The Design and Development of a Pulsed Radar Block for the Rhino Platform

by

Bruce Raw

BSc(Eng) Electrical Engineering/

BSc Computer Science and Computational & Applied Mathematics (UCT)

A thesis submitted to the Department of Electrical Engineering,
University of Cape Town, in fulfilment of the requirements
for the degree of

Master of Science

at the

UNIVERSITY OF CAPE TOWN

Supervisors:

Simon Winberg

Michael Inggs

June 2012

Declaration

I know the meaning of plagiarism and declare that all the work in the document, save for that which is properly acknowledged, is my own. It is being submitted for the degree of Master of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author: . . . . . .

Cape Town

09/02/2012

Abstract

The Reconfigurable Hardware Interface for computiNg and radiO (Rhino) Platform is an FPGA based computing platform that was designed at the University of Cape Town, to provide an FPGA resource that is both affordable and easy to use and learn. Rhino was designed to be used in research and skills development in the areas of Software Designed Radio, Radio Astronomy and Cognitive Radio.

In order to aid development on the Rhino platform, applications have been created to program the FPGA and manage the communications bus. These tools will allow the user to program the FPGA from the ARM processor and transfer files from the ARM to the FPGA. This allows the user to control a gateware application from the ARM.

A framework comprising reusable radar processing modules (referred to as "Radar Blocks" in this text) has been implemented on the Rhino that allows users to use the Rhino to control simple pulse radar. The individual gateware modules of this block each have their own set of configurable parameters which can be changed during runtime from the ARM processor.

A waveform is stored on the file system used by the ARM processor, the user then uses the control pc to copy the waveform to the FPGA. The pulse radar application is implemented on the FPGA using the radar blocks framework, which as mentioned above allows each radar block to be configured from the ARM processor in order to adapt settings during experiments.

The scripts and support tools that were developed during the project allow the Rhino to be easily used for teaching and research projects, with minimal effort required. Blocks are now available for the communications bus, Gigabit Ethernet and simple pulse radar.

Acknowledgements

I would like to express my gratitude to the following people who assisted me during the course of this thesis.

Michael Inggs for overseeing the project, providing invaluable information on radar systems and sourcing funding.

Alan Langman for guiding me through the learning process at the beginning of the project.

Simon Winberg for supervising my project, and making sure I completed the project, and helping with the editing process.

Amit Mishra for giving me guidance and the motivation to complete the project.

Nomenclature

NAS	Network Attached Storage
FPGA	Field Programmable Gate Array
ARM	Advanced RISC Machine
FTDI	Future Technology Devices International
Rhino	Reconfigurable Hardware Interface for computatioN and radiO
DAC	Digital to Analogue Converter
ADC	Analogue to Digital Converter
SDR	Software Defined Radio
IPC	Inter-Process Communication
FMC	FPGA Mezzanine Connector
GPMC	General Purpose Memory Controller
SPI	Serial Peripheral Interface
GPIO	General Purpose Input/Output
PRF	Pulse Repetition Frequency
PRI	Pulse Repetition Interval
CRC error	Cyclic Redundancy Error
FIFO	First In First Out
IP	Intellectual Property
KBps	Kilobytes per Second
Kbps	Kilobits per Second
Block	Block is used in this dissertation to refer to a gateware module.

Note.

The term "Programming the FPGA" is used in this paper to mean configuring the FPGA with a gateware application using a binary configuration file.

Contents

List of Figures

List of Tables

1 Introduction

This dissertation details the design, development and implementation of pulsed radar system which will run on the Reconfigurable Hardware Interface for computiNg and radiO (Rhino) platform. The focus of this project is to develop software and gateware applications for Rhino, which will allow users to easily set-up the platform and begin using if for pulsed radar applications. The Rhino Project is an Open Source project being run at the University of Cape Town. Its goal is to provide a platform for Software Defined Radio (SDR) which is both easy to use, easy to learn and affordable to a broad audience [1].

Reconfigurable computing has been around for many years, but recently interest has been greatly increased in this field [2]. This has been promoted by advances in FPGA technology, combined with performance bottlenecks being reached in machines using Von Neumann architecture. Reconfigurable computing has the ability to greatly accelerate a wide variety of applications by performing computations in hardware, while still maintaining the flexibility of software.

The Rhino platform is an FPGA-based platform that uses a combination of an ARM processor and a Spartan 6 FPGA. In order for the Rhino platform to be useful, it needs to have tools which will allow the user to easily configure the board to perform the tasks required.

This chapter covers the problem description and objectives of the project, as well as the research focus and motivation. User requirements for the project as well as the functionality and performance requirements are discussed.

The dissemination strategy and scope of the project are explained before the final section of this chapter outlines the development of the remainder of this dissertation.

1.1 Problem Description

The Rhino Platform is a newly created FPGA-Based processing platform. While the platform's hardware is capable of handling a large range of tasks, software and gateware tools had yet to be developed for the Rhino at the outset of this project.

1.2 Objective

In this project a gateware application to allow the Rhino to control pulsed radar was developed along with supporting software. The supporting software allowed the user to program the FPGA and control the radar system.

1.3 Research Focus and Motivation

The focus of this research is on the development of a working radar block for the Rhino. The radar block's functionality depends on many other components to work as a system; these components, including the radar block itself, can be used as building blocks for many other systems. A functioning radar block not only provides useful functionality to the Rhino, but also implies that there is a solid framework of resources on which future projects can be developed.

1.4 Methodology Outline

The methodology for this project follows an 11 step process structure (shown inFigure 1-1: Overview of System Development process) similar to the Spiral development model [36]. Every step moves further out the 'spiral' towards a prototype that is better understood and offers improved functionality. Each step is elaborated in the discussion that follows.

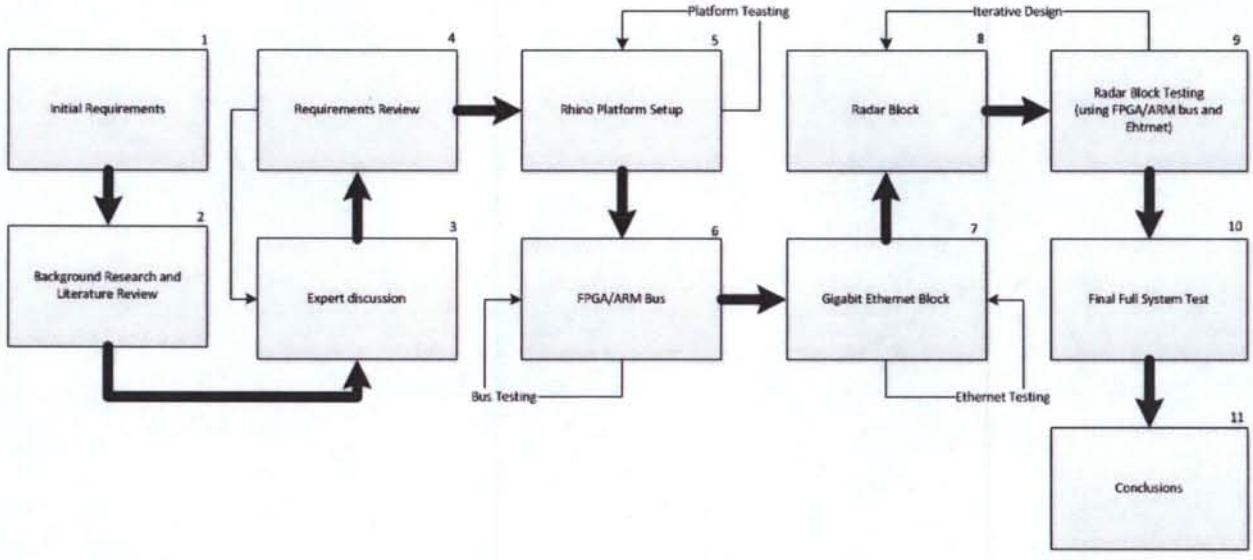

Figure 1-1: Overview of System Development process

The process starts in step #1 by establishing initial requirements and moves on in step #2 to a detailed literature review. Step #3 involves discussion with experts in the field to further examine the system. The fourth step in the methodology model is to review the initial user requirements to form a better set of requirements. Step #5 was to set up the Rhino platform, in order prepare the platform for the development of applications. Step #6 requires the design and development of gateware and software to control the communications bus between the ARM processor and the FPGA. Step #7 was to design and implement a gateware block that would allow the rhino to output data over Gigabit Ethernet. Step #8 incorporates the design and development of each of the components making up the radar block. In step #9, each individual component in the radar block was tested to see if it meets the performance requirements for the system. The penultimate step, step #10, involved testing the system as a whole. Step #11 is the final step in the project methodology, and it involved the drawing up of conclusions and recommending future work based on the outcomes and results of the project. These steps are discussed in more detail in the methodology chapter.

