GEMAC core, the SGMII bridge, and each of the four PHY channels on the Vitesse chip. The Management interface is used to set registers, which are internal to each device, to their required settings. The GEMAC acts as the master, or arbiter, of the management.

There are two similar types of management interfaces encountered in this design. The GEMAC utilizes a Media Independent Interface Management (MIIM) type. MIIM consists of a clock signal, a data output signal, a data input signal and a tristate control signal. Conversely, both the SGMII bridge and each of the PHY channels use a Media Dependent Input/Output (MDIO) type. MDIO consists of a clock signal and a tristate data signal for both input and output. Both types make packets of the management data in the same way.

The interconnection between the components on the Management Interface is straightforward. First, the MIIM of the GEMAC, which is defined as the arbiter, provides the clock signal for all of the devices on the management bus. Next, the input and output are connected to a tristate connection, controlled by the tristate control signal of the MIIM.

The completion of this connection allows software code to be written for the embedded processor to manipulate the configuration registers of the SGMII bridge, and each of the PHY channels. Commands that are sent over the management bus start as settings written by the PowerPC to configuration registers in the GEMAC. There are two dedicated GEMAC registers for this connection, a control register and a data register. The control register, shown in Figure 4.2 controls whether the management operation will be a read or a write operation. Each device on the

management bus is individually addressable. This address is between bits 2 and 6 in Figure 4.2. Following the device address is the register address of the device which the data will be either read or written. Figure 4.3 shows the register by which the sixteen bit data which will read from or written to the addressed device on the management bus.

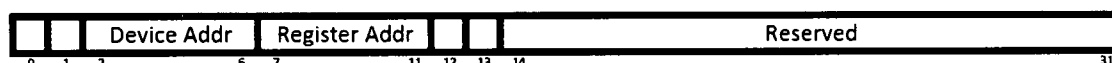


Figure 4.2: Management Control Register

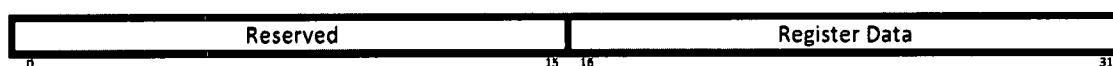


Figure 4.3: Management Control Register

#### 4.4.1 PHY Channel Configuration Register Settings

On the Vitesse PHY, there are four gigabit ethernet PHY devices. Each of these devices are addressable through the management interface. The four PHY devices are addressed individually using the two least significant bits of the management address. The most significant three bits are set to '000' by default but can be changed through setting a management register in the PHY device.

Table 4.1: Default Representation of LED Indications

<i>LED Location</i>	<i>Representation</i>
Connector Right	Link1000
Connector Left	Link/Activity
On PC Board 1	Duplex/Collision
On PC Board 2	Link/Activity

There are forty-eight addressable management registers accessible for each PHY device. The PHY management registers control the operating characteristics of the PHY device. The way in which the PHY connects with external ethernet devices is subject to the register settings. The allowed or forced speed of the connection or the ability for the PHY to autonegotiate a speed can be defined by register settings. The representation of the LEDs, which are connected to the PHY, can also be set using the management registers. Table 4.4.1 lists the default settings for the LED indications.

The management register settings of the PHY devices not only customize the device, but operation of the device is not possible without changing default register settings. One register in particular contains a default value which effectively disables the *signal detected* signal, a signal which is necessary for the GEMAC on the FPGA to know that a link has been established.

Using the management interface, configuration registers in each of the four PHY channels can be read and/or set. All of the settings in these registers affect the operational characteristics of an individual PHY channel. This is inherent in the way in which data is prepared to be transmitted across the management bus as each

command sent is addressed to a particular device as demonstrated previously in Figure 4.2.

Registers can also be set to cause the PHY to return each packet that it receives[10]. This loopback mode can be used to test the functionality of the external ethernet connection. This scenario requires, of course, two ethernet PHY devices. The source code which uses this functionality to verify the external connection is found in Appendix B.

After the electrical functionality is verified, generic packets can be generating through the use of another management register in the PHY. An automatic packet generator exists in the each PHY device and can be set to send valid or invalid data packets from the PHY device. The source code which uses this functionality is also found in Appendix B. Further discussion of valid and invalid packets is found in Chapter 5.

#### **4.4.2 SGMII Bridge Configuration Register Settings**

The SGMII bridge is also connected to the management interface. The address of the SGMII bridge core is explicitly defined in the VHDL code which instantiates it. This design only utilizes one PHY connection and, therefore, only one SGMII bridge which has been given the arbitrary address of six (00110).

In the documentation for the core, the management interface is said to be optional[14]. However, without access to the settings in the management interface, the selected gigabit ethernet PHY will not operate correctly. The default value of the SGMII control register configures the SGMII core to electrically isolate SGMII logic from GMII logic [14]. With this setting, the SGMII bridge cannot operate correctly.

## CHAPTER 5

### Ethernet Software

Once the physical electrical connections for the gigabit ethernet have been established, the data communicated on the ethernet channel is considered. Ethernet communication occurs in layers. With the exception of the outermost layer, each layer of ethernet communication is encapsulated by the layer above it [3]. The outermost layer is referred to as the *Physical Layer*. This layer is the electrical connection discussed in Chapter 4. Inside the *Physical Layer* is the *Data Link Layer*. This layer is implemented in the design by the use of the ethernet data structure. This is described in Section 5.1. Within the *Data Link Layer* is the *Network Layer*. This layer is discussed in Section 5.2.

#### 5.1 Ethernet Packet Structure

The *Data Link Layer* is implemented through the use of the ethernet data structure. In each ethernet packets there are seven fields. The data in each field is referenced in sets of eight bits, called octets. [3]. Figure 5.1 show the basic ethernet packet. The first field, which is seven octets, is the preamble, followed by a start of frame delimiter (SFD) octet. These fields denote the beginning of an ethernet frame. The following field is the MAC address to where the packet is being sent, followed by the MAC address from where the packet originated. Both of these fields are six octets long.

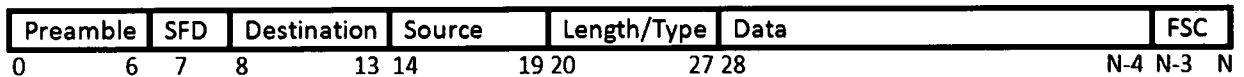


Figure 5.1: Structure of Ethernet Packet

The field that follows the address fields is the *Length/Type* field. This two octet field can either hold the length of the ethernet packet or the type of protocol used in the *Network Layer* encapsulated in the packet. The next field is the data field, which can be 46 to 1500 octets in size [3]. The data field contains the *Network Layer* packet or raw data, depending on the value of the *Length/Type* field. The final field in the ethernet frame is the *Frame Check Sequence (FCS)* which contains a checksum to check for errors in transmission.

### 5.1.1 Internet Protocol (IP) Packet Structure

Internet Protocol (IP) is one protocol that is used in the third layer of ethernet communication, the *Network Layer*. It is the primary mechanism for communicating arbitrary data in this design. Figure 5.2 shows the IP packet structure. The data in the IP packet diagram is referenced in bits.

The first four bits of an IP packet identify which version of the Internet Protocol packet. In this design, version four packets are used exclusively. The next four bits of the header identify the number of 32 bit words in the header. In this design, all IP packet headers have five 32 bit words. The following eight bits identify the *Type of Service*, which communicates the priority of the packet. Bits 16 to 31 are the number of bytes in the entire packet. The second 32 bit word, starts with an eight

bit identification of the packet, if it is part of a sequence of packets. The remainder of the 32 bit word is used for flags.

IP Version	Header Length	Type of Service	Total Length	ID	Flags	Time to Live	Protocol	Header Checksum	Source IP	Destination IP	IP Options	Data
0	3 4	7 8	15 16	31	48	63 64	71 72	79 80	95 96	127 128	160 161	191 192

Figure 5.2: Structure of Internet Protocol Packet

Starting at the sixty-fourth bit, the third 32 bit word, is the eight bit *Time to Live* field, which decrements each time the packet is relayed. When the counter reaches zero, the packet is discarded. An eight bit protocol field follows, identifying the protocol contained in the data. Then a sixteen bit checksum for the header completes the 32 bit word. The 32 bit source IP address and the 32 bit destination IP address are the fourth and fifth word in the header. A sixth 32 bit word is available for IP options, but is not used in this design. The remainder of the IP packet is data which it contains.

### 5.1.2 ARP

In order for the ethernet connection on the DREAM board to interact with other systems for IP communications. The DREAM board must be associated with an IP address. Since the IP address is not known, another *Network Layer* protocol must be used. The manor in which address assignment is accomplished is through an exchange of Address Resolution Protocol (ARP) packets with other connected systems. The structure of an ARP packet is seen in Figure 5.3.

Hardware Type	Protocol Type	Hardware Address Length	Protocol Address Length	Operation	Sender Hardware Address	Sender Protocol Address	Target Hardware Address	Target Protocol Address
0 15	16 31	32 39	40 47	48 63	64 111	112 143	144 192	193 223

Figure 5.3: Structure of ARP Packet

The ARP packet structure is 224 bits long. The first sixteen bits identify the *Hardware Type* being used. In this case, this is ethernet. The second sixteen bits are the *Protocol Type*. This is used to identify which protocol type is being referenced. The next eight bits identifies the length of the hardware address. In this case, 48 bits for the MAC address.

This is followed by eight bits to identify the length of the protocol address. In this design, this is 32 bits for the IP address. Eight bits follow to identify the *Operation* being executed. Starting at the sixty-fourth bit is the 48 bit MAC address of the sender, followed by the 32 bit IP address. Finally, the 48 bit MAC address and the 32 bit IP address of the target of the packet.

### 5.1.3 ICMP

It is desired to communicate control messages as well. A ping exchange is one example of a control message and is accomplished through the use of the Internet Control Message Protocol (ICMP) protocol. ICMP is part of the IP suite [3]. As such, an ICMP packet occurs within an IP packet, starting after the fifth 32 bit word, discussed in the previous section. The ICMP packets in this design are used specifically for echo type packets. The structure of an ICMP echo packet is seen in Figure 5.4.



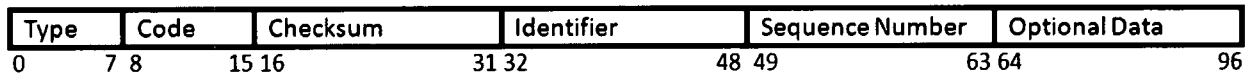


Figure 5.4: Structure of ICMP Echo Packet

The first eight bits of the ICMP packet identify the type of control message is being communicated. An eight bit code identifies any problems that may have been encountered during transmission. This is followed by a sixteen bit checksum. A sixteen bit identifier and a sixteen bit sequence number are generated by the system that forms the packet. This is to allow comparison of sent and received packets. This is used, in particular, during a ping exchanged, which is discussed in the following sections.

## 5.2 Embedded Software

In order to facilitate communication between the design and a general purpose computer, data packets must be formed correctly. Also, certain packets must be generated by the DREAM board to initialize the data transfer and certain packets broadcast by the general purpose computer must be responded to correctly.

### 5.2.1 Communication Path Initialization

As previously noted, in order to communicate using IP the DREAM board must have an IP address. Software is written for the embedded PowerPC on the DREAM board to use an ARP packet which allows the DREAM board to identify itself by an IP address. This is accomplished by broadcasting an ARP packet from the DREAM board.

The packet is broadcast to all connections on the network by using the MAC address FF:FF:FF:FF:FF, an address reserved for broadcast [3]. This packet identifies the operation as being a *Gratuitous ARP Request* for an IP address. The desired IP address is identified in bits 112 to 143 of the packet. (Refer to Figure 5.3.) The use of this section of the design is found in Chapter 6 and the embedded C source code for this behavior is found in Appendix C.

### 5.2.2 Ping Exchange

The ping exchange is a basic method of verifying a channel of communication between two IP addresses. Because IP address are used, a ping can only occur after the DREAM board identifies itself with an IP address. Therefore, it must occur after the communication path, described in the previous section, has completed.

Ping requests and ping replies occur through the use of ICMP echo packets, described in Section 5.1. The echo type of ICMP packet is used. Type 8 is an *Echo Request* packet, which is generated for a ping request, and Type 0 is an *Echo Reply*, which is generated for a ping reply [3].

Once the IP address of the DREAM board has been established, a ping packet, contained within an IP packet, can be generated to respond to ping requests. Software is written for the embedded PowerPC which generates this response. To generate a ping response packet, the DREAM board must identify a ping request packet. This packet contains the identifier and sequence number of the ping request. Responding to the request involves changing the type of echo packet from a request to a reply, copying the unique identifiers, and recalculating the checksum.

In the IP layer of the ping response, the source IP address and the destination IP address are swapped and the header checksum is recalculated. The use of this section of the design is found in Chapter 6 and the embedded C source code for this behavior is found in Appendix C.

### **5.2.3 Data Exchange**

The IP packet structure is used to transmit arbitrary data from the DREAM board to a general purpose computer in this design. The *Type of Service* field is set to raw data and data is inserted into the data field. The use of this section of the design is found in Chapter 6 and the embedded C source code for this behavior is found in Appendix C.

## **5.3 Third-Party Software**

The most appealing aspect of using a general purpose computer is the fact that, for most tasks, software has been developed by a third-party. For this design, the general purpose computer uses Windows XP operating system.

### **5.3.1 Windows XP Network Interface Software**

The Windows operating system has extensive network communication capability. Unfortunately, the methods and mechanisms of Windows ethernet communications are not user controllable. There are, however, useful tools for use with Windows XP, which allow for a more detailed interaction with the network of the computer.

### 5.3.2 WireShark

In order to analyze the data traffic occurring on the ethernet connection, a piece of commercial software, by the name of WireShark, is used. WireShark is an open source network protocol analyzer. Each ethernet packet on the network controller of the general purpose computer, including those generated by the computer, is displayed by the WireShark program.

### 5.3.3 Ethernet Traffic

An example of the traffic experienced on the ethernet connection is seen in Figure 5.5. The Figure shows the activity on the channel when the DREAM board is connected to the general purpose computer during initial communication. It is seen in the Figure that, upon connection, the computer transmits an ARP packet. The second, third, and fourth line show the general purpose computer initiating a *Gratuitous ARP Request* for the IP address 169.254.226.21. There is no response from other computers on the network. The source, shown in WireShark, is then changed to this IP address for further transactions.

Following the request of IP address, there is more activity by the computer. These exchanges are the Windows XP computer identifying itself to the network [3].

## 5.4 General Purpose Computer Software

A simple C# program is designed to provide and demonstrate the ethernet communication link between the DREAM board and the general purpose computer. The interface window is seen in Figure 5.6. There is a text window which is used to display the status of the TestBed and small amounts of data. There is also a button to send

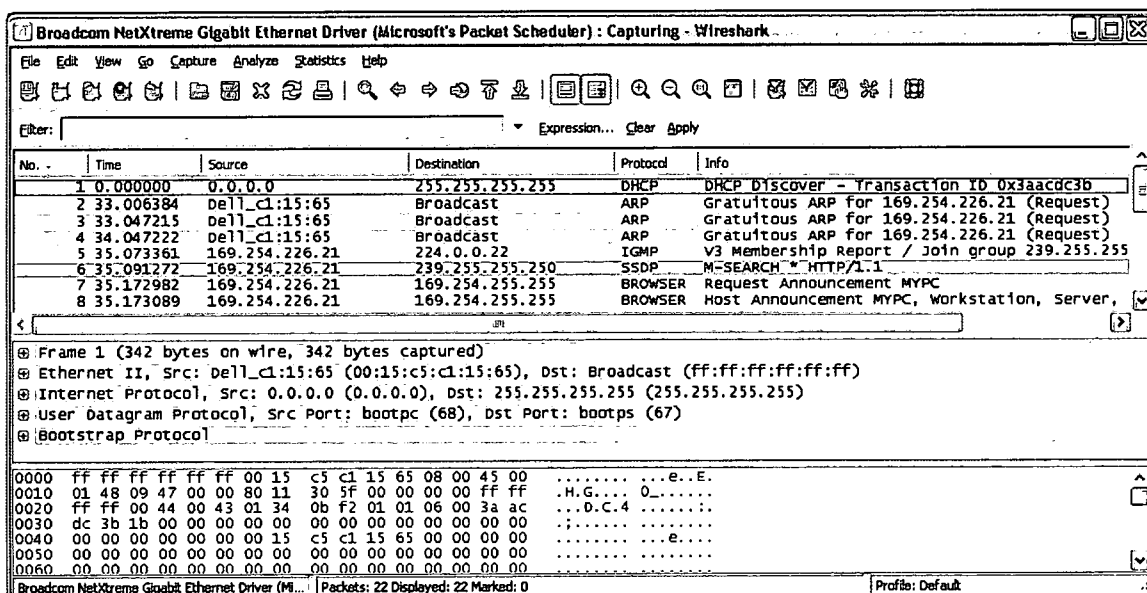


Figure 5.5: WireShark View of Ethernet Controller Traffic

a *send* command to the DREAM board. The DREAM TestBed also provides this *send* command to the board and then waits to receive data. This is discussed in the following section. The source code for this simple TestBed is found in Appendix D.