1.5 Requirements

1.5.1 Project Objectives

At the outset of the project, several objectives were defined, as broad goals for the project. These objectives form the core user requirements for the project.

The first objective was to design and implement a method of programming the FPGA from the ARM processor on the Rhino (Figure 1-2). This programming method will be used as the primary means of programming the FPGA for this project.

The second objective was to design a simple method for controlling the parameters on the FPGA from the ARM processor on the Rhino(seen in Figure 1-2), via the 100M Ethernet interface, using the Linux operating system running on the ARM. This control system was to make use of the bus connecting the General Purpose Memory Controller (GPMC) on the ARM, to the FPGA.

Developing a gateware block to implement pulsed radar on the FPGA of the Rhino board was the third requirement. This radar block, combined with the other developed gateware is to serve as test of the system. This leads to the fourth objective, testing the radar block using loop-back within the FPGA.

The next objective of this project is to develop a gateware block to transfer data out from the radar block out over gigabit Ethernet. This gateware block should be as modular as possible, so that it can be used in applications other that the radar block system.

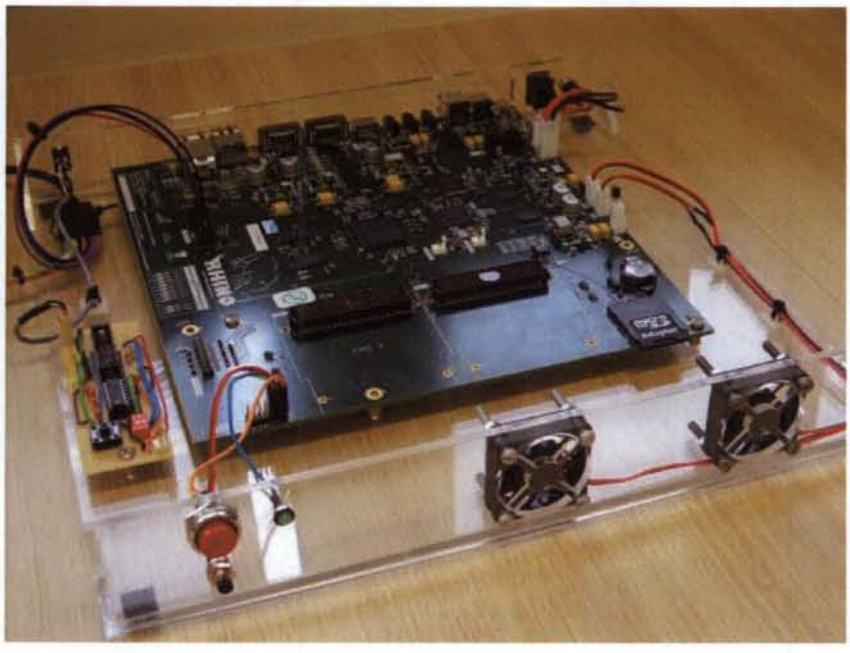

Figure 1-2: Rhino board on a test rig.

1.5.2 User Requirements

Requirements that were given for the project are listed below and labeled from U1 – U12 for reference throughout the text. These user requirements were obtained through an iterative process described in chapter 3.

Supporting Tools:

Using the tools developed in this project the user must be able to:

U1. Program FPGA from ARM

U2. Transfer Data between the ARM and FPGA to modify radar block parameters

Radar Block:

An implementation of the Radar Block gateware must meet the following requirements:

U3. Radar Block parameters must be configurable from ARM.

U4. Pulse Repetition Interval (PRI) between 0 and 1 second, with an accuracy of 1% and resolution steps of 1/1000 of the full range.

U5. Needs to be able to measure Doppler of Mach 2 target in X-Band.
U6. Transmit pulses of up to 8096 complex, 16 bit samples.
U7. Receive up to receive pulses of up to 8096 complex, 16 bit samples.
U8. Delay receiver sampling for a time period up to the length of the PRI.
U9. Control delay length for target with resolution of 750cm.
U10. Use Gigabit Ethernet to output data.
U11. Clear all output buffers before next pulse is received.
U12. Store Output data on Networked Attached Storage (NAS)

1.5.3 System Functionality

From the system requirements listed above, a set of functions is derived; these are labeled F1-F9. These functions form the foundation of the system design. Each function satisfies one or more user requirements, and all the user requirements are satisfied by at least one function.

F1. Program FPGA
This function takes a configuration file from the user, and uses it to program the FPGA over the serial programming interface.

F2. FPGA-ARM Data Bus
Implements the communication between the FPGA and ARM processor and stores in the incoming data in a memory block on the FPGA.

F3. Radar Block Parameters
This function is responsible for storing incoming data from the ARM in the appropriate registers to control the radar block.

F4. Pulse Repetition
Responsible for controlling the pulse repetition rate and triggering the pulse transmission.

F5. Pulse Transmission
Transmits the data to be outputted on the radar through the DAC (DAC is not implemented in this project).

F6. Pulse Reception

Receives data from the ADC (not implemented in this project) and stores it in a memory buffer to await transmission over the network.

F7. Receive Delay

Responsible for controlling the delay between transmission and reception of the pulse.

F8. Gigabit Ethernet

Outputs data out of the system over the gigabit Ethernet port.

F9. Network Storage

Stores data received from the Rhino for later use.

Each of these 9 requirements satisfies one or more of the user requirements; Table 1-1 summarizes the functions and their related requirements.

Table 1-1: Table showing summary of system functions defined from user requirements

Functionality Summary Table		
Number	Function	Requirement Satisfied
F1	Program FPGA	U1
F2	FPGA-ARM Data Bus	U2
F3	Radar Block Parameters	U3
F4	Pulse Repetition	U4U5
F5	Pulse Transmission	U6
F6	Pulse Reception	U7
F7	Receive Delay	U8U9
F8	Gigabit Ethernet	U10U11
F9	Network Storage	U12

1.5.4 **Performance Requirements**

Each function defined in section 1.5.3 needs to achieve a desired level of performance. The performance requirements for these are discussed below.

F1. Programming the FPGA

Programming the FPGA has only one performance requirement, and this is based on the speed at which it programs the FPGA. The FPGA must be programmable in under 5mins. This relatively long period of time given to complete this task is acceptable because programming the FPGA need only be done once.

F2. FPGA-ARM data bus

The bus between the FPGA and ARM is used to control the parameters of the FPGA, and as such is used regularly by the user. In order for the user to be able to properly control the radar block, the bus needs to be able to fully update the radar block registers in less than one second. This gives a required data transfer rate of approximately 32KBps (304Kbps).

F3. Radar Block Parameters

The function that translates the incoming data parameters is required to store the register values with 100% accuracy. All parameter registers must contain the exact values as transferred from the ARM processor.

F4. Pulse Repetition

The pulse repetition function is used to create the Pulse Repetition Frequency (PRF) of the system by implementing a PRI; this means that there is a performance requirement both in terms of its range, accuracy and resolution.

The maximum unambiguous Doppler available to our system is approximately half the PRF. In order to measure the Doppler of a target at Mach 2 in X-Band, which has a Doppler frequency of approximately 45.3 kHz, a PRF of above 90.6 kHz is required. For this project the PRF

required was selected 100 kHz. This gives the desired range of pulse repetition frequencies is from 1Hz to 100 kHz.

The accuracy requirements on the PRI require that the pulse is accurate to 1%; this implies that the jitter on the clock controlling the PRI counter needs to be less that 1%, which for a 100MHz implies a jitter of less than 100ps.

The resolution requires the user to be able to adjust the PRI in steps of 1/1000 of the full range. This requires that the PRI be adjusted in steps of 1ms.

F5. Pulse Transmission

Pulse transmission needs to be able to handle data transmission at a data transfer rate by the DAC in use. For this project, as no particular DAC is being used, the data transfer rate requirement is set at 400Mb/s.

F6. Pulse Reception

Pulse reception needs to be able to handle incoming data at the same speed that the ADC samples data. For this project, as no particular ADC is being used, the data transfer rate requirement is set at 400Mb/s.

F7. Receive Delay

The receive delay relates directly to the distance of the target, the accuracy and resolution of this delay, determine how precisely the delay can be set. For this project the requirements specify that the user needs to be able to set the delay within an accuracy of 750cm, this implies the delay needs to be modifiable with a resolution of 5ns.

F8. Gigabit Ethernet

The gigabit Ethernet needs to be able to transmit all the data in the output buffers, emptying them completely, before the next pulse is received. If it does not completely empty the buffers

after each pulse, data will enter the buffers faster than it is transmitted out, and data will be lost when the buffers become full.

It is suspected that the Gigabit Ethernet will not be able to meet the speed required to run the system with both maximum data size and maximum PRF.