### 5.4.1 Data Transfer

Another function of the DREAM TestBed application is to initiate data transfer between the design and the general purpose computer. This is done by means of the TestBed transmitting a packet of data containing a *send code*. This *send code*, when received by the DREAM board initiates the transfer of test data to the computer. The data is configurable and is set to be an incremented sequence of 1024 bytes starting at zero. The data set is copied 1024 times forming a 1 megabyte packet. This packet is sent to the computer 1024 times.

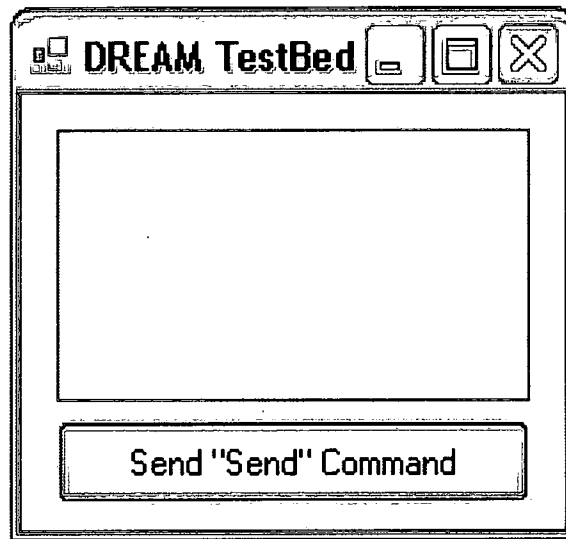


Figure 5.6: DREAM Board TestBed

The computer, expecting the data, counts the received packets. When the expected number of packets are received, the TestBed returns to the state where it is ready to send another *send data* signal.

## CHAPTER 6

### Results

The following sections demonstrate different aspects of the design. Section 6.1 discusses the results of the Pulse Receptor Core design on the DREAM board. Section 6.2 discusses the results of the Ethernet design on the DREAM board.

#### 6.1 Pulse Receptor Core Usage

The Pulse Receptor core, described in Chapter 3, is demonstrated in the following sections. The behavior of the Pulse Receptor core on the DREAM board is controlled through a HyperTerminal interface, described in the following sections.

##### 6.1.1 Global Time of Arrival

The Global Time of Arrival (TOA) Generator is designed to maintain the number of clock cycles which have passed since the beginning of operation. Figure 6.1 shows the output of fourteen lowest bits of the 38 bit TOA output, as well as the 10MHz signal, located at the bottom of the Figure, which is used to drive the TOA Generator.

Along the bottom of Figure 6.1 are measurements made by the oscilloscope. On the left, labeled  $\text{Freq}(D_0)$ , is the frequency of the lowest bit of the TOA Generator. The frequency, 25MHz is the expected value since each half of the cycle represents one cycle of the 50MHz clock. In the center, labeled  $\text{Freq}(D_2)$ , is the third lowest bit

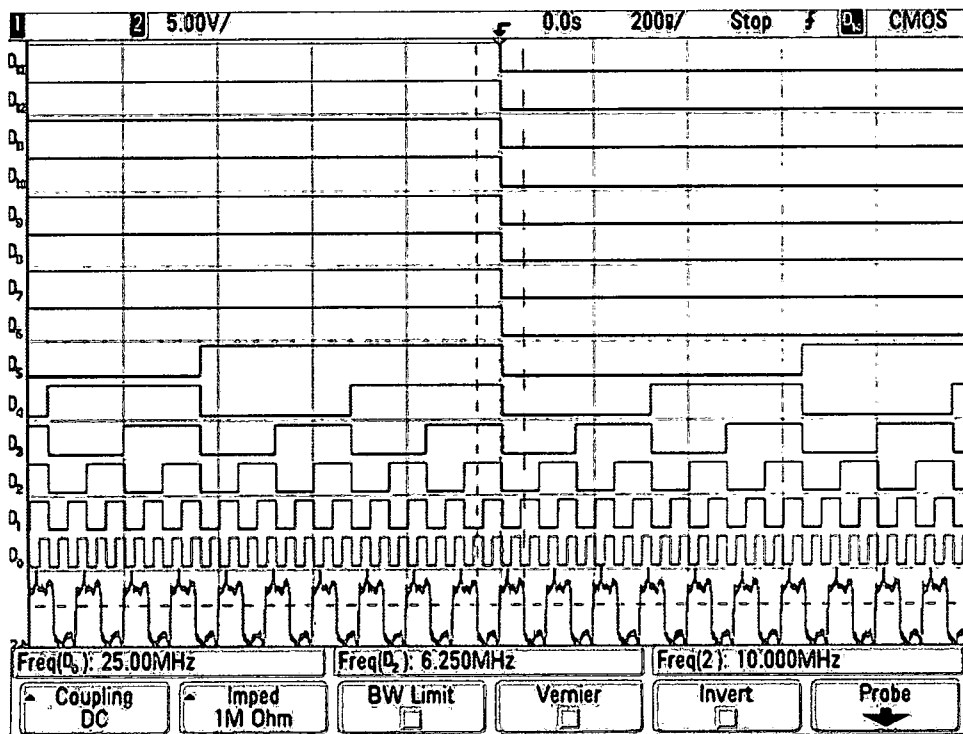


Figure 6.1: Global Time of Arrival Generator Experimental Measurement

of the TOA generator and is 6.25MHz, as expected. Finally, labeled Freq(2) on the left, is a frequency measurement of the input 10MHz clock signal.

### 6.1.2 DREAM Machine Test Set

For development purposes, a signal generator is provided to provide signals to the DREAM board which simulate possible RF sensor signals. This test set, shown in Figure 6.2, generates eight channels of simulated eight bit sensor data along with the associated *Pulse Detected Signals*. Each channel of the signal generator can be activated separately and is configurable. The pulse width of each channel and the repetition rate is set on the front of the test set.



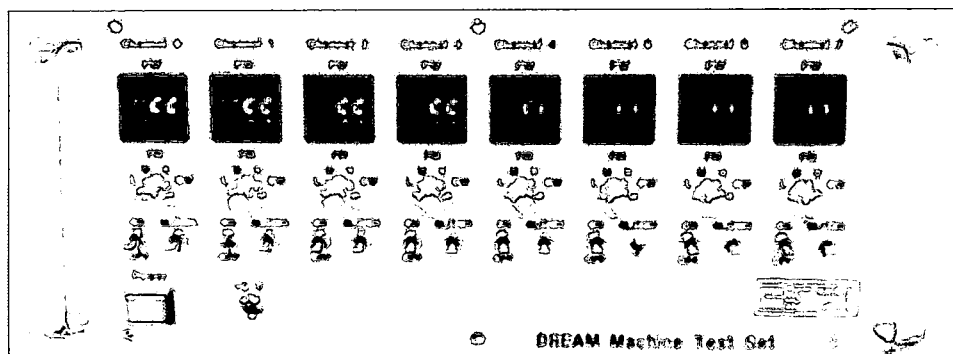


Figure 6.2: DREAM Machine Test Set

### 6.1.3 Reception of Pulse

To demonstrate the operation of the PRC Data Collection core, a pulse is generated by the DREAM Machine Test Set. Channel 1 is shown in Figure 6.3. The pulse on Channel 1 is set to generate a pulse with a duration of  $AC$ , the hexadecimal representation of 172.  $(AC)_{16} = (172)_{10}$ .

Once the pulse has been collected, the width, amplitude, and time of arrival of the pulse is collected by the PowerPC. Figure 6.4 shows the output of the pulse data from the PowerPC to the serial connection on the general purpose computer. It is seen that the pulse reported by the DREAM board is of the pulse width set by the Dream Machine Test Set.

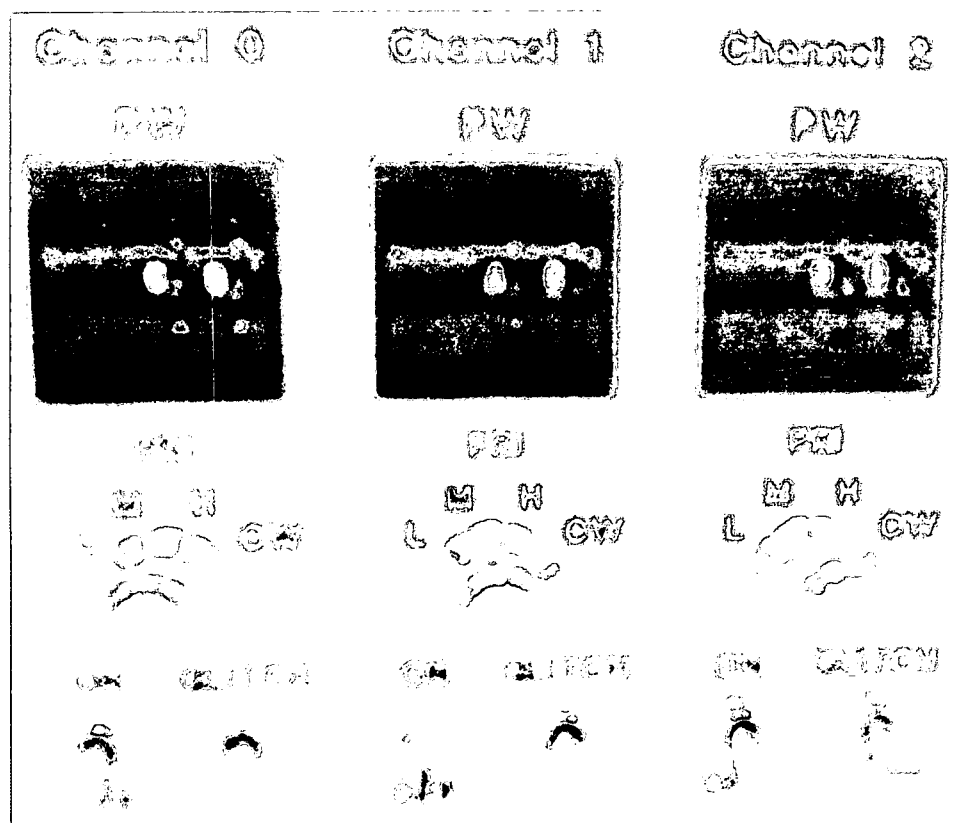


Figure 6.3: DREAM Test Set with Channel 1 Set to a Pulse Width of AC

To demonstrate the pulse is being measured correctly, the Test Set is changed to generate a pulse of width 78, the hexadecimal representation of 120.  $(78)_{16} = (120)_{10}$ . This setting is seen in Figure 6.5 and the corresponding reporting by the PowerPC to the computer is seen in Figure 6.6.

```

DREAM - HyperTerminal
File Edit View Ctrl Transfer Help

Running GpioOutputExample() for LEDs_8Bit...
GpioOutputExample PASSED.
*****
* User Peripheral Self Test
*****

RST/MIR test...
- write 0x0000000A to software reset register
- read 0x30220301 (expected) from module identification register
- RST/MIR write/read passed

User logic slave module test...
- write 1 to slave register 0
- read 0 from register 0
- slave register 0 write/read failed
Ch1=> Amplitude = FF      Pulse Width = AC      TOA = 072C2312E  valid = 1
Ch1=> Amplitude = 4       Pulse Width = AC      TOA = 0730EE44F  valid = 1
Ch1=> Amplitude = 01      Pulse Width = AC      TOA = 0734FC321  valid = 1
Ch1=> Amplitude = 43      Pulse Width = AC      TOA = 073968CF3  valid = 1
Ch1=> Amplitude = CD      Pulse Width = AC      TOA = 073EA4E71  valid = 1
Ch1=> Amplitude = 6E      Pulse Width = AC      TOA = 07425D5A5  valid = 1
Ch1=> Amplitude = 38      Pulse Width = AC      TOA = 07458345B  valid = 1

Connected 2:22:40  Auto detect  9600 8-N-1  SCROLL  CAPS  NUM  Capture  Print Echo

```

Figure 6.4: HyperTerminal Display of DREAM Board Receiving a Pulse on Channel 1 with a Width of AC

The results shown in Figure 6.4 and Figure 6.6 are representative of the results experienced when generating pulses on the other channels.

### 6.1.4 Generation of Control Word

The desired Control Word is entered into the DREAM board by using HyperTerminal to access the serial connection. Figure 6.7 shows the HyperTerminal interface used to set the command word being generated by the Control Word Generator. It is seen that the 32bit code word which is generated is  $(12458635)_{16}$ .

To demonstrate the operation of the Control Word Generation portion of the PRC core, an oscilloscope is connected to the GPIO pins to display the control word and valid signal. The eight most significant bits and the eight least significant bits of the