F9. Network Storage

The network storage application needs to be capable of writing data to disk at a rate which is equal to the rate data is being transmitted from the radar block. If it does not store data fast enough, data will be lost.

1.6 Dissemination Strategy

The main dissemination strategy for this project is in a context of training in the specialist field of using FPGA-based software defined radio and radar systems. Resources that have been produced in the course of this project will feed into both undergraduate coursework and final year BSc projects. Additionally, work done in this project will be built upon by researchers in the Radar and Remote Sensing Research Group. A journal paper will be written discussing the key aspects of this project.

The FPGA programming code has been incorporated into the Rhino port of the Borph operating system, which has been made available through Github.

1.7 Scope and Limitations

The Rhino Platform does not currently have any FMC drivers, and it is not within the scope of this project to create them, therefore testing of the radar block does not use DACs or ADCs. Instead of using DACs and ADCs to test the radar block, output signals that would normally go to the DACs are looped back directly to the input that would normally come from the ADCs.

Ideally the gigabit Ethernet output should use a protocol that ensures that all data is transferred correctly; however due to time constraints this project will use UDP for the transfer of data over the network and store it in a file on the receiving computer.

The project will only cover the basic data transfer operation of the radar; it will not include any signal processing or advanced radar functionality.

1.8 Document Outline

Chapter 1: Introduction

This chapter covers the problem description and objectives of the project, as well as the research focus and motivation. User requirements for the project as well as the functionality and performance requirements are discussed.

The dissemination strategy and scope of the project are explained before the final section of this chapter outlines the development of the remainder of this dissertation.

Chapter 2: Background and Literature Review

This chapter presents the technologies, theories and techniques on which this dissertation builds.

A background on reconfigurable computing is given; discussing the history, advantages, disadvantages and popular applications of reconfigurable computing. The Borph operating system for reconfigurable hardware is discussed, as well as its involvement with the Rhino platform.

The major focus of this chapter is on the Rhino platform. It looks at the Rhino hardware, and discusses the various components that make up the Rhino board as well as why they were

selected and their implications on this project. Some of the components that are focused on include the ARM processor and the Spartan 6 FPGA. The various communication channels on the Rhino are reviewed, examining what interfaces are available for communications both between the various sections of the Rhino board and between the Rhino and other devices. This chapter also reviews platforms that are similar to the Rhino platform.

A very simple review of pulsed radar systems is conducted. This provides an introduction to the basic concepts behind pulsed radar without going into large technical detail. Finally the chapter will look at various interconnection bus architectures, including the wishbone architecture which has been suggested for use in gateware for the Rhino Platform.

Chapter 3: Methodology

This chapter provides an overview of the phases, methods, tools and testing procedures that will be implemented over the course of this project. It covers the process that will be undertaken to get from the project outline and literature review, to the final tested project and conclusions. The methodology follows a process structure shown in Figure 3-1.

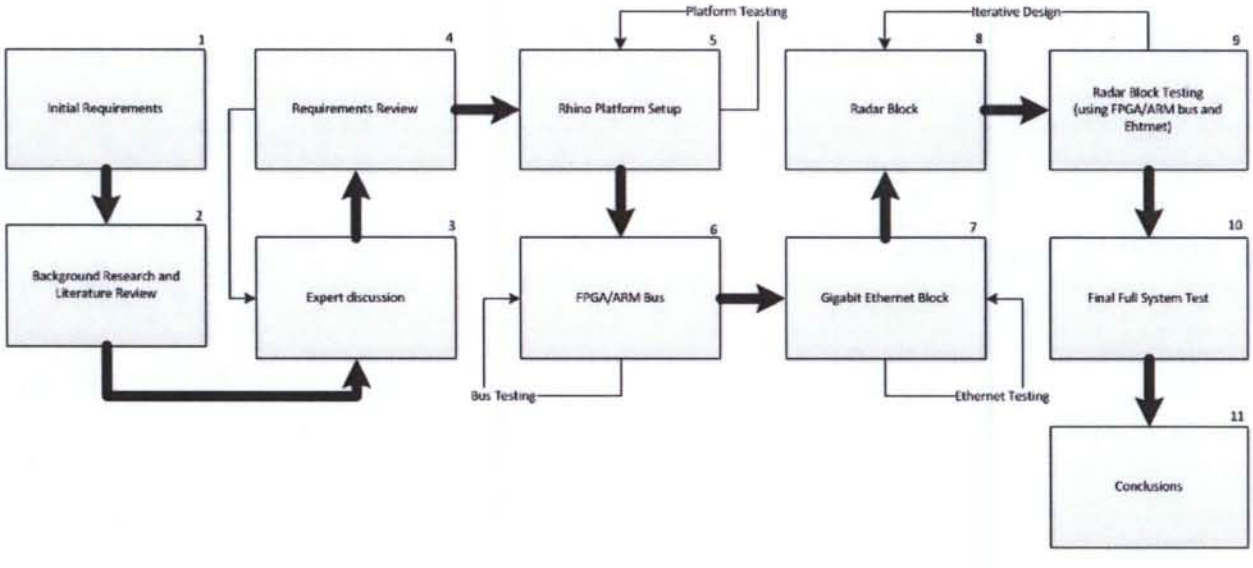

Figure 1-3: Overview of System Development process

Each of the 11 steps in the design methodology is explained in detail, documenting the processes and methods used in each step.

Finally, the system testing process is explained in detail. Testing methods for each function of the system are covered in this section.

Chapter 4: System Design

Chapter 4 gives an overview of the design process for the development of the Rhino System on which the radar block will be implemented.

This chapter starts with a review of the user requirements that were presented in section 1.1. These requirements are reviewed and a final set of system requirements is presented in section 4.1. The parameters for the pulse radar block are discussed and details of the ranges of the radar block parameters are determined.

A high level overview of the system design is presented in Section 4.2, while section 4.3 covers the ARM processor, the setup and configuration of the ARM processor together with design of the programming interface that allows users to program the Spartan 6 FPGA from the ARM processor is discussed.

This chapter also covers the design of the supporting blocks for the radar system, such as the Gigabit Ethernet block.

Chapter 5: Radar Block Design and Implementation

The Radar block is one of the primary outputs of this project, in this chapter the design process for the Radar block will be viewed and discussed. The radar block is intended to be used as a stepping stone for the development of radar systems on the Rhino. While unfortunately actual ADCs and DACs will not be available for this project, it should still serve as a good proof of concept

The aspects of the Radar block, such as the parameters that it requires, the timing and control of the system will be described. The process of setting the parameters, starting and stopping the radar block and reasons for the selections of these processes will all be discussed.

The final output of the block is what is most relevant to users, and the way this output is handled will be discussed in the final section of this chapter.

Chapter 6: Results

This chapter documents the results of the tests described in the methodology. Every function is tested individually and the results of the tests are discussed. All the components passed the functionality tests, but the gigabit Ethernet speed did not reach the requirements.

The final system is also tested, to determine whether or not it meets the requirements outline for the project. It was found that the maximum PRF that the system could manage was based on the pulse width, as there is a limit on the rate at which data can be output.

Chapter 7: Conclusions and Future Work

This chapter covers the conclusions that are drawn regarding the success of the project. The testing results of the final system are discussed and compared to the system specifications to determine whether or not the requirements have been met.
Each of the original objectives given is discussed in relation to the final output system of the project.

In the final section of this chapter recommendations are made for future work.

2 Background & Literature Review

This chapter recaps important technologies, theories and techniques which this dissertation builds upon. It begins with a background to reconfigurable computing; discussing the history, advantages, disadvantages and popular applications of reconfigurable computing. The Borph operating system for reconfigurable hardware is discussed, as well as its involvement with the Rhino platform.

The major focus of this chapter is on the Rhino platform. It looks at the Rhino hardware, and discusses the various components that make up the Rhino board as well as why they were selected and their implications on this project. Some of the components that are focused on include the ARM processor and the Spartan 6 FPGA. The various communication channels on the Rhino are reviewed, examining what interfaces are available for communications both between the various sections of the Rhino board and between the Rhino and other devices. This chapter also reviews platforms that are similar to the Rhino platform.

A very simple review of pulsed radar systems is conducted. This provides an introduction to the basic concepts behind pulsed radar without going into large technical detail. Finally the chapter will look at various interconnection bus architectures, including the wishbone architecture which has been suggested for use in gateware for the Rhino Platform.

2.1 Reconfigurable Computing

Computing historically has been divided primarily into two sections, Hardware and Software [3]. To implement an algorithm in hardware means the development of an Application Specific Integrated Circuit (ASIC). An ASIC is designed for a specific application, and once fabricated cannot be altered. This means that while ASICs provide high performance and efficiency, they are costly to design and are only suited to a single application [5].