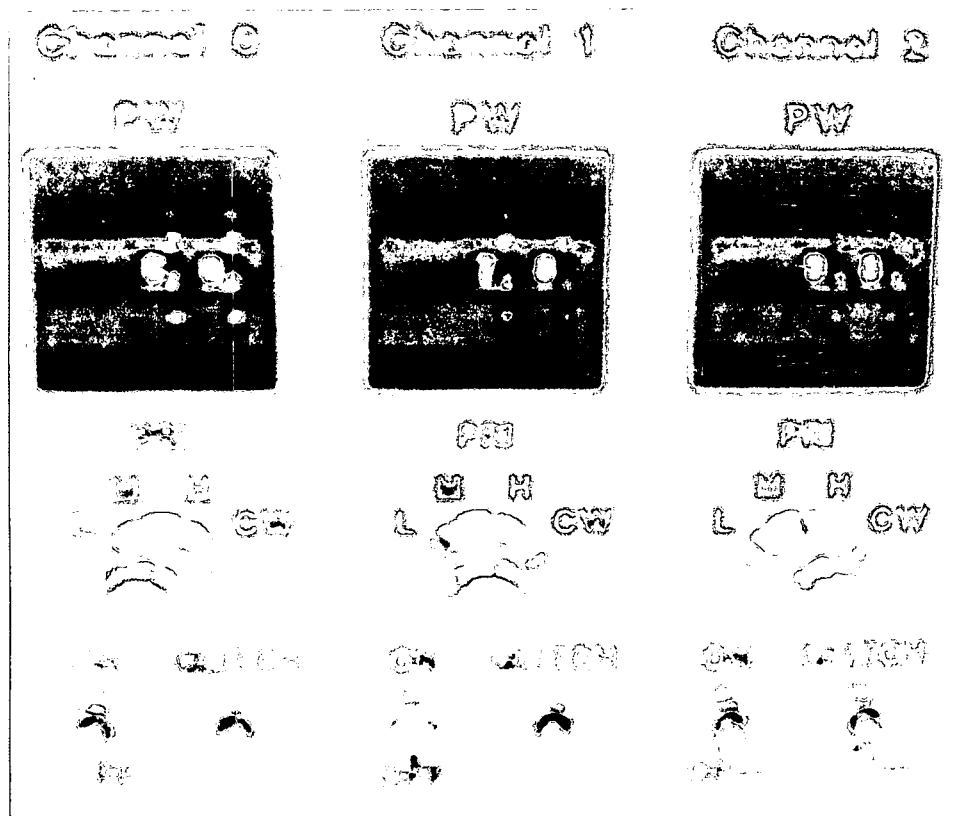


Figure 6.5: DREAM Test Set with Channel 1 Set to a Pulse Width of 78

32 bit control word and the *Data Valid* signal are measured by the oscilloscope, as seen in Figure 6.8. Along the bottom of the oscilloscope display, in the Figure, is the hexadecimal value of the sixteen data lines. The measurement labeled X2(HEX) shows that the hexadecimal value 1235 is present, as expected. Also along the bottom of the display is the duration of the signals. The *Control Word* is present for 1us and the *Data Valid* signal is active for approximately 150ns. This is in accordance with the specification for the design

```

DREAM - Hyperterminal
File Edit View Call Transfer Help
[Icons]

- write 0x0000000A to software reset register
- read 0x30220301 (expected) from module identification register
- RST/MIR write/read passed

User logic slave module test...
- write 1 to slave register 0
- read 0 from register 0
- slave register 0 write/read failed

Ch1=> Amplitude = FF Pulse Width = AC TOA = 072C2312E valid = 1
Ch1=> Amplitude = 4 Pulse Width = AC TOA = 0730EE44F valid = 1
Ch1=> Amplitude = D1 Pulse Width = AC TOA = 0734FC321 valid = 1
Ch1=> Amplitude = 43 Pulse Width = AC TOA = 073968CF3 valid = 1
Ch1=> Amplitude = CD Pulse Width = AC TOA = 073EA4E71 valid = 1
Ch1=> Amplitude = 6E Pulse Width = AC TOA = 07425D5A5 valid = 1
Ch1=> Amplitude = 38 Pulse Width = AC TOA = 07458345B valid = 1
Ch1=> Amplitude = D3 Pulse Width = 78 TOA = 3FBDEF88F valid = 1
Ch1=> Amplitude = AD Pulse Width = 78 TOA = 3FC541E87 valid = 1
Ch1=> Amplitude = 7A Pulse Width = 78 TOA = 3FC95C0B9 valid = 1
Ch1=> Amplitude = 7B Pulse Width = 78 TOA = 3FCE08B6A valid = 1
Ch1=> Amplitude = 82 Pulse Width = 78 TOA = 3FD20452C valid = 1
Ch1=> Amplitude = 6F Pulse Width = 78 TOA = 3FD786AEA valid = 1
Ch1=> Amplitude = 85 Pulse Width = 78 TOA = 3FDC9B249 valid = 1
Ch1=> Amplitude = 3 Pulse Width = 78 TOA = 3FDF34A31 valid = 1

Connected 2:24:12 [Auto detect] [9600 8-N-1] [SCROLL] [CAPS] [NUM] [Capture] [Print echo]

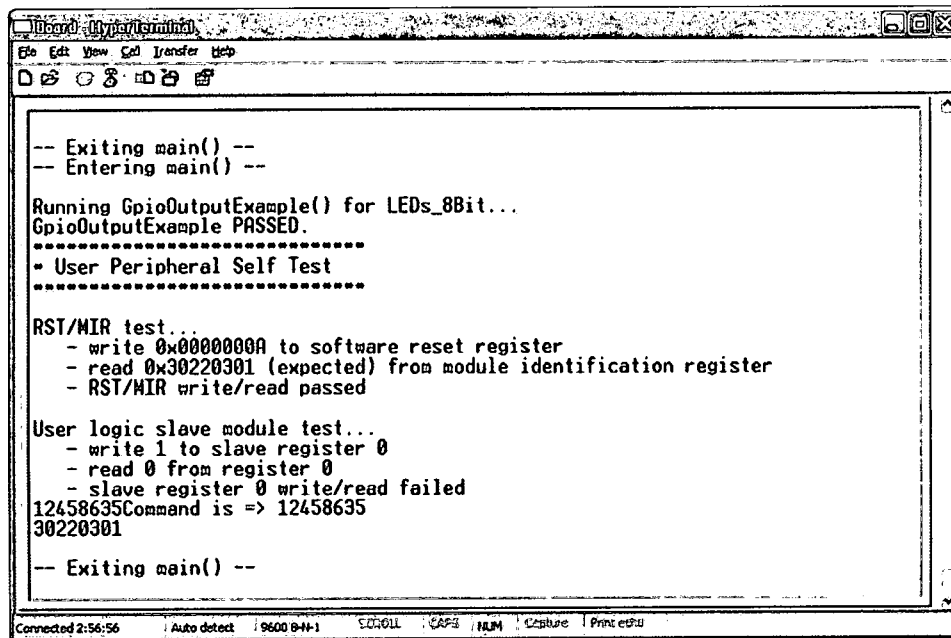
```

Figure 6.6: HyperTerminal Display of DREAM Board Receiving a Pulse on Channel 1 with a Width of 78

It is demonstrated that the requirements for the data collection have been achieved in this design. Also, the requirement for the generation of control data has also been achieved.

## 6.2 Ethernet System Demonstration

The gigabit ethernet hardware, described in Chapter 4, is demonstrated through the use of the ethernet software discussed in Chapter 5. The behavior of the ethernet design on the DREAM board is controlled through a HyperTerminal interface. The software that runs the PowerPC initializes the gigabit ethernet subsystem. Figure 6.9 shows the output of the DREAM board during initialization. After the the software



```
-- Exiting main() --
-- Entering main() --

Running GpioOutputExample() for LEDs_8Bit...
GpioOutputExample PASSED.
-----
* User Peripheral Self Test
-----

RST/MIR test...
- write 0x0000000A to software reset register
- read 0x30220301 (expected) from module identification register
- RST/MIR write/read passed

User logic slave module test...
- write 1 to slave register 0
- read 0 from register 0
- slave register 0 write/read failed
12458635Command is => 12458635
30220301

-- Exiting main() --
```

Figure 6.7: HyperTerminal Interface for PRC Control Word Generation

executes a memory test, the SGMII is configured to be a 4 pin interface with autonegotiation enabled. The registers in the gigabit ethernet PHY channels are then set. In the Figure, the PHY channel that is being addressed is identified along with the specific register and the value to which the register is being set. PHY Channel is the ethernet channel which is used in this demonstration. PHY Channel 6 is the SGMII to GMII bridge.

### 6.2.1 Ethernet Packet Handling

After the DREAM board has been initialized, the capturing of an ethernet packet, which is the starting point of parsing the packet is demonstrated. This is seen in Figure 6.10. The organization of the captured ethernet packet is described in Chapter 5.

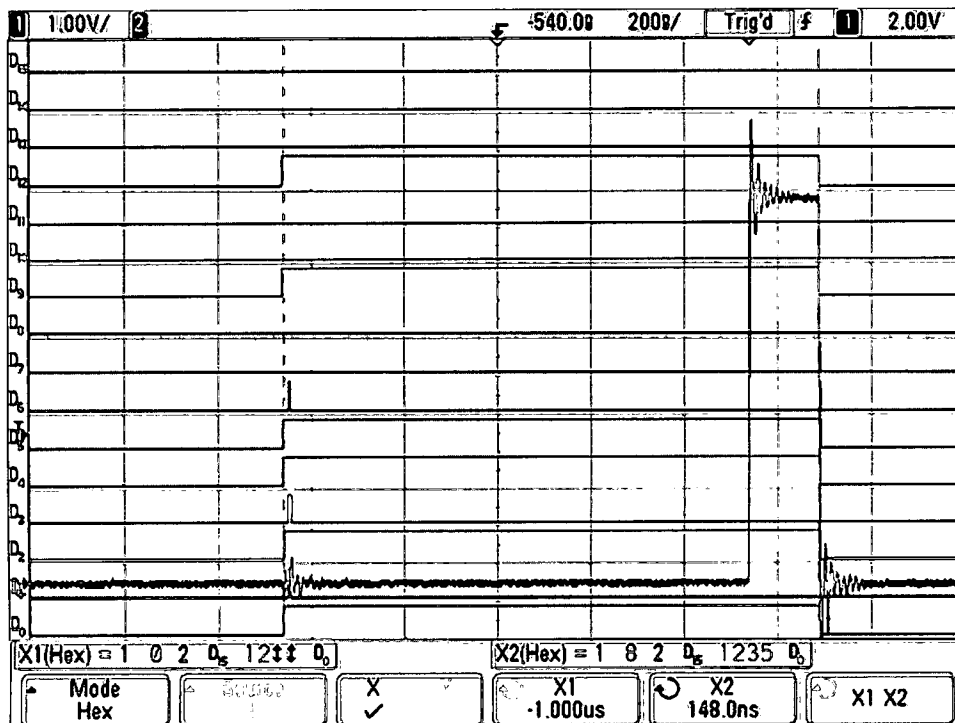


Figure 6.8: PRC Channel Data Collection Experimental Measurement

Parsing the captured ethernet packet is then demonstrated. This is seen in Figure 6.11. Besides the source address and destination address, the type of ethernet packet is shown as type 80 which identifies the packet as an internet protocol packet. Also, CRC-32 information is presented.

### 6.2.2 ARP Packet Exchange

In order for the desktop computer to be able to address the DREAM board, the DREAM board must respond to an ARP request from the computer to identify itself. Figure 6.12 shows the exchange which occurs. Line 1 and line 2 show the computer requesting to address itself as 169.254.162.21. Line 11 shows the broadcast from

```

-- Entering main() --
Starting MemoryTest for plb_bram_if_cntlr_1:
Running 32-bit test...PASSED!
Running 16-bit test...PASSED!
Running 8-bit test...PASSED!
SGWII 4 pin interface with autonegotiation enabled
PHY Channel: 0x00000000 PHY Register: 0x00000017 is now set to 0x0000a002
PHY Channel: 0x00000000 PHY Register: 0x00000000 is now set to 0x00009000
PHY Channel: 0x00000000 PHY Register: 0x0000001b is now set to 0x00000300
PHY Channel: 0x00000000 PHY Register: 0x0000001f is now set to 0x00000001
PHY Channel: 0x00000000 PHY Register: 0x00000013 is now set to 0x00000002
PHY Channel: 0x00000000 PHY Register: 0x0000001f is now set to 0x00000000
PHY Channel: 0x00000001 PHY Register: 0x00000017 is now set to 0x0000a00a
PHY Channel: 0x00000001 PHY Register: 0x00000000 is now set to 0x00009040
PHY Channel: 0x00000001 PHY Register: 0x0000001b is now set to 0x00000300
PHY Channel: 0x00000006 PHY Register: 0x00000000 is now set to 0x00001140

Starting GEMAC
Successfully Started GENAC
0 - Locked in Ping Response Mode
1 - Print Next Received Packet
2 - Parse Ethernet Packet
3 - APR Request
Command: _

```

Figure 6.9: HyperTerminal Display of DREAM Board Initializing

the computer asking *Who has 169.254.162.10*. Line 13 shows the DREAM board answering the request with a *Gratuitous ARP for 169.254.162.10 (Request)*.

### 6.2.3 Ping Exchange

Figure 6.13 shows a ping request being initiated by the general purpose computer and the time it takes for the DREAM board to return the ping.

Figure 6.14 shows the packets which are generated by the general purpose computer during the ping request and the packets which are returned by the DREAM board. Lines 1, 3, 5, and 7 show the ping request being generated by the computer and lines 2, 4, 6, and 8 show the reply from the DREAM board to the ping requests.





```
DREAM - HyperTerminal
File Edit View Go Transfer Help
[Icons]

PHY Channel: 0x00000000 PHY Register: 0x0000001f is now set to 0x00000000
PHY Channel: 0x00000001 PHY Register: 0x00000017 is now set to 0x0000a00a
PHY Channel: 0x00000001 PHY Register: 0x00000000 is now set to 0x00009040
PHY Channel: 0x00000001 PHY Register: 0x0000001b is now set to 0x00003000
PHY Channel: 0x00000006 PHY Register: 0x00000000 is now set to 0x00001140

Starting GENMAC
Successfully Started GENMAC
0 - Locked in Ping Response Mode
1 - Print Next Received Packet
2 - Parse Ethernet Packet
3 - APR Request
Command: 2*
Destination MAC address: FFFFFFFF
Source MAC address: 015C5C11565
Type/Length: 80

CRC-32: E6FBFE8
0 - Locked in Ping Response Mode
1 - Print Next Received Packet
2 - Parse Ethernet Packet
3 - APR Request
Command: *
```

Figure 6.11: HyperTerminal Display of DREAM Board Parsing an Ethernet Packet

the packets, as seen in Figure 6.16. It is seen in the Figure that all 1024 packets are received.

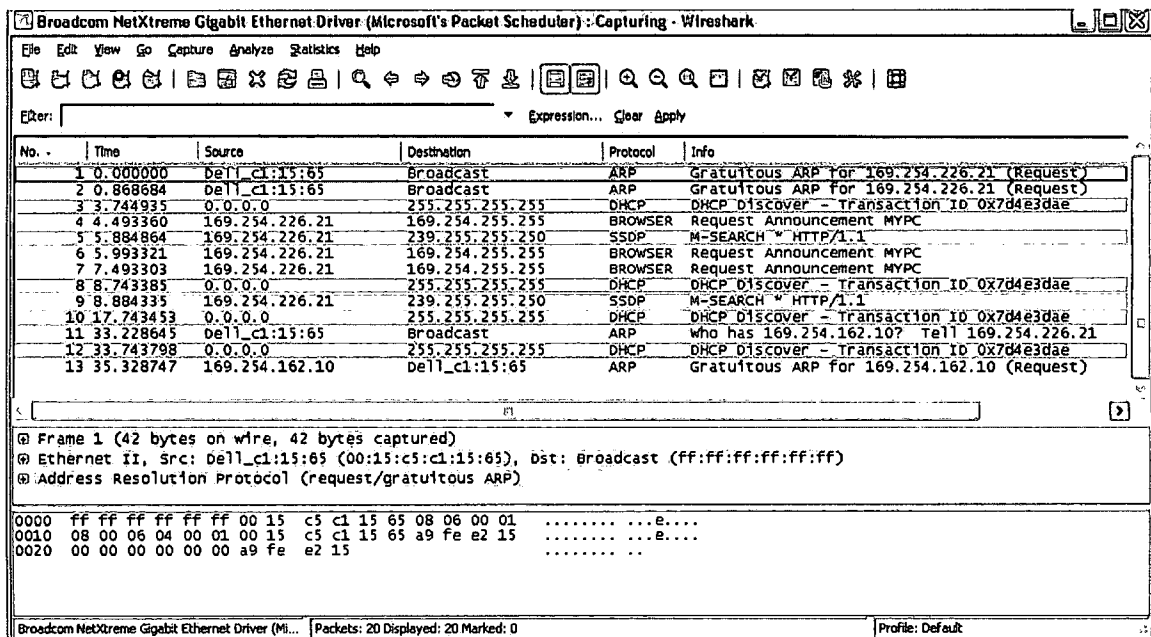


Figure 6.12: WireShark Display of General Purpose Computer and DREAM Board ARP Exchange

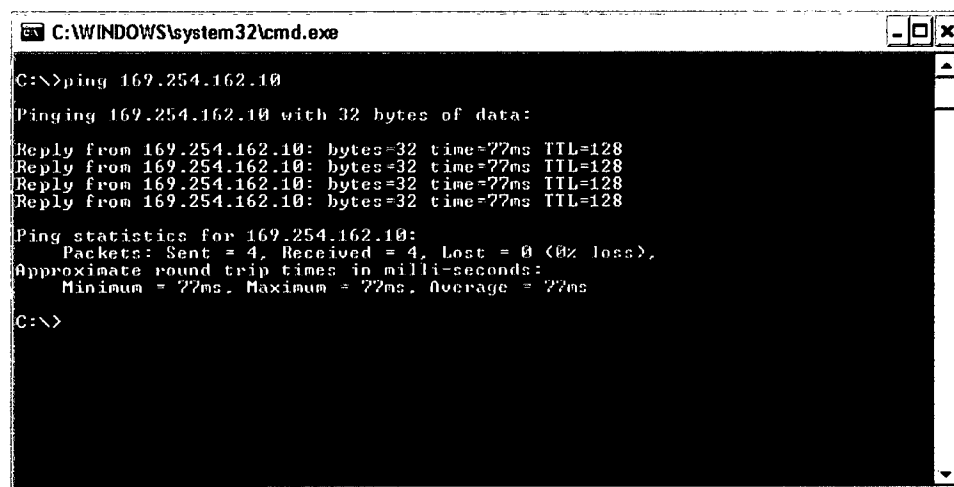


Figure 6.13: Command Terminal Display of General Purpose Computer Executing a Ping Command





Figure 6.16: DREAM TestBed After Receiving Data From DREAM Board

## CHAPTER 7

### Conclusions & Future Work

In this thesis, an RF signal processing system is designed. The system collects high speed parallel data and serialize it for collection by a PowerPC. The system also transmits serialized data to a general purpose computer using a gigabit ethernet connection.

The system collects RF sensor pulse data from eight 8-bit sensors and characterizes the pulse. This is accomplished through the use of an FPGA based data collection design. This system also serializes the data, through the use of a multiplexer, to the PowerPC on the FPGA. The PowerPC sends serialized data through the ethernet connection to a desktop computer.

However, before designing the data collection portion of the design, electrical aspects of the DREAM board are addressed. These aspects of DREAM board design include noise on the power planes and the location and the spacing of components and interconnections.

In the thesis, a gigabit ethernet communications core is also designed. The core is designed to transfer data from a PowerPC on the FPGA to a general purpose computer. This is accomplished through the placement and connection of a Vitesse Gigabit ethernet PHY. Once the electrical connections are made, the communication

methods are developed. This includes ethernet frames, IP packets, ARP packets, and ping exchanges.

## 7.1 Future Work

The completion of all objectives for this project is not completed by this design. The transfer of data between the Pulse Receptor Data Collection Core and the ethernet core is not complete. There are three areas which need to be further developed to accomplish the end goal.

The first area is the design of software to interface efficiently between the PowerPC and the ethernet controller. This interface needs to accommodate any type of ethernet packet received and react accordingly. A method to accomplish this is through the use of a light weight operating system operating on the DREAM board, such as Linux.

However, an on-board operating system requires more memory then is available on the VirtexII FPGA. This leads to the the second area which needs to be addressed, off-chip memory. There is a DDR2 memory module available on the DREAM board. This current design does not utilize this memory. The off-chip memory would allow for the use of a basic operating system on the DREAM board. A Linux operating system could be configured to provide the necessary ethernet packet handling mechanism needed to communicate with a general purpose computer.

After an operating system is operational on the DREAM board with connectivity to the ethernet controller, the third and final area is designing the software required to transfer data between the Pulse Receptor Core and the ethernet. Along with the Pulse Receptor Core and associated controller, an embedded program for the

operating system can be developed which transmits the Pulse Receptor Core packets to a general purpose computer.



## **APPENDIX A**

### **MATLAB code**

## IDAL Bypass2.m

```
%%% IDAL PC Board Bypass layout
close all; clear all

frequency_points = 1001;
f=logspace(0,12,frequency_points);
w=2*pi.*f;

L_Via = 1000e-12;
L_Plane = 205e-12;

C1_Qty = 4;
C2_Qty = 14;
C3_Qty = 27;
C4_Qty = 55;

%% choose the first bank RLC parameters
C1 = 470e-6;
ESL_Tant = 3200e-12;
ESR_Tant = 0.12;

L1 = ESL_Tant + 2*L_Via + L_Plane;

Rs = ESR_Tant;
Xl = L1*w;
Xc = 1./(w*C1);
z1_prime = sqrt(Rs^2 + (Xl - Xc).^2);

figure
loglog(f,Xl,'k',f,Xc,'k',f,Rs,'k');hold on;
loglog(f,z1_prime,'k','linewidth',2.5);hold off;
xlabel('Log Frequency (Hz)');ylabel('Impedance (\Omega)');
axis([0 1e12 10e-4 10e3])

%% choose the second bank RLC parameters
C2 = 2.2e-6;
ESL_0805 = 2300e-12;
ESR_0805 = .0051;

L2 = ESL_0805 + 2*L_Via + L_Plane;

Rs = ESR_0805;
Xl = L2*w;
Xc = 1./(w*C2);
z2_prime = sqrt(Rs^2 + (Xl - Xc).^2);

figure
loglog(f,Xl,'k',f,Xc,'k',f,Rs,'k');hold on;
loglog(f,z2_prime,'k','linewidth',2.5);hold off;
xlabel('Log Frequency (Hz)');ylabel('Impedance (\Omega)');
axis([0 1e12 10e-4 10e3])
```

```

%% choose the third bank RLC parameters
C3 = 0.1e-6;
ESL_0603 = 1990e-12;
ESR_0603 = 0.101;
L3 = ESL_0603 + 2*L_Via + L_Plane;

Rs = ESR_0603;
Xl = L3*w;
Xc = 1./(w*C3);
z3_prime = sqrt(Rs^2 + (Xl - Xc).^2);

%Display
figure
loglog(f,Xl,'k',f,Xc,'k',f,Rs,'k');hold on;
loglog(f,z3_prime,'k','linewidth',2.5);hold off;
xlabel('Log Frequency (Hz)');ylabel('Impedance (\Omega)');
axis([0 1e12 10e-4 10e3])

%% choose the fourth bank RLC parameters
C4 = 0.01e-6;
ESL_0402 = 990e-12;
ESR_0402 = 0.634;
L4 = ESL_0402 + 2*L_Via + L_Plane;

Rs = ESR_0402;
Xl = L4*w;
Xc = 1./(w*C4);
z4_prime = sqrt(Rs^2 + (Xl - Xc).^2);

figure
loglog(f,abs(Xl),'k',f,abs(Xc),'k',f,Rs,'k');hold on;
loglog(f,z4_prime,'k','linewidth',2.5);hold off;
xlabel('Log Frequency (Hz)');ylabel('Impedance (\Omega)');
axis([0 1e12 10e-4 10e3])

%% Calculate the Overall Impedance
z1 = z1_prime./C1_Qty;
z2 = z2_prime./C2_Qty;
z3 = z3_prime./C3_Qty;
z4 = z4_prime./C4_Qty;
z_Total = 1./(1./z1 + 1./z2 + 1./z3 + 1./z4);

figure
loglog(f,z1,'k-.',f,z2,'k-.',f,z3,'k-.',f,z4,'k-.');
hold on
loglog(f,abs(z_Total),'k','LineWidth',2.5)
xlabel('Log Frequency (Hz)');ylabel('Impedance (\Omega)');
%title('Overall Impedance Vs. Frequency')
axis([0 1e12 10e-4 10e3])

```



## idealStep.m

```
%idealStep.m
%calculates and displays the frequency response of a step funtion

close all;
clear all;

stepSize = 1e-9; %2 nanosecond stepsize corresponds to 500MHz
duration = 2e-6; %1 microsecond

t=0:stepSize:duration; %time scale

%forming step function with a transition equal to the step size
x=ones(1,length(t));
x(1:(length(x)-1)/2)=zeros(1,(length(x)-1)/2);

%resolution of Frequency plot
resolution=2^16;
f=linspace(-1./stepSize/2,1./stepSize/2,resolution);

%calculate the frequency response
X = fftshift(fft(x,resolution));

%Plot the frequency response
plot(f/1e6,X)
xlabel('Frequency (MHz)')
ylabel('Magnitude')
axis([-100 100 -50 50])
```

## APPENDIX B

### HDL Code for PRC Module

## PRC user logic.vhd

```

-----
-- user_logic.vhd - entity/architecture pair
-----

--
-- *****
-- ** Copyright (c) 1995-2006 Xilinx, Inc. All rights reserved. **
-- **                                                                 **
-- ** Xilinx, Inc.                                                                 **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND **
-- ** SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT, **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE **
-- ** FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS **
-- ** FOR A PARTICULAR PURPOSE. **
-- **                                                                 **
-- *****
--

-----
-- Filename:          user_logic.vhd
-- Version:           1.00.a
-- Description:       User logic.
-- Date:             Mon Mar 31 11:48:38 2008 (by Create and Import Peripheral Wizard)
-- VHDL Standard:    VHDL'93
-----

-- Naming Conventions:
--   active low signals:          "*_n"
--   clock signals:              "clk", "clk_div#", "clk_#x"
--   reset signals:              "rst", "rst_n"
--   generics:                   "C_#"
--   user defined types:         "*_TYPE"
--   state machine next state:   "*_ns"
--   state machine current state: "*_cs"
--   combinatorial signals:      "*_com"
--   pipelined or register delay signals: "*_d#"
--   counter signals:            "*cnt*"
--   clock enable signals:       "*_ce"
--   internal version of output port: "*_i"
--   device pins:                "*_pin"
--   ports:                      "- Names begin with Uppercase"
--   processes:                  "*_PROCESS"
--   component instantiations:   "<ENTITY_>I_<#|FUNC>"
-----

```

```

-- DO NOT EDIT BELOW THIS LINE -----
library ieee;
Library UNISIM;
library proc_common_v2_00_a;

use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use UNISIM.vcomponents.all;
use proc_common_v2_00_a.proc_common_pkg.all;
-- DO NOT EDIT ABOVE THIS LINE -----

--USER libraries added here

-----
-- Entity section
-----
-- Definition of Generics:
--      C_DWIDTH          -- User logic data bus width
--      C_NUM_CE          -- User logic chip enable bus width
--      C_IP_INTR_NUM     -- User logic number of interrupt event
--
-- Definition of Ports:
--      Bus2IP_Clk        -- Bus to IP clock
--      Bus2IP_Reset      -- Bus to IP reset
--      IP2Bus_IntrEvent  -- IP to Bus interrupt event
--      Bus2IP_Data       -- Bus to IP data bus for user logic
--      Bus2IP_BE         -- Bus to IP byte enables for user logic
--      Bus2IP_RdCE       -- Bus to IP read chip enable for user logic
--      Bus2IP_WrCE       -- Bus to IP write chip enable for user logic
--      IP2Bus_Data       -- IP to Bus data bus for user logic
--      IP2Bus_Ack        -- IP to Bus acknowledgement
--      IP2Bus_Retry      -- IP to Bus retry response
--      IP2Bus_Error      -- IP to Bus error response
--      IP2Bus_ToutSup    -- IP to Bus timeout suppress
-----

entity user_logic is
  generic
  (
    -- ADD USER GENERICS BELOW THIS LINE -----
    --USER generics added here
    -- ADD USER GENERICS ABOVE THIS LINE -----

    -- DO NOT EDIT BELOW THIS LINE -----
    -- Bus protocol parameters, do not add to or delete
    C_DWIDTH          : integer          := 32;
    C_NUM_CE          : integer          := 16;
    C_IP_INTR_NUM     : integer          := 8
    -- DO NOT EDIT ABOVE THIS LINE -----
  );

```



```

port
(
    -- ADD USER PORTS BELOW THIS LINE -----
    --USER ports added here
    ADC : in  STD_LOGIC_VECTOR (63 downto 0);
    Video : in  STD_LOGIC_VECTOR (7 downto 0);
    DEES_Start : in  STD_LOGIC;
    GPIO_OE : out  STD_LOGIC_VECTOR (29 downto 0);
    global_TOA_0 : out  STD_LOGIC_VECTOR (37 downto 0);
    user_sma_clk : in  STD_LOGIC;
    -- ADD USER PORTS ABOVE THIS LINE -----

    -- DO NOT EDIT BELOW THIS LINE -----
    -- Bus protocol ports, do not add to or delete
    Bus2IP_Clk : in  std_logic;
    Bus2IP_Reset : in  std_logic;
    IP2Bus_IntrEvent : out  std_logic_vector(0 to C_IP_INTR_NUM-1);
    Bus2IP_Data : in  std_logic_vector(0 to C_DWIDTH-1);
    Bus2IP_BE : in  std_logic_vector(0 to C_DWIDTH/8-1);
    Bus2IP_RdCE : in  std_logic_vector(0 to C_NUM_CE-1);
    Bus2IP_WrCE : in  std_logic_vector(0 to C_NUM_CE-1);
    IP2Bus_Data : out  std_logic_vector(0 to C_DWIDTH-1);
    IP2Bus_Ack : out  std_logic;
    IP2Bus_Retry : out  std_logic;
    IP2Bus_Error : out  std_logic;
    IP2Bus_ToutSup : out  std_logic;
    -- DO NOT EDIT ABOVE THIS LINE -----
);
end entity user_logic;

-----
-- Architecture section
-----

architecture IMP of user_logic is

    --USER signal declarations added here, as needed for user logic
    component TOA_generator
    Port ( clk_50 : in  STD_LOGIC;
          DEES_Start : in  STD_LOGIC;
          global_TOA : out  STD_LOGIC_VECTOR (37 downto 0)
        );
    end component;

    component PRC_v_0_0_1
    Port (
        clk_50 : in std_logic;
        clk_100 : in std_logic;
        rst : in std_logic;
        DEES_Start : in std_logic;
        video : in std_logic;

```

```

ADC_in : in std_logic_vector(7 downto 0);
global_TOA : in std_logic_vector(37 downto 0);
amplitude_out : out std_logic_vector(7 downto 0);
pulse_width : out std_logic_vector(15 downto 0);
TOA : out std_logic_vector(37 downto 0);
data_valid : out std_logic;
data_read : in std_logic;
overflow : out std_logic
);
end component;

```

```

signal channel_out : STD_LOGIC_VECTOR (511 downto 0);
signal global_TOA : STD_LOGIC_VECTOR (37 downto 0);
--signal data_valid : STD_LOGIC_VECTOR (7 downto 0);
signal data_read : STD_LOGIC_VECTOR (7 downto 0);
signal clk_50 : STD_LOGIC;
signal clk_50_dcm : STD_LOGIC;
signal clk_10_dcm : STD_LOGIC;
signal Start_to_OE : STD_LOGIC;

```

```

-----
-- Signals for user logic slave model s/w accessible register example
-----

```

```

-- signal slv_reg0           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg1           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg2           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg3           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg4           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg5           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg6           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg7           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg8           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg9           : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg10          : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg11          : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg12          : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg13          : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg14          : std_logic_vector(0 to C_DWIDTH-1);
-- signal slv_reg15          : std_logic_vector(0 to C_DWIDTH-1);
signal slv_reg_write_select  : std_logic_vector(0 to 15);
signal slv_reg_read_select   : std_logic_vector(0 to 15);
signal slv_ip2bus_data       : std_logic_vector(0 to C_DWIDTH-1);
signal slv_read_ack          : std_logic;
signal slv_write_ack         : std_logic;

```

```

-----
-- Signals for user logic interrupt example
-----

```

```

signal interrupt             : std_logic_vector(0 to C_IP_INTR_NUM-1);

```

```

begin
IBUFG_inst : IBUFG
generic map (
IOSTANDARD => "DEFAULT")
port map (
O => clk_10_dcm, -- Clock buffer output
I => user_sma_clk-- Clock buffer input (connect directly to top-level port)
);

--Generate the 50 MHz prc clock
DCM_inst : DCM
generic map (
CLKDV_DIVIDE => 2.0, -- Divide by: 1.5,2.0,2.5,3.0,3.5,4.0,4.5,5.0,5.5,6.0,6.5
-- 7.0,7.5,8.0,9.0,10.0,11.0,12.0,13.0,14.0,15.0 or 16.0
CLKFX_DIVIDE => 10, -- Can be any interger from 1 to 32
CLKFX_MULTIPLY => 5, -- Can be any integer from 1 to 32
CLKIN_DIVIDE_BY_2 => FALSE, -- TRUE/FALSE to enable CLKIN divide by two feature
CLKIN_PERIOD => 100.0, -- Specify period of input clock
CLKOUT_PHASE_SHIFT => "NONE", -- Specify phase shift of NONE, FIXED or VARIABLE
CLK_FEEDBACK => "NONE", -- Specify clock feedback of NONE, 1X or 2X
DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS", -- SOURCE_SYNCHRONOUS, SYSTEM_SYNCHRONOUS or
-- an integer from 0 to 15
DFS_FREQUENCY_MODE => "LOW", -- HIGH or LOW frequency mode for frequency synthesis
DLL_FREQUENCY_MODE => "LOW", -- HIGH or LOW frequency mode for DLL
DUTY_CYCLE_CORRECTION => TRUE,-- Duty cycle correction, TRUE or FALSE
FACTORY_JF => X"C080", -- FACTORY JF Values
PHASE_SHIFT => 0, -- Amount of fixed phase shift from -255 to 255
STARTUP_WAIT => FALSE) -- Delay configuration DONE until DCM LOCK, TRUE/FALSE
port map (
CLK0 => OPEN, -- 0 degree DCM CLK ouptput
CLK180 => OPEN, -- 180 degree DCM CLK output
CLK270 => OPEN, -- 270 degree DCM CLK output
CLK2X => OPEN, -- 2X DCM CLK output
CLK2X180 => OPEN, -- 2X, 180 degree DCM CLK out
CLK90 => OPEN, -- 90 degree DCM CLK output
CLKDV => OPEN, -- Divided DCM CLK out (CLKDV_DIVIDE)
CLKFX => clk_50_dcm,-- DCM CLK synthesis out (M/D)
CLKFX180 => OPEN, -- 180 degree CLK synthesis out
LOCKED => OPEN, -- DCM LOCK status output
PSDONE => OPEN, -- Dynamic phase adjust done output
STATUS => OPEN, -- 8-bit DCM status bits output
CLKFB => OPEN, -- DCM clock feedback
CLKIN => Bus2IP_Clk,-- Clock input (from IBUFG, BUFG or DCM)
PSCLK => '0', -- Dynamic phase adjust clock input
PSEN => '0', -- Dynamic phase adjust enable input
PSINCDEC => '0', -- Dynamic phase adjust increment/decrement
RST => Bus2IP_Reset -- DCM asynchronous reset input
);

CLK_BUFF : BUFG