The traditional alternative to hardware solutions as mentioned is software. Software applications running on general purpose microprocessors give a large amount of flexibility as the application code can be edited without changing the hardware. This flexibility inevitably has a cost associated with it; in this case it is performance. The performance of software solutions is significantly worse than that of comparable hardware solutions because each instruction must be read from memory, understood, and then only executed, making the process lengthy [3].

One of the core differences between hardware and software is the way they perform computation. The difference between these computations can be described in terms of Spatial and Temporal computation. In Temporal computation instructions are spread over time, with each instruction being executed at a specific time (e.g. as soon as the previous instruction has completed executing). In Spatial computation instructions are spread over a space and can all be executed in parallel [6]. Figure 2-1 shows a comparison between spatial and temporal computation for a simple mathematical expression. The inherent parallelism in temporal computation is what provides for much of the increased performance of hardware implementations.

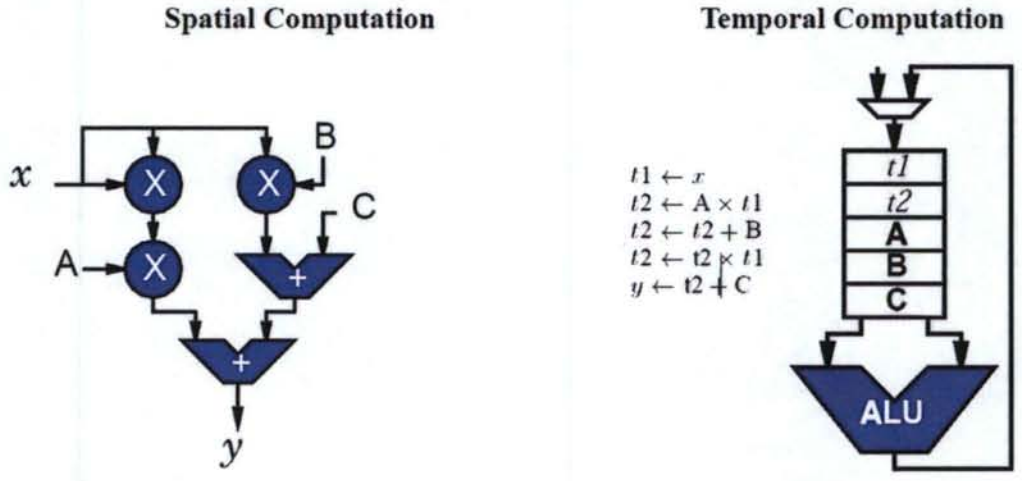

Figure 2-1: Comparison between the implementation of $y = Ax^2 + Bx + C$ in Spatial and Temporal computation [6]

Reconfigurable computing aims to fill the gap between software and hardware implementations by giving the performance of hardware with the flexibility of software [6] . Hardware such as Field Programmable Gate Arrays (FPGA) allows users to configure individual logic cells to program the device to provide the desired functionality [6]. This type of reconfigurable hardware supports spatial computation while having the ability for configuration after fabrication [6]. FPGA development also takes significantly less time than ASIC development as can be seen in Figure 2-2.

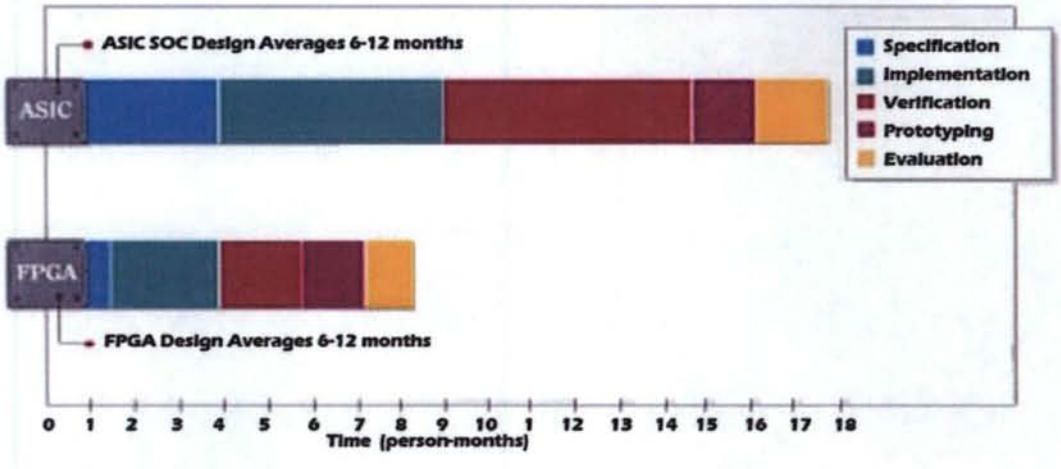

Figure 2-2: Comparison of ASIC and FPGA Design Times [7]

FPGA's provide a significant increase in performance over software while maintaining far more flexibility than is available from hardware. Table 2-1 illustrates the various tradeoffs between performance, flexibility and cost between FPGAs ASICs and Microprocessors.

Table 2-1: Comparison of ASIC, FPGA and Generic Microprocessors [7]

Technology	Performance/Cost	Time until running	Time to high performance	Time to change code functionality
ASIC	Very High	Very Long	Very Long	Impossible
FPGA	Low-Medium	Short	Short	Short
Generic	Low-Medium	Short	Not Attainable	Short

The high performance requirements of modern radar are sometimes not possible in software, while low production volumes, make hardware solutions very expensive. FPGA's provide a solution to this issue [9].

2.2 **BORPH**

BORPH (Berkeley Operating system for Re-Programmable Hardware) is an extended Linux kernel that allows control of FPGA resources as if they were native computational resources [8]. BORPH was developed with the goal of accelerating the development of high level applications on FPGA-based reconfigurable computers [11]. BORPH was designed to run on reconfigurable platforms that consist of a main processor coupled with one or more reconfigurable fabrics [10]. BORPH has been implemented on various FPGA platforms including BEE2 and NetFPGA [13,12]

BORPH tries to a achieve its goals by firstly providing kernel support to FPGA applications, much like a conventional operating system provides support for software applications[10]. Secondly it

abstracts away most of the low level details of the system, allowing users to focus on the actual development of applications [10].

BORPH uses semantics that are already well known, such as UNIX file I/O, this makes it easier for users to adapt and adopt the operating system [14].

Applications running on a reconfigurable fabric connected to BORPH are treated by the operating system as "hardware processes" [10]. These hardware processes are the same as standard UNIX processes, except that instead of being a software program it is actually a gateware design running on an FPGA [14].

The concept of a hardware process not only provides the user with a familiar environment when running applications on the FPGA, but also allows increased communication between processes. Because BORPH treats hardware processes exactly the same as software processes, communications between hardware process and the rest of the system can be achieved through conventional UNIX Inter-Process Communication(IPC)[13].

2.3 **Development of Rhino**

Rhino is an open hardware, FPGA based computing platform that was developed at the University of Cape Town [14]. It was developed with the goal of providing a hardware platform for Software Defined Radio (SDR) that is easy to use, easy to learn and affordable to a broad audience[1].

The Rhino Platform was designed to run the BORPH operating system to control the FPGA, and thus features a microprocessor coupled with an FPGA. The board was designed with the notion that the processor would be used for the control of the system and the FPGA would be used for data processing [14].

The primary applications that were considered during the development of Rhino were; radar, radio astronomy and bioinformatics [14].

2.4 Rhino Hardware

A Xilinx Spartan 6 FPGA and Texas Instruments AM3517 processor (seen in Figure 2-3) provide the two main processing elements for Rhino. These are supported by a wide range of peripherals that enable various communication and data capture interfaces, these peripherals can be seen in Figure 2-4.

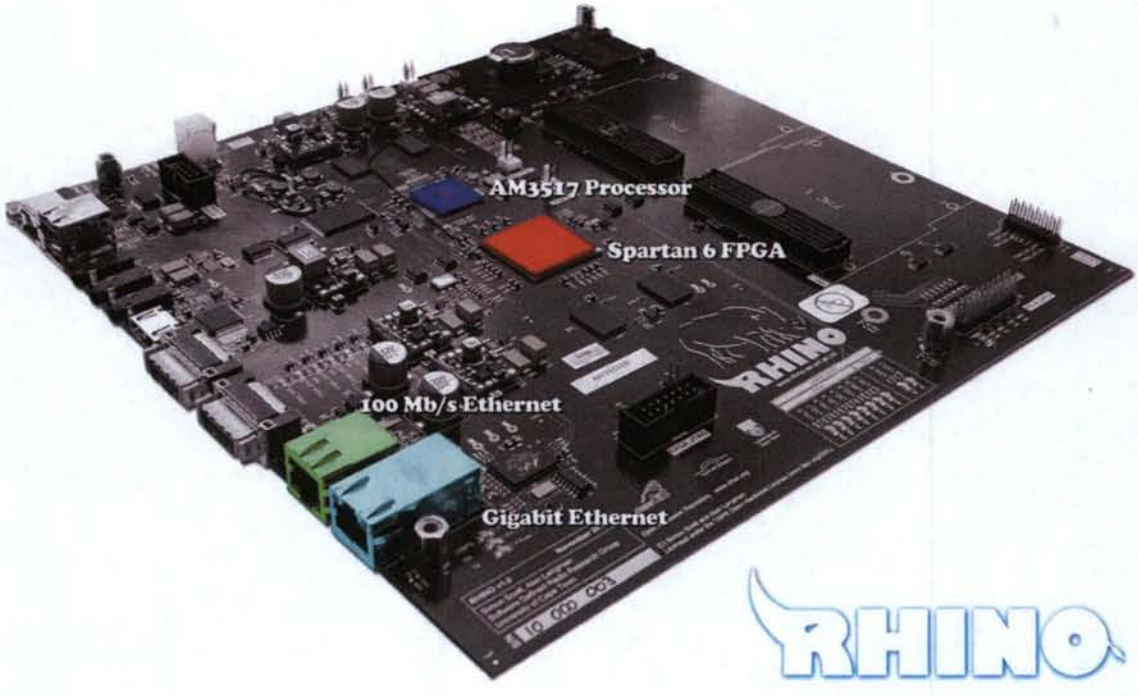

Figure 2-3: Image of the Rhino Platform, highlighting some key components.

The AM3517 from Texas Instruments processor uses an ARM Cortex A-8 processor, and was designed primarily for mobile applications [15]. The ARM processor on the Rhino is responsible for controlling the functionality of the board. It acts as the interface through which the user will interact with Rhino.

The AM3517 allows several possible boot options, allowing it to boot off Ethernet, USB, NAND memory or SD card. These boot options are controlled by two switches on the Rhino board. Varying the two switches gives a total of four possible boot orders.

Rhino features the largest of the Spartan 6 devices, (at the time of its design) the XC6SLX150T. This is the primary computation device on the platform.

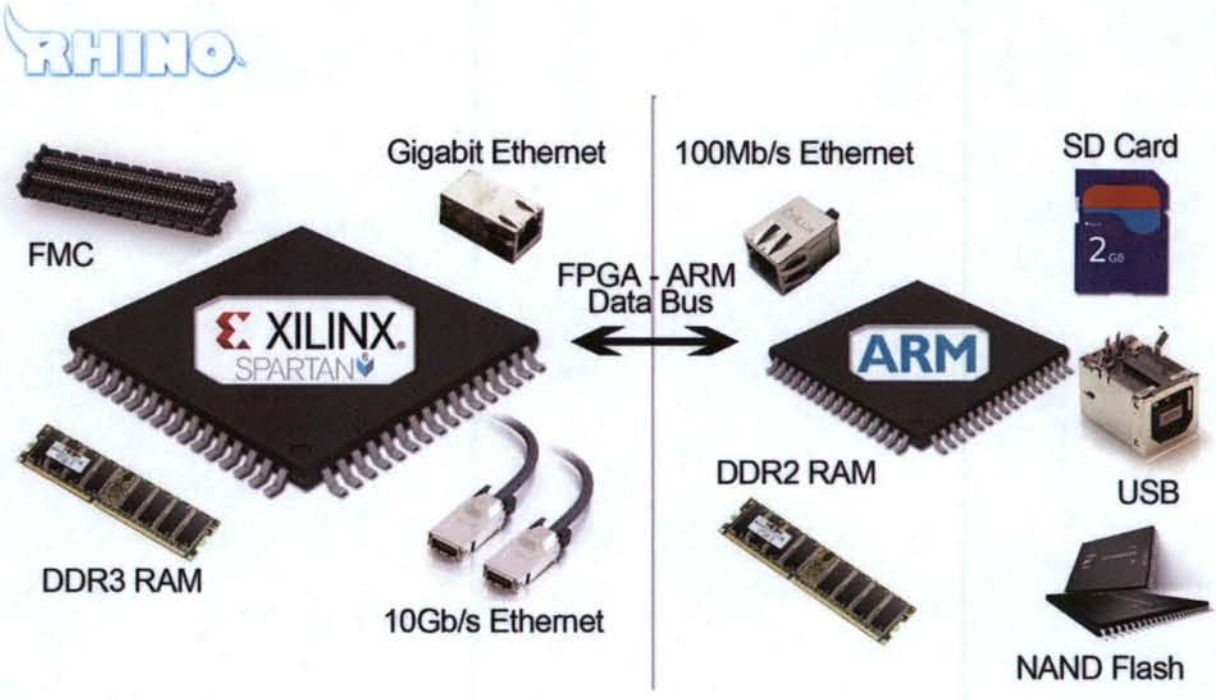

Figure 2-4: Graphical representation of peripherals connected to the Spartan 6 and ARM processor on the Rhino platform

2.4.1 FPGA – ARM communication

The Spartan 6 is connected to the ARM processor through a few different interfaces. The primary interface between the ARM and the Spartan 6 is a FPGA-Processor communication bus (shown in Figure 2-4) that connects the GPMC (General Purpose Memory Controller) on the ARM to the FPGA. This bus handles the bulk of all communications between the FPGA and the ARM processor. The FPGA-Processor bus has 16 bits for data, 10 bits for addressing, 8 CS (Cable Select) lines and various other control lines. The GPMC on the ARM allows for multiplexing of the bus, so that both the 16 data bits and 10 address bits can be used for the address. This allows 26 bits to be used for addressing of data,

Another interface between the ARM and the Spartan 6 is the programming interface. This is used solely to program the FPGA and connects the FPGA_config pin of the Spartan 6 to an SPI (Serial Peripheral Interface) capable GPIO (General Purpose Input/Output) pin on the ARM.

2.4.2 Power Supplies

Power supplies on the board are connected to a management system connected to the ARM processor via an i^2c power management bus. Power supplies for the FPGA are switched off on start-up, and so to use them the power supplies need to be powered up from the ARM. The power for the FMC cards goes through load switches that allow the user to turn the FMC on and off as required, these are off at start-up.

2.4.3 Communications

There are 4 network ports on the Rhino, one 100Mb/s Ethernet port, one Gigabit Ethernet port and two 10Gb/s Ethernet Ports. The 100Mb/s Ethernet port is connected to the ARM processor and is primarily intended to be used to control the platform. It allows the user to telnet into the Linux kernel on the ARM. The other ports are all connected to the Spartan 6, and are intended to be used for the bulk of data transfer in and out of the system.

The Rhino also allows the user to connect to the ARM through a serial link over USB, provided by an FTDI chip.

2.4.4 Expansion Boards

FPGA Mezzanine Card (FMC) connectors are available on the board, and are intended to be used to connect expansion boards such as DACs and ADCs. The Rhino uses the low pin count FMC standard (160 Pins [16]) for general use, there is however an exception to this case. In order to make it possible for an adapter between FMC and the Z-DOK+ used by CASPER, some

additional data pins that would normally only be used for high pin count FMC have been connected [14].

2.5 Alternate Platforms for Software Defined Radio

Rhino is not the only platform built on the principle of an FPGA coupled with a microprocessor. Other systems that are similar to Rhino include the ROACH and USRP platforms.

2.5.1 ROACH

ROACH (Reconfigurable Open Architecture Computing Hardware) is an FPGA platform that is based on the Virtex-5 FPGA from Xilinx [17]. ROACH was designed primarily for radio astronomy, and as such has higher performance than the Rhino hardware and is more expensive. ROACH is used by CASPER (Collaboration for Astronomy Signal Processing and Electronics Research) was created as an upgrade from previous CASPER hardware, combining concepts from previous hardware together in one board[17].

The ROACH was used as a guideline for the development of Rhino, and positive features of the ROACH such as the BORPH operating system it implements, were used for Rhino.

There are features of the roach that were not adopted by the Rhino, for example ROACH uses the Z-DOK+ [17] connector as opposed to the FMC connection used on the Rhino. The ROACH also has four 10 gigabit outputs, while Rhino only has one, the cost of ROACH however can be prohibitive when it comes to teaching and research in SDR.

Figure 2-5: Roach in rack-mountable enclosure [18]

2.5.2 USRP N200

The USRP N200 series by Ettus Research is the successor to the USRP2 series [19]. These boards feature the Spartan3 FPGA, and because they were designed for Software Defined Radio include built in ADC and DAC cards [19]. The USRP N200 uses 16bit 400MSPS DAC and a 14bit 100MSPS ADC [19]. A photo of the USRP N210 can be seen in Figure 2-6.