```

```

port map (
  I => clk_50_dcm,
  O => clk_50
);

--Instantiate the Time of Arrival Generator
TOA_Inst : TOA_generator
port map (
  clk_50 => clk_50,
  DEES_Start => DEES_Start,
  global_TOA => global_TOA
);

--Instantiates the 8 PRC channels
PRC_Parts:
for N in 0 to 7 generate
  PRC_Channel : PRC_v_0_0_1

  Port map(
    clk_50 => clk_50,
    clk_100 => Bus2IP_Clk,
    rst => Bus2IP_Reset,
    DEES_Start => DEES_Start,
    video => Video(N),
    ADC_in => ADC(N*8 + 7 downto N*8),
    global_TOA => global_TOA,
    amplitude_out => channel_out(N*64 + 7 downto N*64),
    pulse_width => channel_out(N*64 + 63 downto N*64 + 48),
    TOA => channel_out(N*64 + 45 downto N*64 + 8),
    data_valid => channel_out(N*64 + 46),
    --data_valid => data_valid(N),
    data_read => data_read(N),
    overflow => channel_out(N*64 + 47)
  );
end generate;

--Send the proper control signals to the GPIO buffers on the DREAM board
Start_to_OE <= not DEES_Start;

GPIO_OE(0) <= Start_to_OE;
GPIO_OE(1) <= Start_to_OE;
GPIO_OE(2) <= Start_to_OE;
GPIO_OE(3) <= Start_to_OE;
GPIO_OE(4) <= Start_to_OE;
GPIO_OE(5) <= Start_to_OE;
GPIO_OE(6) <= Start_to_OE;
GPIO_OE(7) <= Start_to_OE;
GPIO_OE(8) <= Start_to_OE;
GPIO_OE(9) <= Start_to_OE;
GPIO_OE(10) <= Start_to_OE;
GPIO_OE(11) <= Start_to_OE;

```

```

GPIO_OE(12) <= Start_to_OE;
GPIO_OE(13) <= Start_to_OE;
GPIO_OE(14) <= Start_to_OE;
GPIO_OE(15) <= Start_to_OE;
GPIO_OE(16) <= Start_to_OE;
GPIO_OE(17) <= Start_to_OE;
GPIO_OE(18) <= Start_to_OE;
GPIO_OE(19) <= Start_to_OE;

```

```

GPIO_OE(20) <= DEES_Start;
GPIO_OE(21) <= Start_to_OE;
GPIO_OE(22) <= DEES_Start;
GPIO_OE(23) <= Start_to_OE;
GPIO_OE(24) <= DEES_Start;
GPIO_OE(25) <= Start_to_OE;
GPIO_OE(26) <= DEES_Start;
GPIO_OE(27) <= Start_to_OE;
GPIO_OE(28) <= DEES_Start;
GPIO_OE(29) <= Start_to_OE;

```

```

--temp signal to monitor TOA on GPIO port
global_TOA_0 <= global_TOA;

```

```

-- code to read/write user logic slave model s/w accessible registers

```

```

slv_reg_write_select <= Bus2IP_WrCE(0 to 15);

```

```

slv_reg_read_select <= Bus2IP_RdCE(0 to 15);-- and (Data_Valid(0) & Data_Valid(0) & Data_Va

```

```

slv_write_ack <= Bus2IP_WrCE(0) or Bus2IP_WrCE(1) or Bus2IP_WrCE(2) or Bus2IP_WrCE(3)

```

```

slv_read_ack <= Bus2IP_RdCE(0) or Bus2IP_RdCE(1) or Bus2IP_RdCE(2) or Bus2IP_RdCE(3)

```

```

SLAVE_REG_READ_PROC : process( slv_reg_read_select) is

```

```

begin

```

```

    case slv_reg_read_select is

```

```

        when "1000000000000000" => slv_ip2bus_data <= channel_out(31 downto 0);

```

```

        when "0100000000000000" => slv_ip2bus_data <= channel_out(63 downto 32);

```

```

        when "0010000000000000" => slv_ip2bus_data <= channel_out(95 downto 64);

```

```

    when "0001000000000000" => slv_ip2bus_data <= channel_out(127 downto 96);

```

```

        when "0000100000000000" => slv_ip2bus_data <= channel_out(159 downto 128);

```

```

        when "0000010000000000" => slv_ip2bus_data <= channel_out(191 downto 160);

```

```

    when "0000001000000000" => slv_ip2bus_data <= channel_out(223 downto 192);

```

```

        when "0000000100000000" => slv_ip2bus_data <= channel_out(255 downto 224);

```

```

        when "0000000010000000" => slv_ip2bus_data <= channel_out(287 downto 256);

```

```

    when "0000000001000000" => slv_ip2bus_data <= channel_out(319 downto 288);

```

```

        when "0000000000100000" => slv_ip2bus_data <= channel_out(351 downto 320);

```

```

        when "0000000000010000" => slv_ip2bus_data <= channel_out(383 downto 352);

```

```

    when "0000000000001000" => slv_ip2bus_data <= channel_out(415 downto 384);

```

```

        when "0000000000000100" => slv_ip2bus_data <= channel_out(447 downto 416);

```

```

        when "0000000000000010" => slv_ip2bus_data <= channel_out(479 downto 448);

```

```

    when "0000000000000001" => slv_ip2bus_data <= channel_out(511 downto 480);

```

```

    when others => slv_ip2bus_data <= (others => '0');

```

```

    end case;

```

```

end process SLAVE_REG_READ_PROC;

```

```

-- implement slave model register(s)
SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is
begin
    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
        if Bus2IP_Reset = '1' then
data_read <= "00000000";
        else
            case slv_reg_write_select is
                when "1000000000000000" =>data_read <= "00000001";
                when "0100000000000000" =>data_read <= "00000001";
                when "0010000000000000" =>data_read <= "00000010";
                when "0001000000000000" =>data_read <= "00000010";
                when "0000100000000000" =>data_read <= "00000100";
                when "0000010000000000" =>data_read <= "00000100";
                when "0000001000000000" =>data_read <= "00001000";
                when "0000000100000000" =>data_read <= "00001000";
                when "0000000010000000" =>data_read <= "00010000";
                when "0000000001000000" =>data_read <= "00010000";
                when "0000000000100000" =>data_read <= "00100000";
                when "0000000000010000" =>data_read <= "00100000";
                when "0000000000001000" =>data_read <= "01000000";
                when "0000000000000100" =>data_read <= "01000000";
                when "0000000000000010" =>data_read <= "10000000";
                when "0000000000000001" =>data_read <= "10000000";
                when others => null;
            end case;
        end if;
    end if;
end process SLAVE_REG_WRITE_PROC;

interrupt <= Video;
IP2Bus_IntrEvent <= interrupt;
IP2Bus_Data <= slv_ip2bus_data;
IP2Bus_Ack <= slv_write_ack or slv_read_ack;
IP2Bus_Error <= '0';
IP2Bus_Retry <= '0';
IP2Bus_ToutSup <= '0';

end IMP;

```

## PRC v 0 0 1.vhd

```
-----
-- Engineer: Frank Fradette
-- Create Date:      10:15:47 08/23/05
-- Design Name:      DREAM
-- Module Name:      PRC_v_0_0_1 - Behavioral
-- Project Name:      TYIGR_PRC_v0_0_1
-- Target Device:     Xilinx V2P 30
-- Tool versions:     ISE 8.2.01i
-- Description: The Pulse Receptor Core is used to find the amplitude,
--pulse width, and time of arrival values of the input
--signals.
--
-- Dependencies:
--
-- Revision:
-- Revision 0.01 - File Created
-- Revision 0.11 - State Transitions and State Actions seperated
-- - Variable names changed to be more descriptive
-- Additional Comments:
```

```
-----
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;
Library XilinxCoreLib;
```

```
entity PRC_v_0_0_1 is
port( clk_50 : in std_logic;
      clk_100 : in std_logic;
      rst : in std_logic;
      DEES_Start : in std_logic;
      video : in std_logic;
      ADC_in : in std_logic_vector(7 downto 0);
      global_TOA : in std_logic_vector(37 downto 0);
      amplitude_out : out std_logic_vector(7 downto 0) := x"00";
      pulse_width : out std_logic_vector(15 downto 0) := x"0000";
      TOA : out std_logic_vector(37 downto 0) := "00"&x"0000000000";
      data_valid : out std_logic:= '0';
      data_read : in std_logic;
      overflow : out std_logic:= '0'
--;
----state signaling test signals
--state_reset : out std_logic;
--state_wait : out std_logic;
--state_pre_thresh : out std_logic;
```

```

--state_post_thresh : out std_logic;
--state_valid : out std_logic;
--state_overflow : out std_logic
);
end PRC_v_0_0_1;

architecture Behavioral of PRC_v_0_0_1 is

constant ADC_pipeline_delay: std_logic_vector(7 downto 0) := x"05";
constant pulse_width_discriminator: std_logic_vector(15 downto 0) := x"0005";

type state_type is (S_Reset, S_Wait, S_Pre_Threshold, S_Post_Threshold, S_Valid, S_Pulse_Width);
signal state : state_type;
signal n_state : state_type;

--Pulse Counter Signals
signal pulse_width_counter : std_logic_vector(15 downto 0):= x"0000";

--Signal Registers
signal amplitude_register : std_logic_vector(7 downto 0):= x"00";
signal pulse_width_register: std_logic_vector(15 downto 0):= x"0000";
signal toa_register : std_logic_vector(37 downto 0):= "00"&x"0000000000";
signal overflow_register : std_logic:= '0';

signal valid : std_logic;
signal data_q : std_logic:= '0';
--signal counter_error : std_logic;

begin

-----
--Runs the Pulse Width Counter if DEES Start is Active and Video is present
-----

process (clk_50)
begin
if clk_50='1' and clk_50'event then
if video='0' then
pulse_width_counter <= x"0000";
elsif DEES_Start='1' then
pulse_width_counter <= pulse_width_counter + 1;
end if;
end if;
end process;

-----
--State Switching Engine
-----

SETUP: process(clk_50, rst, DEES_Start)
begin
if(rst = '1' or DEES_Start = '0') then
state <= S_Reset;

```



```

elsif( clk_50 = '1' and clk_50'event) then
state <= n_state;
end if;
end process;

-----
--State Steering
-----

State_Transitions: process(state,pulse_width_counter,video)
begin
case state is
when S_Reset =>
n_state <= S_Wait;

when S_Wait =>
if (video = '1') then
n_state <= S_Pre_Threshold;
else
n_state <= S_Wait;
end if;

when S_Pre_Threshold =>
if((video = '1') and (pulse_width_counter = pulse_width_discriminator-1)) then
n_state <= S_Post_Threshold;
elsif (video = '0') then
n_state <= S_Wait;
else
n_state <= S_Pre_Threshold;
end if;

when S_Post_Threshold =>
if((video = '1') and (pulse_width_counter = x"FFFF")) then
n_state <= S_Pulse_Width_Overflow;
elsif (video = '0') then
n_state <= S_Valid;
else
n_state <= S_Post_Threshold;
end if;

when S_Valid =>
n_state <= S_Wait;

when S_Pulse_Width_Overflow =>
n_state <= S_Reset;

when others =>
n_state <= S_Reset;
end case;
end process;

```

-----  
--TOA Register  
-----

```
process (clk_50)
begin
    if clk_50'event and clk_50='1' and n_state=S_Pre_Threshold and state=S_Wait then
        toa_register <= global_TOA;
    end if;
end process;
```

-----  
--Pulse Width Register  
-----

```
process (clk_50)
begin
    if clk_50'event and clk_50='1' and state=S_Post_Threshold and video='0' then
        pulse_width_register <= pulse_width_counter;
    end if;
end process;
```

-----  
--Amplitude Register  
-----

```
process (clk_50)
begin
    if clk_50'event and clk_50='1' and pulse_width_counter+1= ADC_pipeline_delay then
        amplitude_register <= ADC_in;
    end if;
end process;
```

-----  
--Pulse Width Overflow Register  
-----

```
process (clk_50)
begin
    if clk_50'event and clk_50='1' then
    if state=S_Pulse_Width_Overflow then
        overflow_register <= '1';
    else
        overflow_register <= '0';
    end if;
    end if;
end process;
```

-----  
--Data Valid Register  
-----

```
process (clk_100)
begin
    if clk_100'event and clk_100='1' then
        data_q <= valid or (data_q and (not data_read));
    end if;
end process;
```

```

        end if;
    end process;

```

```

process(state)
begin
    if state = S_Valid then
        valid <= '1';
    else
        valid <= '0';
    end if;
end process;

```

```

-----
--Tying signals to Output Ports
-----

```

```

amplitude_out <= amplitude_register;
pulse_width <= pulse_width_register;
TOA <= toa_register;
overflow <= overflow_register;
data_valid <= data_q;

```

```

-----
---- This process is for troubleshooting state transitions
-----

```

```

--STATE_test: process(clk_50,state)
--begin
--case state is
--when S_Reset =>
--state_reset <='1';
--state_wait <='0';
--state_pre_thresh <='0';
--state_post_thresh <='0';
--state_valid <='0';
--state_overflow <='0';
--
--when S_Wait =>
--state_reset <='0';
--state_wait <='1';
--state_pre_thresh <='0';
--state_post_thresh <='0';
--state_valid <='0';
--state_overflow <='0';
--
--when S_Pre_Threshold =>
--state_reset <='0';
--state_wait <='0';
--state_pre_thresh <='1';
--state_post_thresh <='0';
--state_valid <='0';

```

```

--state_overflow <='0';
--
--when S_Post_Threshold =>
--state_reset <='0';
--state_wait <='0';
--state_pre_thresh <='0';
--state_post_thresh <='1';
--state_valid <='0';
--state_overflow <='0';
--
--when S_Valid =>
--state_reset <='0';
--state_wait <='0';
--state_pre_thresh <='0';
--state_post_thresh <='0';
--state_valid <='1';
--state_overflow <='0';
--
--when S_Pulse_Width_Overflow =>
--state_reset <='0';
--state_wait <='0';
--state_pre_thresh <='0';
--state_post_thresh <='0';
--state_valid <='0';
--state_overflow <='1';
--
--end case;
--end process;
end Behavioral;

```

## TOA\_generator.vhd

```
-----
-- Engineer: Frank Fradette
-- Create Date: 21:10:36 01/09/2008
-- Design Name: DREAM
-- Module Name: TOA_generator - Behavioral
-- Project Name: Time of Arrival Signal Generator
-- Target Devices: VirtexII
-- Tool versions: ISE 8.1
-- Description: The Time of arrival signal generator increments and outputs a
--38 bit signal which is a count of how many clock cycles have
--occured since the beginning of a simulation
-- Dependencies:
--
-- Revision 0.01 - File Created
-- Additional Comments:
-----
```

```
library IEEE;
Library UNISIM;
use UNISIM.vcomponents.all;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity TOA_generator is
    Port ( clk_50 : in STD_LOGIC;
          DEES_Start : in STD_LOGIC;
          global_TOA : out STD_LOGIC_VECTOR (37 downto 0));
end TOA_generator;

architecture Behavioral of TOA_generator is
    signal toa_temp: std_logic_vector(37 downto 0);

begin
    process (clk_50)
    begin
        if (clk_50'event and clk_50 = '1') then
            if DEES_Start = '1' then
                toa_temp <= toa_temp + 1;
            else
                toa_temp <= "00"&x"0000000000"; --Initial and Reset Value
            end if;
        end if;
    end process;
    global_TOA <= toa_temp;
end Behavioral;
```