Figure 2-6: Photo of USRP N210 [19]

The USRP does not have a dedicated processor for control like Rhino and ROACH do, instead it uses a Microblaze processor on the FPGA. The drawback to this approach is that a third of the FPGA's logic and memory resources are used by the Microblaze, meaning that there are fewer resources available for signal processing

2.6 **Pulse Radar**

Rhino is particularly well suited for applications such as controlling a simple pulse radar. Its high performance at a relatively low cost makes it a good option for small research groups wanting to implement a pulse radar.

The most basic principle of pulse radar is that of sending out short pulses, and then receiving echoes after these pulses have reflected off the target. The echoes returned carry information about the target [20]. This process is repeated at a rate defined by the radars Pulse Repetition Frequency (PRF), the PRF is determined primarily by the maximum expected range of targets [21]. The reason for this is that if the next pulse is transmitted before the echoes from the previous pulse have returned there will be ambiguity created as the receiver cannot tell if the echo is from the current pulse or the previous one. A representation of this timing is shown Figure 2-7 where there are two time axes, one showing pulse and echo with time moving slowly, and the other showing how this is repeated with time sped up.

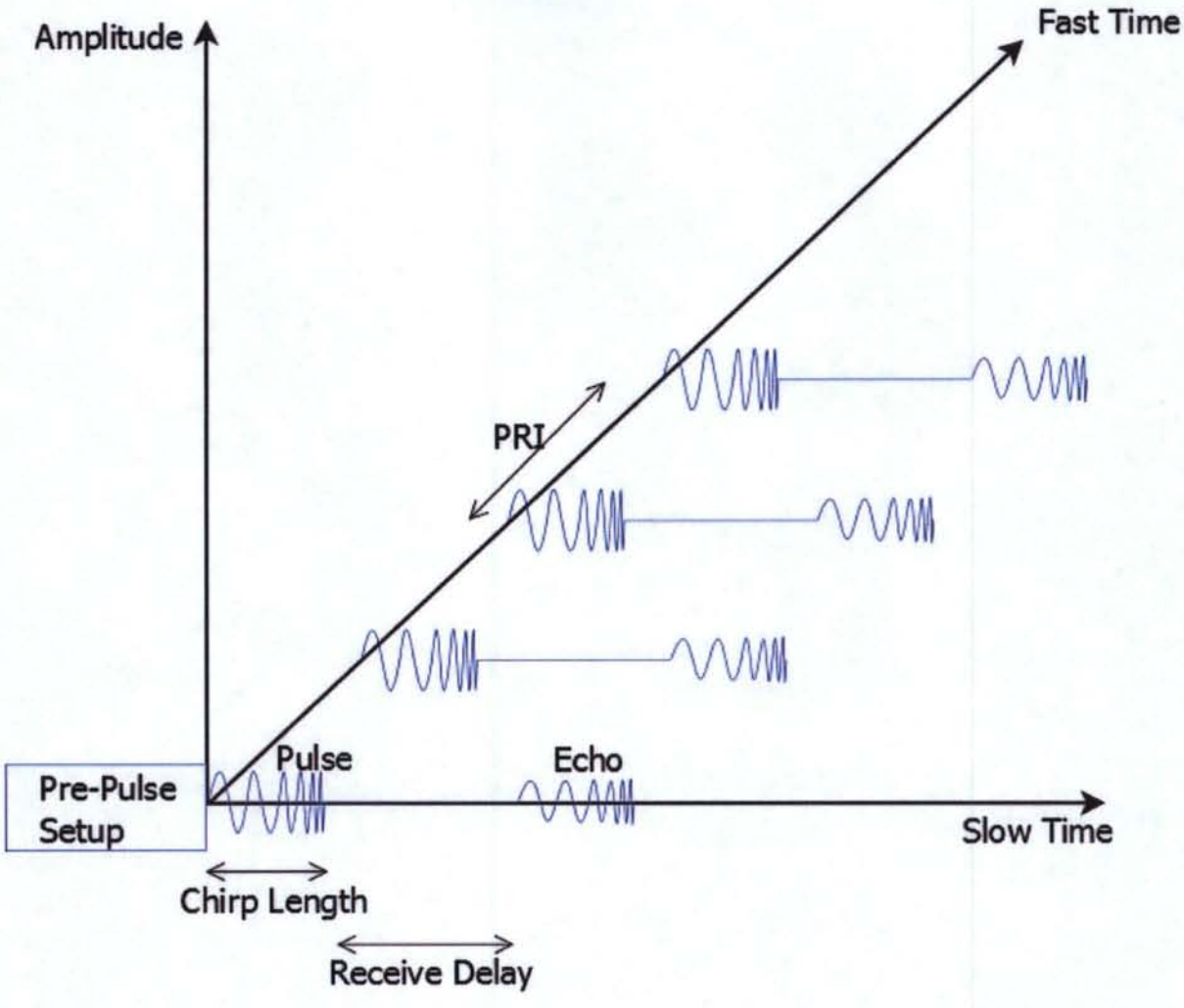

Figure 2-7: Fast/Slow timing diagram for pulse radar.

2.7 Interconnection Busses

Applications such as pulsed radar are seldom comprised of a single element, in the general case an application will use multiple gateware cores and interconnect them to perform the desired task. The use of standardised interconnect busses facilitates the integration of different cores [22]. Among the more popular interconnect busses [23] are Wishbone [24], Avalon [25], AMBA [26] and CoreConnect [27]. In this section these more popular busses will be discussed in brief detail.

2.7.1 Wishbone

The Wishbone System-On-Chip is the interconnect recommended by OpenCores for the FPGA design [28]. It is a completely open standard, and was created to encourage the reuse of designs by creating a common interface between IP cores [29]. Wishbone uses simple, logical and compact interfaces that require very few logic gates [29]. Wishbone is also the suggested interconnect bus for use in Rhino applications.

2.7.2 Avalon

Avalon is an interconnect bus designed by the Altera Corporation, a manufacture of programmable logic devices [32]. The Avalon bus is mainly used in System on Chip (SoC) designs for FPGAs using the NIOS embedded processor [23]. Despite being created Altera, the Avalon interface is an open standard and requires no royalties or licensing fees to develop or sell products using it [32].

2.7.3 AMBA

AMBA (Advanced Microcontroller Bus Architecture) is an interconnect bus standard designed by ARM for use in on-chip interconnection of ARM processor cores [23]. Later versions of AMBA were developed in conjunction the FPGA manufacturer Xilinx, and were developed to be compatible with embedded FPGA design [31]. The AMBA standard offers a choice between three distinct busses. The designer has the option between the Advanced High-performance Bus (AHB), the Advanced System Bus (ASB) and the Advanced Peripheral Bus (APB). AHB is for

high performance systems, while APB is optimised for lower power consumption. The ASB is used for high performance designs that do not require the features of the AHB [32].

2.7.4 CoreConnect

IMB's CoreConnect is an interconnect which was designed for the IBM power Pc, and has since been licensed to over 1500 IP (Intellectual Property) developers [33]. The CoreConnect bus was designed for high performance systems, and as such has an increased complexity which may not be suited to simple embedded applications [32].

2.8 Gigabit Ethernet and Networking

While the 10 Gigabit Ethernet connected to the FPGA is much faster than standard gigabit Ethernet, it also requires less common hardware. There are many applications that have I/O requirements that can be fulfilled by gigabit Ethernet [36], for these applications it makes sense to use the cheaper, more widely available hardware.

Connected to the FPGA is a Gigabit Ethernet PHY chip, this must be connected to a MAC block on the FPGA to control the network interface. As shown in Figure 2-8 below, Ethernet only makes up a small part of the whole network. On top of the basic network there are other layers that still need to be implemented.

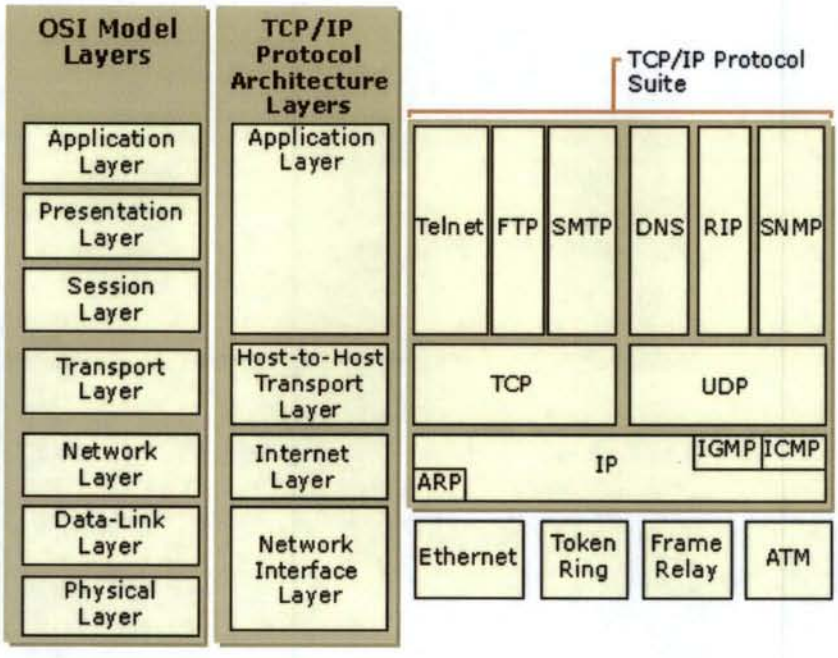

Figure 2-8: Networking Layers [35]

The most common transport layer protocols used are TCP and UDP. TCP provides reliable data delivery, duplicate data suppression and flow control [36], while UDP is a simple protocol that has lower overhead attached.