## PRC control user logic.vhd

```
-----
-- user_logic.vhd - entity/architecture pair
-----
-- *****
-- ** Copyright (c) 1995-2006 Xilinx, Inc. All rights reserved. **
-- **                                                                 **
-- ** Xilinx, Inc.                                                                 **
-- ** XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" **
-- ** AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND **
-- ** SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE, **
-- ** OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, **
-- ** APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION **
-- ** THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT, **
-- ** AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE **
-- ** FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY **
-- ** WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE **
-- ** IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR **
-- ** REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF **
-- ** INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS **
-- ** FOR A PARTICULAR PURPOSE. **
-- **                                                                 **
-- *****
-----
-- Filename:          user_logic.vhd
-- Version:           1.00.a
-- Description:       User logic.
-- Date:             Thu Jun 12 13:42:12 2008 (by Create and Import Peripheral Wizard)
-- VHDL Standard:    VHDL'93
-----

-- DO NOT EDIT BELOW THIS LINE -----
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

library proc_common_v2_00_a;
use proc_common_v2_00_a.proc_common_pkg.all;
-- DO NOT EDIT ABOVE THIS LINE -----

-----
-- Entity section
-----
-- Definition of Generics:
--   C_DWIDTH          -- User logic data bus width
--   C_NUM_CE          -- User logic chip enable bus width
--   C_RDFIFO_DWIDTH   -- Data width of Read FIFO
--
-- Definition of Ports:
--   Bus2IP_Clk        -- Bus to IP clock
```

```

-- Bus2IP_Reset          -- Bus to IP reset
-- Bus2IP_Data           -- Bus to IP data bus for user logic
-- Bus2IP_BE             -- Bus to IP byte enables for user logic
-- Bus2IP_RdCE           -- Bus to IP read chip enable for user logic
-- Bus2IP_WrCE           -- Bus to IP write chip enable for user logic
-- IP2Bus_Data           -- IP to Bus data bus for user logic
-- IP2Bus_Ack            -- IP to Bus acknowledgement
-- IP2Bus_Retry          -- IP to Bus retry response
-- IP2Bus_Error          -- IP to Bus error response
-- IP2Bus_ToutSup        -- IP to Bus timeout suppress
-- IP2RFIFO_WrReq        -- IP to RFIFO : IP write request
-- IP2RFIFO_Data         -- IP to RFIFO : IP write data
-- RFIFO2IP_WrAck        -- RFIFO to IP : RFIFO write acknowledge
-- RFIFO2IP_AlmostFull   -- RFIFO to IP : RFIFO almost full
-- RFIFO2IP_Full         -- RFIFO to IP : RFIFO full
-----

```

entity user\_logic is

generic

(

-- DO NOT EDIT BELOW THIS LINE -----

-- Bus protocol parameters, do not add to or delete

C\_DWIDTH : integer := 32;

C\_NUM\_CE : integer := 1;

C\_RDFIFO\_DWIDTH : integer := 32

-- DO NOT EDIT ABOVE THIS LINE -----

);

port

(

-- ADD USER PORTS BELOW THIS LINE -----

--USER ports added here

GPIO\_Out : out STD\_LOGIC\_VECTOR (31 downto 0);

DAV : out STD\_LOGIC;

-- ADD USER PORTS ABOVE THIS LINE -----

-- DO NOT EDIT BELOW THIS LINE -----

-- Bus protocol ports, do not add to or delete

Bus2IP\_Clk : in std\_logic;

Bus2IP\_Reset : in std\_logic;

Bus2IP\_Data : in std\_logic\_vector(0 to C\_DWIDTH-1);

Bus2IP\_BE : in std\_logic\_vector(0 to C\_DWIDTH/8-1);

Bus2IP\_RdCE : in std\_logic\_vector(0 to C\_NUM\_CE-1);

Bus2IP\_WrCE : in std\_logic\_vector(0 to C\_NUM\_CE-1);

IP2Bus\_Data : out std\_logic\_vector(0 to C\_DWIDTH-1);

IP2Bus\_Ack : out std\_logic;

IP2Bus\_Retry : out std\_logic;

IP2Bus\_Error : out std\_logic;

IP2Bus\_ToutSup : out std\_logic;

IP2RFIFO\_WrReq : out std\_logic;

IP2RFIFO\_Data : out std\_logic\_vector(0 to C\_RDFIFO\_DWIDTH-1);

RFIFO2IP\_WrAck : in std\_logic;

```

        RFIFO2IP_AlmostFull      : in  std_logic;
        RFIFO2IP_Full            : in  std_logic
    -- DO NOT EDIT ABOVE THIS LINE -----
    );
end entity user_logic;

-----

-- Architecture section
-----

architecture IMP of user_logic is

    component prc_control_top is
        Port(
            clk : in  STD_LOGIC;
            rst : in  STD_LOGIC;
            A_register_changed : in STD_LOGIC;
            Command_in : in  STD_LOGIC_VECTOR (31 downto 0);
            GPIO_Out : out  STD_LOGIC_VECTOR (31 downto 0);
            DAV : out  STD_LOGIC
        );
    end component;

    -----
    -- Signals for user logic slave model s/w accessible register
    -----

    signal slv_reg0                : std_logic_vector(0 to C_DWIDTH-1);
    signal slv_reg_write_select    : std_logic_vector(0 to 0);
    signal slv_reg_read_select     : std_logic_vector(0 to 0);
    signal slv_ip2bus_data         : std_logic_vector(0 to C_DWIDTH-1);
    signal slv_read_ack            : std_logic;
    signal slv_write_ack           : std_logic;

begin
    controller: prc_control_top
        PORT MAP(
            clk => Bus2IP_Clk,
            rst => Bus2IP_Reset,
            A_register_changed => slv_reg_write_select(0),
            Command_in => slv_reg0,
            GPIO_Out => GPIO_Out,
            DAV => DAV
        );

    slv_reg_write_select <= Bus2IP_WrCE(0 to 0);
    slv_reg_read_select  <= Bus2IP_RdCE(0 to 0);
    slv_write_ack        <= Bus2IP_WrCE(0);
    slv_read_ack         <= Bus2IP_RdCE(0);

    -- implement slave model register(s)
    SLAVE_REG_WRITE_PROC : process( Bus2IP_Clk ) is

```



```

begin

    if Bus2IP_Clk'event and Bus2IP_Clk = '1' then
        if Bus2IP_Reset = '1' then
            slv_reg0 <= (others => '0');
        else
            case slv_reg_write_select is
                when "1" =>
                    for byte_index in 0 to (C_DWIDTH/8)-1 loop
                        if ( Bus2IP_BE(byte_index) = '1' ) then
                            slv_reg0(byte_index*8 to byte_index*8+7) <= Bus2IP_Data(byte_index*8 to byte_index*8+7);
                        end if;
                    end loop;
                when others => null;
            end case;
        end if;
    end if;

end process SLAVE_REG_WRITE_PROC;

-- implement slave model register read mux
SLAVE_REG_READ_PROC : process( slv_reg_read_select, slv_reg0 ) is
begin

    case slv_reg_read_select is
        when "1" => slv_ip2bus_data <= slv_reg0;
        when others => slv_ip2bus_data <= (others => '0');
    end case;

end process SLAVE_REG_READ_PROC;

-----
-- Code to drive IP to Bus signals
-----

IP2Bus_Data          <= slv_ip2bus_data;

IP2Bus_Ack            <= slv_write_ack or slv_read_ack;
IP2Bus_Error          <= '0';
IP2Bus_Retry          <= '0';
IP2Bus_ToutSup        <= '0';

end IMP;

```

## clock\_counter.vhd

```
-----
-- Engineer:      Frank Fradette
--
-- Create Date:    13:47:04 04/04/2007
-- Design Name:    Clock Counter for PRC Control
-- Module Name:    clock_counter - Behavioral
-- Project Name:   DREAM board
-- Target Devices: VirtexII Pro
-- Tool versions:  ISE 8.2
-- Description:    Counts a 50 MHz clock and generates signals denoting 1ms and
--150ns have passed
-- Dependencies:
--
-- Revision 0.01 - File Created
-- Additional Comments:
--
-----

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity clock_counter is
    Port ( clk : in  STD_LOGIC;
          clk_tick : in  STD_LOGIC;
          one_us : out STD_LOGIC;
          one_fifty_ns: out STD_LOGIC);
end clock_counter;

architecture Behavioral of clock_counter is

    -- Signals -----
    signal tick_count :std_logic_vector (6 downto 0);

begin

    --process to increment or reset counter
    Count: process(clk,clk_tick)
    begin
        if(clk_tick = '0') then
            tick_count <= "0000000"; --reset
        elsif(clk'event and clk='1') then
            tick_count <= tick_count + 1; --increment
        end if;
    end process;

end process;
```

```

--process to assert output signals when the specified value is reached
tock: process(tick_count)
begin
if (tick_count = "1100011") then -- 99 counts
one_us <= '1';
one_fifty_ns <= '0';
elsif (tick_count = "1110010") then -- 14 more counts
one_us <= '0';
one_fifty_ns <= '1';
else
one_us <= '0';
one_fifty_ns <= '0';
end if;
end process;

end Behavioral;

```

## APPENDIX C

### C Code for Embedded PowerPC

## TestApp Memory.c

```
//TestApp_Memory.c
//Main application for testing gigabit ethernet connectivity and functionality.
//Provides all initialization for DREAM board gigabit ethernet function
//Initializes Gigabit Ethernet MAC on FPGA
//Initialized SGMII to GMII bridge
//Initializes Vitesse gigabit PHY
//Provides functions to set MDIO registers
//Provides functions to set GEMAC registers

// Located in: ppc405_0/include/xparameters.h
#include "xparameters.h"
#include "stdio.h"
#include "xutil.h"
#include "xgemac.c"

//=====
void printBits(Xuint32 input);
void printBits_Sp(Xuint32 input, int spaces);
void print16Bits(Xuint32 input, int shift, int spaces);
void printBottomBits(Xuint32 input);
void printTopBits(Xuint32 input);
void wait(Xuint32 seconds);
void setPHYRegister(Xuint32 regNum, Xuint32 regValue, Xuint32 PHYChannel);
void readPHYRegister(Xuint32 regNum, Xuint32 PHYChannel);
void printGEMACRegister(Xuint32 regNum);
void PrintGEMACRegisters(void);
void PrintGEMACExtendedRegisters(void);
void ReceiveInterrupt(void);
void PrintBool(Xboolean stat);
void PrintOtoF(void);
void PrintSGMIIRegisters(void);
void PrintPHYRegisters(void);
void PrintGEMACRegisters(void);
void InitializeGEMAC(Xuint32 GemacAddress);
void InitializeSGMII(Xuint8 SgmiiAddress);
void InitializePHY(Xuint32 PHYChannel);
void EPG_on(void);
void EPG_off(void);
void EPG_on_bad(void);

int main (void) {
    print("-- Entering main() --\r\n");

    /* Testing BRAM Memory (plb_bram_if_cntlr_1)*/
    {
        XStatus status;

        print("Starting MemoryTest for plb_bram_if_cntlr_1:\r\n");
        print("  Running 32-bit test...");
        status = XUtil_MemoryTest32((Xuint32*)XPAR_PLB_BRAM_IF_CNTLR_1_BASEADDR, 512, 0xAAAA5555
```

```

        if (status == XST_SUCCESS) {
            print("PASSED!\r\n");
        }
        else {
            print("FAILED!\r\n");
        }
        print(" Running 16-bit test...");
        status = XUtil_MemoryTest16((Xuint16*)XPAR_PLB_BRAM_IF_CNTLR_1_BASEADDR, 1024, 0xAA55, X
        if (status == XST_SUCCESS) {
            print("PASSED!\r\n");
        }
        else {
            print("FAILED!\r\n");
        }
        print(" Running 8-bit test...");
        status = XUtil_MemoryTest8((Xuint8*)XPAR_PLB_BRAM_IF_CNTLR_1_BASEADDR, 2048, 0xA5, XUT_A
        if (status == XST_SUCCESS) {
            print("PASSED!\r\n");
        }
        else {
            print("FAILED!\r\n");
        }
    }
    int i;

InitializePHY(0x0);

//enable far end loop back modefor PHY port B
setPHYRegister(0x17, 0xA00A, 0x1);
//PHY must be reset to enact change
setPHYRegister(0x00, 0x9040, 0x1);
//change one of the status leds for PHY port B
setPHYRegister(0x1B, 0x0300, 0x1);

InitializeSGMII(0x6);

//Read and Print PHY's registers
PrintPHYRegisters();

//Print sgmmii bridge MDIO registers
PrintSGMIIRegisters();

//Print sgmmii bridge MDIO registers again
PrintSGMIIRegisters();
PrintGEMACExtendedRegisters();

////PHY Loopback mode
// print("Place PHY into Near End Loopback mode\r\n");
// setPHYRegister(0x00, 0x5040, 0x0);
// wait(3);
//

```

```

// XGmacSetup();
//
// print("Take PHY out of Near End Loopback mode\r\n");
// setPHYRegister(0x00, 0x9040, 0x0);
// wait(3);

////Connector Loopback mode
// print("Place PHY into Connector Loopback mode\r\n");
// setPHYRegister(0x09, 0x1E00, 0x0);
// setPHYRegister(0x18, 0x0241, 0x0);
// setPHYRegister(0x12, 0x0029, 0x0);
// setPHYRegister(0x00, 0x0040, 0x0);
// wait(10);

// print("Take PHY out of Connector Loopback mode\r\n");
// setPHYRegister(0x00, 0x1040, 0x0);
// setPHYRegister(0x09, 0x0600, 0x0);
// setPHYRegister(0x12, 0x0009, 0x0);
// setPHYRegister(0x18, 0x0240, 0x0);
// wait(5);

// PHY Ethernet Packet Generator
// EPG_on();
// wait(1);
// EPG_off();
//
// EPG_on_bad();
// wait(1);
// EPG_off();

while(1)
{
//ParseEthernetPacket();
print("\t0 - Locked in Ping Response Mode\r\n");
print("\t1 - Print Next Received Packet\r\n");
print("\t2 - Parse Ethernet Packet\r\n");
print("\t3 - APR Request\r\n");
print("Command: ");
switch((int)XUartLite_RecvByte(STDIN_BASEADDRESS))
{
case 48: while(1){ParseEthernetPacket();}break;
case 49: PrintNextReceivedPacket(); break;
case 50: ParseEthernetPacket(); break;
case 51: ARPRequest(); break;
case 52: print("4"); break;
case 53: print("5"); break;
case 54: print("6"); break;
case 55: print("7"); break;
case 56: print("8"); break;
case 57: print("9"); break;
case 97: print("a"); break;//A

```

```

case 98: print("b"); break;//B
case 99: print("c"); break;//C
case 100: print("d"); break;//D
case 101: print("e"); break;//E
case 102: print("f"); break;//F
default: break;
}
print("\r\n");
}

print("-- Exiting main() --\r\n");
return 0;

}

//function to print the input variable in bits
void printBits(Xuint32 input)
{
printTopBits(input);
printBottomBits(input);
}

//function to print the input variable in bits with a number of spaces between each bit
void printBits_Sp(Xuint32 input, int spaces)
{
print16Bits(input, 0,spaces);
print16Bits(input, 16,spaces);
}

//function to print the upper part of the input variable in bits
void printTopBits(Xuint32 input)
{
print16Bits(input, 0,2);
}

//function to print the lower part of the input variable in bits
void printBottomBits(Xuint32 input)
{
print16Bits(input, 16,2);
}

//function to print the input variable in bits
void print16Bits(Xuint32 input, int shift, int spaces)
{
int i;
int k;
Xuint32 inputHolder = input << shift;
for(i=0; i<16; i++)
{
Xuint32 thisBit = 0x80000000 & inputHolder;
if (thisBit == 0)

```



```

{
print("0");
}
else
{
print("1");
}
inputHolder = inputHolder << 1;

for (k=spaces; k>0; k--)
{
print(" ");
}
}

//function to cause the PowerPC to wait to avoid overloading the serial connection
void wait(Xuint32 seconds)
{
print("Forcing Wait");
int i;
for(i=0; i<seconds; i++)
{
print("*");
int winks=0;
for (winks=0; winks<20000000; winks++)
{
winks++;
winks--;
}
}
print("\r\n");
}

//function to send a register settings to a PHY channel
void setPHYRegister(Xuint32 regNum, Xuint32 regValue, Xuint32 PHYChannel)
{
print("PHY Channel: 0x");
putnum(PHYChannel);
print(" PHY Register: 0x");
putnum(regNum);

PHYChannel = PHYChannel << 25;
regNum = ((regNum << 4) + 0x00008008) << 16;
regNum = regNum + PHYChannel;

// print("XGmac_mWriteReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1018, 0x");
// putnum(regValue);
// print(");\r\nXGmac_mWriteReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1014, 0x");
// putnum(regNum);
// print(");");