3 Methodology

This chapter provides an overview of the phases, methods, tools and testing procedures that will be implemented over the course of this project. It covers the process that will be undertaken to get from the project outline and literature review, to the final tested project and conclusions.

The methodology for this project follows an 11 step process structure (shown in Figure 3-1) similar to the Spiral development model [36]. Every step moves further out the 'spiral' towards a prototype that is better understood and offers improved functionality. Each step is elaborated in the discussion that follows.

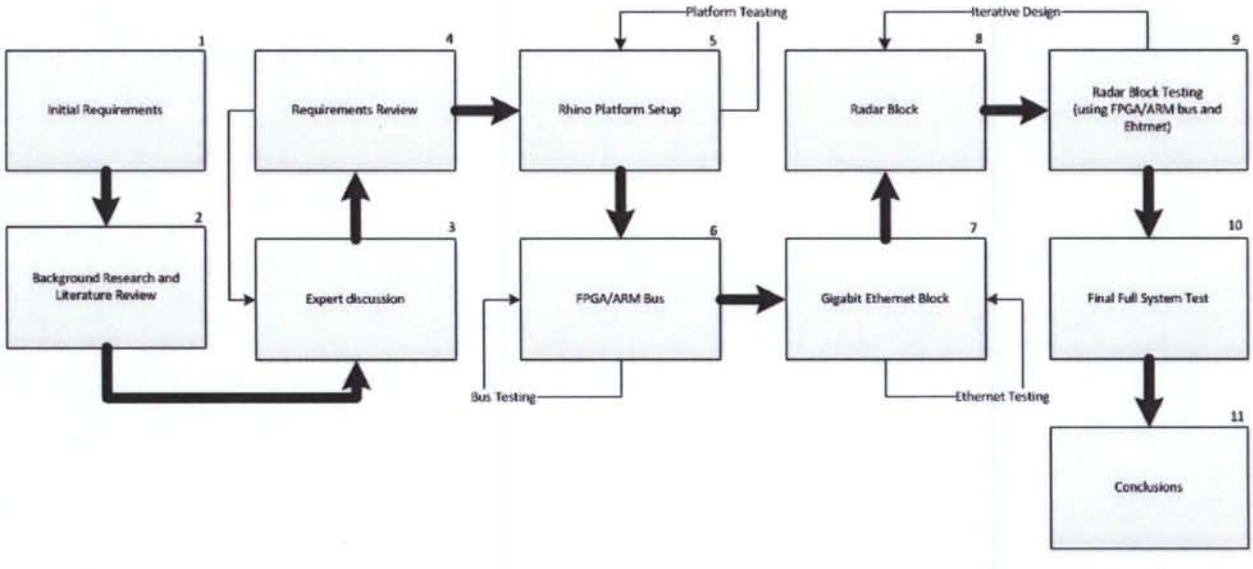

Figure 3-1: Overview of System Development process

The process starts in step #1 by establishing initial requirements and moves on in step #2 to a detailed literature review. Step #3 involves discussion with experts in the field to further examine the system. The fourth step in the methodology model is to review the initial user requirements to form a better set of requirements. Step #5 was to setup the Rhino platform, getting it to a point where development of applications could be run on the platform. Step #6 requires the design and development of gateware and software to control the communications

bus between the ARM processor and the FPGA. Step #7 was to design and implement a gateware block that would allow the rhino to output data over Gigabit Ethernet. Step #8 incorporates the design and development of each of the components making up the radar block. In step #9, each individual component in the radar block was tested to see if it meets the performance requirements for the system. The penultimate step, step #10, involved testing the system as a whole. Step #11 is the final step in the project methodology, and it involved the drawing up of conclusions and recommending future work based on the outcomes and results of the project. These steps are discussed in more detail in the following sections.

3.1 Initial User Requirements

The first step, step #1 in Figure 3-1, in the system development process was to establish the initial requirements for the project. This process involved meeting with the stakeholders of the project in order to discuss desired outcomes. From these discussions initial requirements were developed, with the need for a pulse radar block for the Rhino being identified, and consequently the focus for this project was selected accordingly.

3.2 Background and Literature Review

Step #2 of the methodology was to conduct background research and a literature review. This process created aided in familiarization with the topics of pulse radar and FPGA-based design. While the literature review is shown in Figure 3-1 as a single process, it was actually a continuous process where literature was constantly being consulted throughout the project. The reason it is shown as a single block, is because this is the point at which the bulk of the work was done.

The initial literature review created a good understanding of the current technologies available in the relevant fields and provided a good starting point for further development of the project requirements. From the information gathered a solid knowledge base was developed from

which to further refine the system requirements, as well as aid in development later in the project. Having a good understanding of the hardware before beginning the design helped to understand what was possible from the system. Knowing the physical limits of the hardware available helped to ensure that the system was feasible on the hardware platform being used.

3.3 Expert Discussion

While the literature review provides a good understanding of the overall field, alone it doesn't provide enough information to properly specify the system. More guidance for the development of the requirements is found in step #3, where the system was discussed with an expert in the field. This gives a better understanding of what is most relevant to the users of the system. For this reason most of the user requirements were based heavily on expert input. The expert discussion was again not a single event, but rather a continuous process, with the bulk of the discussion taking place at this point in development. At each point in the development of the requirements this was repeated, to make sure the system being designed matched the system that was desired.

3.4 Requirement Review

The fourth step in the methodology model is to review the initial user requirements. With a better understanding of the system, new insights were obtained on what the user requirements should be, and thus could be re-worked to better suit the problem. This process was repeated until a final set of requirements was developed. The expert discussion was very closely linked to the requirement review process.

In this step parameters specified by the initial requirements were also investigated. Parameters that did not have forms that correspond well to hardware development, such as the Pulse Repetition Frequency (PRF), were translated into a more suitable parameters (in the

case of the PRF, instead a Pulse Repetition Interval which is more easily mapped to a delay function is used.) Units for parameters were also evaluated into units used in hardware, such as bits and bytes (in the case of the PRF, it was ultimately represented in terms of a number of bits required to store the goal for the counter.)

3.5 Rhino Platform Setup

Step #5 was to setup the Rhino platform, getting it to a point where development of applications could be run on the platform. As Rhino was a new platform this process was documented in order to help future users to easily configure the Rhino.

This process was actually started while Rhino was still under design, using an AM3517 development board to determine specifics about the board such as booting. This initial testing aimed to serve as a "proof of concept" for the ARM processor that was used in the Rhino platform.

Setting up the Rhino involved both the physical tasks such as wiring up power supplies and setting jumpers, as well as software tasks, such as configuring the boot procedure and developing software to program the FPGA.

It is important to note that while the Rhino platform has many interfaces through which access can be gained, it does not have any "human interface devices". This means that a computer is needed to enable the user to access the Rhino, and as such the first part of setting up the Rhino platform was to connect it to the Control PC. For this project a Pc running Ubuntu was used to control and interface with the Rhino.

3.6 ARM to FPGA Bus

Using the ARM to power up and program the FPGA, while being extremely useful, comprises only a small section of the ARM processor's desired functionality. Step #6 requires the design and development of gateware and software to control the communications bus between the ARM processor and the FPGA. Through this bus the ARM processor provides control over applications being run on the FPGA.

The Rhino platform provides a bus connecting the General Purpose Memory Controller (GPMC) on the ARM to pins on the FPGA. At this stage in development a software driver is created to allow users to write data onto the bus, without them requiring any knowledge of the structure of the bus.

Drivers running on the ARM allowing information to be written to the pins on the FPGA provide only one half of the communication bus desired. The other half of the bus is provided by gateware that runs on the FPGA to interpret this information on the pins and respond to it appropriately.