```

```

XGmac_mWriteReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1018, regValue); //whats getting stuffed in a re
XGmac_mWriteReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1014, regNum); //where its getting stuffed
print(" is now set to 0x");
putnum(regValue);
print("\r\n");
}

//function to read a register settings from a PHY channel
void readPHYRegister(Xuint32 regNum, Xuint32 PHYChannel)
{
//print("\r\nXGmac_mWriteReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1014, ");
//putnum(regNum);
//print("\t");
PHYChannel = PHYChannel << 25;
regNum = regNum << 20;
regNum = regNum + 0xC0080000;
regNum = regNum + PHYChannel;
XGmac_mWriteReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1014, regNum);

//putnum(regNum);
putnum(regNum - PHYChannel - 0xC0080000);
print(" ");
//printBottomBits(XGmac_mReadReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1018));
putnum(XGmac_mReadReg(XPAR_PLB_GEMAC_0_BASEADDR, 0x1018));
print("\r\n");
}

//function to read and print a GEMAC register setting to teminal
void printGEMACRegister(Xuint32 regNum)
{
//print("\r\nXGmac_mReadReg(XPAR_PLB_GEMAC_0_BASEADDR, ");
print("\r\n");
putnum(regNum);
print(" ");
Xuint32 temp = XGmac_mReadReg(XPAR_PLB_GEMAC_0_BASEADDR, regNum);
printTopBits(temp);
print("\r\n");
printBottomBits(temp);
}

//function to print out boolean values to terminal
void PrintBool(Xboolean stat)
{
if(stat==XTRUE)
print("True\r\n");
else
print("False\r\n");
wait(1);
}

//function to read and print all GEMAC register settings to teminal

```

```

void PrintGEMACRegisters(void)
{
    int i;
    print("Print All PLB GEMAC Registers\r\n");
    PrintOtoF();
    for (i = 0x1000; i < 0x1038; i = i + 0x0004)
    {
        printGEMACRegister(i);
    }
    PrintOtoF();
}

//function to read and print extended GEMAC register settings to teminal
void PrintGEMACExtendedRegisters(void)
{
    int i;
    print("Print All Extended PLB GEMAC Registers\r\n");
    print("\r\n \t\t0 1 2 3 4 5 6 7 8 9 A B C D E F\r\n");
    for (i = 0x2300; i < 0x2374; i = i + 0x0004)
    {
        printGEMACRegister(i);
    }
    print("\r\n \t\t0 1 2 3 4 5 6 7 8 9 A B C D E F\r\n");
    printGEMACRegister(0x2000);
    printGEMACRegister(0x2004);
    printGEMACRegister(0x2100);
    printGEMACRegister(0x2010);
    printGEMACRegister(0x2014);
    printGEMACRegister(0x2200);
    printGEMACRegister(0x0000);
    printGEMACRegister(0x0004);
    printGEMACRegister(0x0008);
    printGEMACRegister(0x0018);
    printGEMACRegister(0x001C);
    printGEMACRegister(0x0020);
    printGEMACRegister(0x0028);
    printGEMACRegister(0x0040);
    print("\r\n \t\t0 1 2 3 4 5 6 7 8 9 A B C D E F\r\n");
}

//function to read and print SGMII register settings to teminal
void PrintSGMIIRegisters(void)
{
    int i;
    print("Print All SGMII Bridge Registers\r\n");
    PrintOtoF();
    for (i = 0x00; i < 0x20; i = i + 0x01)
    {
        readPHYRegister(i, 0x6);
    }
    PrintOtoF();
}

```

```

}

//function to read and print PHY register settings to terminal
void PrintPHYRegisters(void)
{
    int i;
    print("Print All PHY Registers\r\n");
    Print0toF();
    for (i = 0x00; i < 0x20; i = i + 0x01)
    {
        readPHYRegister(i, 0x0);
    }
    Print0toF();
}

//function to print a header to help identify the position of bits
void Print0toF(void)
{
    print("\r\n \t F E D C B A 9 8 7 6 5 4 3 2 1 0 \r\n");
}

//function to set SGMII register settings for initialization
void InitializeSGMII(Xuint8 SgmiiAddress)
{
    //setPHYRegister(0x10, 0x0000, SgmiiAddress);
    setPHYRegister(0x00, 0x1140, SgmiiAddress);
    //wait(5);
}

//function to set PHY register settings for initialization
void InitializePHY(Xuint32 PHYChannel)
{
    //SGMII 4 pin interface with autonegotiation enabled
    print("SGMII 4 pin interface with autonegotiation enabled \r\n");
    setPHYRegister(0x17, 0xA002, PHYChannel);
    //setPHYRegister(0x00, 0x9040, PHYChannel); //PHY must be reset to enact change
    setPHYRegister(0x00, 0x9000, PHYChannel); //PHY must be reset to enact change
    //wait(2);

    //change one of the status leds
    setPHYRegister(0x1B, 0x0300, PHYChannel);

    //Switch to PHY's extended register set
    setPHYRegister(0x1F, 0x0001, PHYChannel);

    //Switch PHY's 125MHz clk to 125MHz
    // setPHYRegister(0x14, 0x0145, PHYChannel);

    //Switch SIGDET pin direction to output, active high
    setPHYRegister(0x13, 0x0002, PHYChannel);

```

```

//Switch back to regular register set
setPHYRegister(0x1F, 0x0000, PHYChannel);
}

//function to set ethernet PHY into Ethernet Packet Generation mode
void EPG_on(void)
{
print("\r\nStart EPG\r\n");
//Switch to PHY's extended register set
setPHYRegister(0x1F, 0x0001, 0x0);
//set EPG data
setPHYRegister(0x1E, 0xACF0, 0x0);
//Turn PHY's EPG on
setPHYRegister(0x1D, 0xC7C0, 0x0);
//Switch back to regular register set
setPHYRegister(0x1F, 0x0000, 0x0);
}

//function to set ethernet PHY into erroneous Ethernet Packet Generation mode
void EPG_on_bad(void)
{
print("\r\nStart EPG with invalid checksums\r\n");
//Switch to PHY's extended register set
setPHYRegister(0x1F, 0x0001, 0x0);
//set EPG data
setPHYRegister(0x1E, 0xAAAA, 0x0);
//Turn PHY's EPG on
setPHYRegister(0x1D, 0xC7C1, 0x0);
//Switch back to regular register set
setPHYRegister(0x1F, 0x0000, 0x0);
}

//function to exit ethernet PHY from Ethernet Packet Generation mode
void EPG_off(void)
{
print("\r\nStop EPG\r\n");
//Switch to PHY's extended register set
setPHYRegister(0x1F, 0x0001, 0x0);
//Turn PHY's EPG off
setPHYRegister(0x1D, 0x0000, 0x0);
//Switch back to regular register set
setPHYRegister(0x1F, 0x0000, 0x0);
}

```

## gemac functions.c

```
#include "xgemac_example.h"
```

```
/* ***** Variable Definitions ***** */
```

```
static JumboFrame TxFrame;    /* Frame used to send with */
static JumboFrame RxFrame;    /* Frame used to receive data */

static JumboFrame MyFrame;    /* Frame used to receive data */

static volatile int RxMutex;  /* Shared var used for receive sequencing */
static volatile int TxMutex;  /* Shared var used for transmit sequencing */
static XStatus ErrorCode;     /* Shared var set when the error handler is called */
static XStatus RxStatus;      /* Shared var set when the recv handler is called */
static Xuint32 RxFrameLength; /* Shared var set when the recv handler is called */

XGmac Gemac;                  /* XGmac instance used throughout examples */
char GemacMAC[6] = { 0x00, 0x11, 0xF1, 0x11, 0x11, 0x00 }; /* Local MAC address */
char RemoteMAC[6] = { 0x00, 0x15, 0xC5, 0xC1, 0x15, 0x65 }; /* A remote MAC address */
char BroadcastMAC[6] = { 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF }; /* the Broadcast MAC address */
```

```
/* ***** Function Prototypes ***** */
```

```
int XGmacFifoIntrExample(void);
static void SetupInterrupts(void);
static void ResetMutex(void);
void SendPingPacket(void);
void PrintNextReceivedPacket(void);
static char* ReceivePacket(void);
void ParseEthernetPacket(void);
void PrintIPPacket(char* IPPacket, Xuint32 IPPacketLength);
void PrintARPPacket(char* ARPPacket, Xuint32 ARPPacketLength);
void ARPPacketResponse(char* ARPPacket, Xuint32 ARPPacketLength);
void IPPacketResponse(char* IPPacket, Xuint32 IPPacketLength);
Xuint16 CalculateChecksum(char* Packet, Xuint32 Length);
void SendPacket(char TY1, char TY2, char* OutData, Xuint32 length);
```

```
void ARPRequest(void);
```

```
/* ISR callbacks */
```

```
static void RxHandlerLoopback(void *Callback);
static void TxHandlerLoopback(void *Callback);
static void ErrHandlerLoopback(void *Callback, XStatus ErrCode);
```

```
int XGmacSetup(void)
{
    int i;
    XStatus Status;
    int PayloadSize;
    Xuint32 TxFrameLength;
    char *TxData = (char*)&TxFrame;
```

```

char *RxData = (char*)&RxFrame;

print("\r\nStarting GEMAC\r\n");

/* setup TxFrame header and payload data */
Util_FrameHdrFormatMAC(&TxFrame, BroadcastMAC);
Util_FrameSetPayloadData(&TxFrame);

/*
 * Change the configuration table properties to force: no dma, no counters,
 * no GMII. This can be done whether or not the actual hardware instance
 * contains these features.
 */
Util_UpdateConfigTable(XGE_CFG_NO_DMA, 0, 1);

/* initialize gemac instance */
Status = XGmac_Initialize(&Gmac, XPAR_PLB_GEMAC_0_DEVICE_ID);
if (Status != XST_SUCCESS)
{
    Util_ErrorTrap("Error in initialize");
    return(-1);
}

/* set its mac address */
Status = XGmac_SetMacAddress(&Gmac, (Xuint8*)GmacMAC);
if (Status != XST_SUCCESS)
{
    Util_ErrorTrap("Error setting MAC address");
    return(-1);
}

/* hookup ISRs and callbacks */
SetupInterrupts();

/* start the device */
Status = XGmac_Start(&Gmac);

print("Successfully Started GEMAC\r\n");
/* completed without error */
return(0);
}

/*****
 * Receive handler informs the main thread that a frame has been received.
 * This routine runs in ISR context.
 *
 * Parameters:
 *   Callback - a mutex that signals the main execution thread that a frame has
 *   been received.
 *
 * Returns:

```

```

*   nothing
*
*****/
static void RxHandlerLoopback(void *Callback)
{
    int *Mutex = (int*)Callback;
    print("*");
    RxFrameLength = sizeof(JumboFrame);
    RxStatus = XGmac_FifoRecv(&Gmac, (Xuint8 *)&RxFrame, &RxFrameLength);
    *Mutex = 1;
}

/*****
* TxHandlerLoopback
*
* Informs the main thread that the FifoSend operation is completed
*
* Parameters:
*   Callback - a mutex that signals the main execution thread that a frame has
*   been transmitted.
*
* Returns:
*   nothing
*
*****/
static void TxHandlerLoopback(void *Callback)
{
    int *Mutex = (int*)Callback;
    *Mutex = 1;
}

/*****
* ErrHandlerLoopback
*
* Increment the error counter so that the main thread knows an error occurred
*
* Parameters:
*   Callback - Not used
*   ErrCode - Saved in shared variable
*
* Returns:
*   nothing
*
*****/
static void ErrHandlerLoopback(void *Callback, XStatus ErrCode)
{
    ErrorCode = ErrCode;
}

/*****
* Setup ISR constructs. When this function returns, the callbacks will have

```



```

* been registered with the GEMAC driver and the ISR for all GEMAC interrupts
* will have been hooked up to the external interrupt signal on the PPC.
*
* Parameters:
*   none
*
* Returns:
*   nothing
*
*****/
static void SetupInterrupts(void)
{
/* Hook up the driver's ISR to the hardware interrupt vector */
#if (TARGET_BSP == BSP_VXWORKS_PPC405)

    /* disable interrupt vector from gemac, then hook its ISR up */
    intDisable(GEMAC_INT_VEC);
    intConnect((VOIDFUNCPTR*)GEMAC_INT_VEC,
               (VOIDFUNCPTR)XGmac_IntrHandlerFifo, (int)&Gmac);

#elif (TARGET_BSP == BSP_EDK_STANDALONE_PPC405)

    static XIntc InterruptController;
    XStatus Status;

    /* initialize the interrupt controller and connect the ISR */
    Status = XIntc_Initialize(&InterruptController, XPAR_OPB_INTC_0_DEVICE_ID);
    if (Status != XST_SUCCESS)
    {
Util_ErrorTrap("Unable to initialize the interrupt controller");
if(Status == XST_DEVICE_IS_STARTED)
{
Util_ErrorTrap(". Device already started.\r\n");
}

        Status = XIntc_Connect(&InterruptController, GEMAC_INT_VEC,
                               (XInterruptHandler)XGmac_IntrHandlerFifo, &Gmac);
        if (Status != XST_SUCCESS)
        {
            Util_ErrorTrap("Unable to connect GEMAC ISR to interrupt controller");
        }

        /* start the interrupt controller */
        Status = XIntc_Start(&InterruptController, XIN_REAL_MODE);
        if (Status != XST_SUCCESS)
        {
            Util_ErrorTrap("Error starting intc");
        }

        /* initialize the PPC405 exception table */

```

```

XExc_Init();

/* register the interrupt controller with the exception table */
XExc_RegisterHandler(XEXC_ID_NON_CRITICAL_INT,
                    (XExceptionHandler)XIntc_InterruptHandler,
                    &InterruptController);

#endif

/* Register the callback handlers with the driver */
XGemac_SetFifoSendHandler(&Gemac, (void*)&TxMutex,
                        (XGemac_FifoHandler)TxHandlerLoopback);
XGemac_SetFifoRecvHandler(&Gemac, (void*)&RxMutex,
                        (XGemac_FifoHandler)RxHandlerLoopback);
XGemac_SetErrorHandler(&Gemac, (void*)0,
                      (XGemac_ErrorHandler)ErrHandlerLoopback);

/* Enable interrupts from the hardware */
#if (TARGET_BSP == BSP_VXWORKS_PPC405)

    intEnable(GEMAC_INT_VEC); /* intc to processor */

#elif (TARGET_BSP == BSP_EDK_STANDALONE_PPC405)

    XIntc_Enable(&InterruptController, GEMAC_INT_VEC); /* gemac to intc */
    XExc_mEnableExceptions(XEXC_NON_CRITICAL); /* intc to processor */

#endif
}

/*****
* Clear the Tx and Rx mutexes and other shared vars for the next pass.
*
* Parameters:
*   none
*
* Returns:
*   nothing
*
*****/
static void ResetMutex(void)
{
    /* disable interrupts */
    #if (TARGET_BSP == BSP_VXWORKS_PPC405)

        int key = intLock();

    #elif (TARGET_BSP == BSP_EDK_STANDALONE_PPC405)

```

```

        XExc_mDisableExceptions(XEXC_NON_CRITICAL);

#ifdef
    /* reset mutexes and shared vars */
    TxMutex = 0;
    RxMutex = 0;
    RxStatus = XST_SUCCESS;
    ErrorCode = XST_SUCCESS;

    /* enable interrupts */
    #if (TARGET_BSP == BSP_VXWORKS_PPC405)

        intUnlock(key);

    #elif (TARGET_BSP == BSP_EDK_STANDALONE_PPC405)

        XExc_mEnableExceptions(XEXC_NON_CRITICAL);

    #endif
}

void PrintNextReceivedPacket(void)
{
    int i;
    char* RxPacket;
    RxPacket = ReceivePacket();
    print("Received:\r\n");
    for (i = 0; i < RxFrameLength; i++)
    {
        xil_printf("%x", RxPacket[i]);
    }
}

char* ReceivePacket(void)
{
    //char *RxData = (char*)&RxFrame;
    ResetMutex();
    while(RxMutex == 0)
    {
    }
    return (char*)&RxFrame;
}

//Funtion to parse an ethernet packet and print it to the terminal
void ParseEthernetPacket(void)
{
    int i;
    char* RxPacket;
    char* IPPacket;
    RxPacket = ReceivePacket();

```

```

print("\r\n");
print("Destination MAC address: ");
for (i = 0; i < 6; i++){xil_printf("%x",RxPacket[i]);} print("\r\n");

print("Source MAC address: ");
for (i = 6; i < 12; i++){xil_printf("%x",RxPacket[i]);} print("\r\n");

print("Type/Length: ");
xil_printf("%x",RxPacket[12]); xil_printf("%x",RxPacket[13]); print("\r\n");

//If the packet is an ARP packet
if(RxPacket[12]==0x08 && RxPacket[13]==0x06)
{
  IPPacket = &(RxPacket[14]);
  ARPPacketResponse(IPPacket,RxFrameLength-18);
}
//If the packet is an IP packet
else if(RxPacket[12]==0x08 && RxPacket[13]==0x00)
{
  IPPacket = &(RxPacket[14]);
  IPPacketResponse(IPPacket,RxFrameLength-18);
}
//All other types of packets
else
{
  print("Data: ");
  for (i = 14; i < RxFrameLength-4; i++)
  {
    xil_printf("%x",RxPacket[i]);
  }
}
print("\r\n");

print("CRC-32: ");
for (i = RxFrameLength-4; i < RxFrameLength; i++)
{
  xil_printf("%x",RxPacket[i]);
}
print("\r\n");
}

//function to print an IP packet
void PrintIPPacket(char* IPPacket, Xuint32 IPPacketLength)
{
  int i;
  print("IP Packet: \r\n");

  print("\tVersion/IHL: ");
  xil_printf("%02x",IPPacket[0]); print("\r\n");

  print("\tType of Service: ");

```

```

xil_printf("%02x",IPPacket[1]); print("\r\n");

print("\tTotal Length: ");
xil_printf("%02x",IPPacket[2]); xil_printf("%02x",IPPacket[3]); print("\r\n");

print("\tIdentification: ");
xil_printf("%02x",IPPacket[4]); xil_printf("%02x",IPPacket[5]); print("\r\n");

print("\tFlags/Offset: ");
xil_printf("%02x",IPPacket[6]); xil_printf("%02x",IPPacket[7]); print("\r\n");

print("\tTime To Live: ");
xil_printf("%02x",IPPacket[8]); print("\r\n");

print("\tProtocol: ");
xil_printf("%02x",IPPacket[9]); print("\r\n");

print("\tHeader Checksum: ");
xil_printf("%02x",IPPacket[10]); xil_printf("%02x",IPPacket[11]); print("\r\n");

print("\tSource IP Address: ");
for(i=12;i<16;i++){xil_printf("%d.",IPPacket[i]);} print("\r\n");

print("\tDestination IP Address: ");
for(i=16;i<20;i++){xil_printf("%d.",IPPacket[i]);} print("\r\n");

print("\tOptions: ");
for (i = 20; i < IPPacketLength; i++)
{
    xil_printf("%02x",IPPacket[i]);
}

//function to print an ARP packet
void PrintARPPacket(char* ARPPacket, Xuint32 ARPPacketLength)
{
    int i;

    print("APR Packet: \r\n");

    print("\tHardware Type: ");
    xil_printf("%02x",ARPPacket[0]); xil_printf("%02x",ARPPacket[1]); print("\r\n");

    print("\tProtocol Type: ");
    xil_printf("%02x",ARPPacket[2]); xil_printf("%02x",ARPPacket[3]); print("\r\n");

    print("\tHardware Length: ");
    xil_printf("%02x",ARPPacket[4]); print("\r\n");

    print("\tProtocol Length: ");
    xil_printf("%02x",ARPPacket[5]); print("\r\n");

```

```

print("\tOperation: ");
xil_printf("%02x",ARPPacket[6]); xil_printf("%02x",ARPPacket[7]); print("\r\n");

print("\tSender Hardware Address: ");
for(i=8;i<14;i++){xil_printf("%02x",ARPPacket[i]);} print("\r\n");

print("\tSender Protocol Address: ");
for(i=14;i<18;i++){xil_printf("%d.",ARPPacket[i]);} print("\r\n");

print("\tTarget Hardware Address: ");
for(i=18;i<24;i++){xil_printf("%02x",ARPPacket[i]);} print("\r\n");

print("\tTarget Protocol Address: ");
for(i=24;i<28;i++){xil_printf("%d.",ARPPacket[i]);} print("\r\n");

print("\tLeftovers: ");
for (i = 28; i <ARPPacketLength; i++)
    { xil_printf("%02x",ARPPacket[i]);}
}

//function to generate an appropriate ARP packet response
void ARPPacketResponse(char* ARPPacket, Xuint32 ARPPacketLength)
{
    int i;
    char *ARPOutData;
    Xuint8 selfname;
    ARPOutData=(char*)malloc(sizeof(char)*28);

    //PrintARPPacket(ARPPacket, ARPPacketLength);

    //copy incomingtype and length data
    for (i = 0; i < 6; i++)
    {
        ARPOutData[i] = ARPPacket[i];
    }
    //set ARP operation to reply
    ARPOutData[6]=0x00;
    ARPOutData[7]=0x02;

    //copy sender MAC address from PHY
    for (i = 8; i < 14; i++)
    {
        ARPOutData[i] = GemacMAC[i-8];
    }

    //copy sender IP address from the requested target
    for (i = 14; i < 18; i++)
    {
        ARPOutData[i] = ARPPacket[i+10];
    }
}

```

```

    //copy target MAC address from the sender MAC
    for (i = 18; i < 28; i++)
    {
        ARPOutData[i] = ARPPacket[i-10];
    }

    //determine id sender is asking for its own name
    selfname = 1;
    for (i = 14; i < 18; i++)
    {
        if(ARPPacket[i] != ARPPacket[i+10])
        {
            selfname = 0;
        }
    }

    //PrintARPPacket(ARPOutData,28);
    if(selfname == 0)
    {
        SendPacket (0x08,0x06,ARPOutData,28);
    }
}

//function to generate an appropriate IP packet response
void IPPacketResponse(char* IPPacket, Xuint32 IPPacketLength)
{
    int i;
    char *IPOutData;
    Xuint16 TotalLength;
    Xuint8 HeaderLength;
    Xuint16 Checksum;

    HeaderLength = ((IPPacket[0])<<4)>>2; //Number of bytes in header
    TotalLength = IPPacket[2]*256 + IPPacket[3];

    IPOutData=(char*)malloc(sizeof(char)*TotalLength);

    // xil_printf("Header Length: %d , Total Length: %d",HeaderLength,TotalLength);
    // PrintIPPacket(IPPacket, TotalLength);

    if(IPPacket[9]==0x1) //ICMP Ping packet response
    {
        //copy incoming type,length, protocol, etc.
        for (i = 0; i < 10; i++)
        {
            IPOutData[i] = IPPacket[i];
        }

        //swap sender IP address and target IP address
        for (i = 12; i < 16; i++)

```

```

{
    IPOutData[i] = IPPacket[i+4];
    IPOutData[i+4] = IPPacket[i];
}

IPOutData[10] = 0;
IPOutData[11] = 0;

//IP Packet Checksum
Checksum = CalculateChecksum(IPOutData,HeaderLength);
IPOutData[10]=(char)(Checksum>>8);
IPOutData[11]=(char)((Checksum<<8)>>8); //because I don't know how to mask

//copy payload
for (i = HeaderLength; i < TotalLength; i++)
{
    IPOutData[i] = IPPacket[i];
}

//set ICMP operation to reply
IPOutData[20]=0x00;
IPOutData[21]=0x00;
//set checksum to 0
IPOutData[22]=0x00;
IPOutData[23]=0x00;

//ICMP Checksum
Checksum = CalculateChecksum(&(IPOutData[HeaderLength]),TotalLength-HeaderLength);

// xil_printf("\r\nChecksum = %04x\r\n",Checksum);
IPOutData[22]=(char)(Checksum>>8);
IPOutData[23]=(char)((Checksum<<8)>>8); //because I don't know how to mask

// PrintIPPacket(IPOutData,Length+2);
SendPacket (0x08,0x00,IPOutData,TotalLength);
}

if(IPPacket[9]==17 && IPPacket[28]==169) //UDP packet
{
    int j;
    Xuint32 TestDataLength = 1024;
    Xuint32 NumberOfPacketsToSend = 1024;
    for(j=0; j<2; j++)
    {

        if(j==1){ TestDataLength = 4;}

        TotalLength = HeaderLength + 8 + TestDataLength;
        //copy incoming type,length, protocol, etc.
        for (i = 0; i < 10; i++)

```



```

{
IPOutData[i] = IPPacket[i];
}

//Set length of UDP packet
IPOutData[2]=TotalLength>>8;
IPOutData[3]=(TotalLength<<8)>>8;

//swap sender IP address and target IP address
for (i = 12; i < 16; i++)
{
IPOutData[i] = IPPacket[i+4];
IPOutData[i+4] = IPPacket[i];
}

//Set IP Packet Checksum to zero for calculation
IPOutData[10] = 0;
IPOutData[11] = 0;

//Calculate and stuff IP Packet Checksum
Checksum = CalculateChecksum(IPOutData,HeaderLength);
IPOutData[10]=(char)(Checksum>>8);
IPOutData[11]=(char)((Checksum<<8)>>8); //because I don't know how to mask

//set source port to incoming destination port
IPOutData[20]=IPPacket[22];
IPOutData[21]=IPPacket[23];
//set destination port to incoming source port
IPOutData[22]=IPPacket[20];
IPOutData[23]=IPPacket[21];

//Set length of UDP packet
IPOutData[24]=(8+TestDataLength)>>8;
IPOutData[25]=((8+TestDataLength)<<8)>>8;

//Set checksum
IPOutData[26]=0;
IPOutData[27]=0;

for(i=28; i < TotalLength; i++)
{
IPOutData[i]=i;
}

for(i=j*(NumberOfPacketsToSend-1); i<NumberOfPacketsToSend+1; i++)
{
SendPacket (0x08,0x00,IPOutData,TotalLength);
}
}
}
}
}

```

```

//function to generate a 16 bit checksum
Xuint16 CalculateChecksum(char* Packet, Xuint32 Length)
{
    Xuint32 Checksum;
    Xuint32 i;
    //calculate checksum
    Checksum = 0;
    for (i = 0; i < Length; i=i+2)
    {
        Checksum+= ((int)Packet[i]*256)+((int)Packet[i+1]);
        if(Checksum > 0xFFFFE)
        {
            Checksum++;
            Checksum = Checksum - 0x10000;
        }
        // if(Checksum > 0x0) //Used in UDP Checksum
        // {
        //     Checksum = 0xFFFF;
        // }
    }
    return (~Checksum);
}

//funtion to send an ethernet packet
void SendPacket(char TY1, char TY2, char* OutData, Xuint32 length)
{
    int i;
    char* TxData= (char*)&TxFrame;
    Util_FrameHdrFormatType(&TxFrame, length);
    Util_FrameHdrFormatMAC(&TxFrame, RemoteMAC);

    //Ethertype/Length
    TxData[12]=TY1;
    TxData[13]=TY2;

    for (i = 0; i < length; i++)
    {
        TxData[i+14] = OutData[i];
    }

    XGmac_FifoSend(&Gmac, (Xuint8 *)&TxFrame, XGE_HDR_SIZE + length);
}

//funtion to form and send an ARP packet
void ARPRequest(void)
{
    int i;
    char *ARPOutData;
    ARPOutData=(char*)malloc(sizeof(char)*28);

```

```

//incomingtype and length data
ARPOutData[0] = 0x00;
ARPOutData[1] = 0x01;
ARPOutData[2] = 0x08;
ARPOutData[3] = 0x00;
ARPOutData[4] = 0x06;
ARPOutData[5] = 0x04;

//set ARP operation to request
ARPOutData[6]=0x00;
ARPOutData[7]=0x01;

//copy sender MAC address from PHY
for (i = 8; i < 14; i++)
{
ARPOutData[i] = GemacMAC[i-8];
}

//copy sender IP address from the requested target
ARPOutData[14] = 169;
ARPOutData[15] = 254;
ARPOutData[16] = 162;
ARPOutData[17] = 10;

//destination MAC
ARPOutData[18] = 0x00;
ARPOutData[19] = 0x00;
ARPOutData[20] = 0x00;
ARPOutData[21] = 0x00;
ARPOutData[22] = 0x00;
ARPOutData[23] = 0x00;

//copy target IP from the sender IP
for (i = 24; i < 28; i++)
{
ARPOutData[i] = ARPOutData[i-10];
}

// PrintARPPacket(ARPOutData,28);
SendPacket (0x08,0x06,ARPOutData,28);
}

```

## APPENDIX D

### C# Code for PC Ethernet Control

## DreamTestCs.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;
using System.Net.Sockets;
using System.Net;
using System.IO;

namespace DreamTestCS
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            //IP address of the DREAM board
            Byte[] dreamIP = new Byte[4];
            dreamIP[0] = (Byte)169;
            dreamIP[1] = (Byte)254;
            dreamIP[2] = (Byte)162;
            dreamIP[3] = (Byte)10;

            //Definition of the "Send" command
            Byte[] runWord = new Byte[1];
            runWord[0] = (Byte)169;

            //Initialize variables to be filled with data received from the DREAM board
            int RecvLength = 0;
            FileStream FileOut = new FileStream("dataout.txt", FileMode.Append);
            IPAddress destAddr = new IPAddress(dreamIP);
            Byte[] RecvBytes = new Byte[2048];
            Socket socket = null;

            //Initialize the ethernet socket
            IPEndPoint endpoint = new IPEndPoint(destAddr, 0);
            SocketAddress socketaddress = endpoint.Serialize();
            socket = new Socket(endpoint.AddressFamily, SocketType.Dgram, ProtocolType.Udp);

            try
            {
                socket.Connect(destAddr, 80);

                //Disable Send Command Button
            }
        }
    }
}
```

```

        button1.Enabled = false;

        //Send the "Send" command to the DREAM board
        socket.Send(runWord);
        socket.ReceiveTimeout = 1000;

        //Retreive the packet received through the ethernet
        RecvLength = socket.Receive(RecvBytes);

        //Output to the textbox the number of packets which were received
        textBox1.Text = "Received " + RecvLength + " Packets ";

        //Write the data received from the DREAM board to a file
        while (RecvLength != 4)
        {
            RecvLength = socket.Receive(RecvBytes);
            FileOut.Write(RecvBytes, 0, RecvLength);

            //Output to the textbox the contents of each received packet
            //for (int i = 0; i < RecvLength; i++)
            //{
            //    textBox1.Text = textBox1.Text + (int)RecvBytes[i] + " ";
            //}

            socket.Disconnect(true);
        }
        catch (SocketException)
        {
        }
        //Re-enable the Send Command button
        button1.Enabled = true;
        FileOut.Close();
    }
}

```

R002594737

## BIBLIOGRAPHY

- [1] Gordon Brebner. "Single-chip Gigabit Mixed-version IP Router on Virtex-II Pro". In *Proc. of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, Edinburgh, United Kingdom, 2002.
- [2] Kendall Castor-Perry and Tamara Schmitz. "Know the Sometimes-Surprising Interactions in Modeling a Capacitor-Bypass Network". 2007.
- [3] Inc. Cisco Systems. "*Cisco Networking Academy Program: CCNA 1 and 2 Companion Guide*". Cisco Press, Indianapolis, IN, 2005.
- [4] Robert Resnick David Halliday and Jearl Walker. "*Fundamentals of Physics*". John Wiley & Sons, Inc., New York, NY, 2001.
- [5] John A. DeFalco. "Reflection and Crosstalk in Logic Circuit Interconnections". 1970.
- [6] IBM. "CoreConnect Bus Architecture". 1999.
- [7] C. Timossi E. Williams J. Weber, M Chin. "Hardware and Software Development and Integration in an FPGA Embedded Processor Based Control System Module for the ALS\*". In *Proc. of PAC07, Albuquerque, New Mexico, USA*, LBNL, Berkeley, CA 94720, 2007.
- [8] Jaesung Lee, Hyuk-Jae Lee, and Kyoung Park. "An Efficient Implementation of the InfiniBand Link Layer". 2003.
- [9] Lee W. Richey. *Right The First Time - A Practical Handbook on High Speed PCB and System Design*. Speeding Edge, 2003.
- [10] Vitesse. "Quad 10/100/1000BASE-T PHY with SGMII/SerDes MAC I/F". *VSC8234 datasheet*, Aug. 2004 [Revised Mar. 2005].
- [11] Xilinx. "Power Distribution System (PDS) Design: Using Bypass/Decoupling Capacitors". *Application Note: Virtex-II Series*, Feb. 2005.
- [12] Xilinx. "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet". Jan. 2002 [Revised Jun. 2004].
- [13] Xilinx. "RocektIO Transceiver User Guide". Nov. 2001 [Revised Feb. 2007].
- [14] Xilinx. "LogiCORE Ethernet 1000BASE-X PCS/PMA or SGMII v8.0". Oct. 2006.
- [15] Xilinx. "LogiCORE 1-Gigabit Ethernet MAC v8.1". Sept. 2006.