3.7 Gigabit Ethernet

Step #7 was to design and implement a gateware block that would allow the Rhino to output data over Gigabit Ethernet. This block was designed to be reusable in other Rhino projects to allow for easier development of applications on the Rhino. In this stage the Gigabit Ethernet block was developed as a stand-alone system, and tested as such. The Gigabit Ethernet was to be used as a method of testing the components of the radar block, and thus had to be completely functional before the radar block could be developed.

3.8 Radar Block Development

Development for the radar block was planned to be done in small components, with each component fulfilling one of the system functions; step #8 incorporates the design and development of each of the components making up the radar block.

3.9 Radar Block testing

In step #9, each individual component in the radar block was tested to see if it meets the performance requirements for the system. If a component does not meet the required performance benchmarks it is re-worked, until either it met the required level, or reached a point at which it could no longer be improved due to some limiting factor. The tests run in this step are described in Section 3.11

3.10 Full System Test

The penultimate step, step #10, involved testing the system as a whole. Here all the components of the radar block are connected to form the final system prototype. This prototype is then tested to determine how the system functions as a whole. The tests run in this step are described in Section 3.11.10

3.11 System Acceptance Test Procedure

In the steps explained above, various tests are carried out in order to ensure the system functions in an acceptable manner. The details of the tests performed out outlined in the following discussion, the tests are organised according to the system functions which they test. Tests that either evaluate multiple functions or cannot easily be related to a single function are grouped together at the end of this section.

3.11.1 Performance Tests for F1. Programming the FPGA

Programming Time Test

The goal of this test is to determine the time taken to correctly program the FPGA. This test determines both the ability to program the FPGA and the time taken to do so.

To test that the FPGA has been programmed correctly gateware that provides an output visible to the user is used. A simple gateware application that cycles the LEDs connected to the FPGA was programmed onto the FPGA, allowing visual confirmation that the FPGA has been programmed correctly.

The time taken to program the full FPGA image is done by comparing the timestamps at the start of the messages signifying the start of programming with those signifying the end. For this test it is required that the FPGA be programmed in under 5 minutes.

3.11.2 Performance Tests for F2. FPGA-ARM Data Bus

Transfer Integrity Test

To determine the correct functioning of communications between the ARM and the FPGA a simple application was used to write a series of data to the FPGA, then read back the data and compare it to the original data. If the data read from the system matches the input data exactly, the data transfer is deemed successful.

Reading the data back was done both through the Gigabit Ethernet and through the data bus. This was important because the memory on the FPGA was written 16 bits at a time on the bus side, and read 32 bits at a time on the radar block side.

For the purposes of the test it was assumed that the data which was to be transferred to the FPGA was correct. This assumption was easy to make as the file system for the Linux running

on the ARM processor was run through a Networked File System over TCP/IP, which should provide error free transmission[37].

Transfer Rate Test

This test was designed to determine that average time taken to upload configuration settings for the radar block over the FPGA-ARM bus. The test uses an application to repeatedly write configuration files to the FPGA, and record the length of time taken to write data. The application is given a data file, and the number of times it must write this file to the bus. This determines the speed of transfer that can be expected when using the bus for the radar application (or similar application).

The test was run for 100 byte, 1000 byte and 30000 byte files. The reason the files were written multiple times is to give a good indication of the average time for transfer.

The bus is required to be able to re-configure the radar block in less than one second, with the largest configuration file being 32Kb; an average transfer speed of at least 32Kb/s is required.

3.11.3 **Performance Tests for F3. Radar Block Parameters**

Register Value Test

To confirm that the correct values are being stored in each of the registers, they are printed out to the boards LEDs and the value shown on the LEDs is then compared with the desired value.

3.11.4 **Performance Tests for F4. Pulse Repetition**

Pulse Repetition Interval Range

The Pulse Repetition range was tested through calculation. The clock speed combined with the register size used to store the PRI, allow calculations to be performed to determine the range of intervals that can be used. Dividing the maximum PRI counter value by the Clock Frequency, gives the maximum delay created by the counter.

$$Maximum\ Interval = \frac{Max\ PRI\ Register\ Value}{Clock\ Frequency}$$

The system requires that the maximum PRI be at least one second long.

Pulse Repetition Interval Resolution

The resolution for the PRI was tested through calculation. The resolution of the PRI is equal to the inverse of the Clock Frequency.

$$Resolution = \frac{1}{Clock\ Frequency}$$

Performance requirements state that the resolution needs to be at least 1/1000 of the full range; this translates to a PRI resolution of 1µs.

Pulse Repetition Interval Accuracy

The accuracy of the PRI can again be determined through calculation. The accuracy of the pulse repetition function is going to be based on the accuracy of the clock that is driving it, which is available in the clocks datasheet. The jitter of the clock needs to be under the required jitter for the PRI, which is 1%.

Pulse Repetition Timing

This test aimed to show the system was operating according to the desired timing. An oscilloscope was used to monitor signals which were routed to output pins for the purposes of this test. Two signals were routed to IO pins, the first showed when data was being

transmitted, and the second showed when data was being received. From the output waveforms of these two signals it is possible to determine if the system has the desired PRI.

3.11.5 **Performance Tests for F5 Pulse Transmission**

Pulse Data Transmission test

In a complete radar system, it is possible to either check that the data coming out of the Digital to Analogue (DAC) converter is correct, or that the waveform radiating from the antenna is correct. Unfortunately for the purposes of this project there is no DAC, and the transmitted output data was being sent to an 'FMC mimic' block that would store the data for a few cycles and then return it to the receive block. For this reason, the blocks functionality was confirmed be examining the data returned by the receive block.

Pulse Transmission Speed.

The speed of transmission is tested through calculation, by multiplying the clock speed of the data transfer by the width of the bus.

$$Bus\ Speed = Bus\ Width\ \times Clock\ Speed$$

The performance requirements suggest that the optimal speed for output transmission should be around 25 GB/s, but also mention that this goal is unlikely to be achieved due to hardware limitations of the DACs. An acceptable speed would be any speed above 400Mb/s.

3.11.6 **Performance Tests for F6 Pulse Reception**

Pulse Data Reception Test

The pulse reception suffers from the same issue as pulse transmission, in that ADC's are not implemented in this project. In order to confirm the proper operation of the pulse reception function (and at the same time the pulse transmission function as the received data was exactly the same as the input data), data was transferred from the output using gateware designed to write the contents of a memory block to the network. The received data was compared to the

input data, if the pulse reception and transmission functions were working correctly, these two data sets should be identical.

Pulse Reception Speed.

The speed of data reception is tested through calculation, by multiplying the clock speed of the data transfer by the width of the bus.

$$Bus\ Speed = Bus\ Width \times Clock\ Speed$$

The performance requirements suggest that the optimal speed for input reception should be around 25 GB/s, but also mention that this goal is unlikely to be achieved due to hardware limitations of the ADC. An acceptable speed would be any speed above 400Mb/s.

3.11.7 Performance Tests for F7. Receive Delay

Receive Delay Interval Range

The Receive Delay range was tested through calculation. The clock speed combined with the register size used to store the delay, allow calculations to be performed to determine the range of intervals that can be used. Dividing the maximum delay counter value by the Clock Frequency, gives the maximum delay created by the counter.

$$Maximum\ Interval = \frac{Max\ Delay\ Register\ Value}{CLock\ Frequency}$$

The system requires that the maximum PRI be at least one second long.

Receive Delay Resolution

The resolution for the delay was tested through calculation. The resolution of the delay is equal to the inverse of the Clock Frequency.

$$Resolution = \frac{1}{Clock\ Frequency}$$

Performance requirements state that the delay resolution needs to be at least 5ns.

Receive Delay Accuracy

The accuracy of the delay can again be determined through calculation. The accuracy of the receive delay function is going to be based on the accuracy of the clock that is driving it, which is available in the clocks datasheet. The jitter of the clock needs to be under the required jitter for the delay, which is 1%.

Receive Delay Timing

This test aimed to show the system was operating according to the desired timing. An oscilloscope was used to monitor signals which were routed to output pins for the purposes of this test. Two signals were routed to IO pins, the first showed when data was being transmitted, and the second showed when data was being received. From the output waveforms of these two signals it is possible to determine if the system has the desired receive delay.

3.11.8 **Performance Tests for F8. Gigabit Ethernet**

Data Transfer Test

Testing of the Gigabit Ethernet was done with a piece of gateware that outputs a chunk of data over the network. A pc running WireShark (Figure 3-2) was then used to confirm that the correct data had been transferred.

bus between the ARM processor and the FPGA. Step #7 was to design and implement a gateware block that would allow the rhino to output data over Gigabit Ethernet. Step #8 incorporates the design and development of each of the components making up the radar block. In step #9, each individual component in the radar block was tested to see if it meets the performance requirements for the system. The penultimate step, step #10, involved testing the system as a whole. Step #11 is the final step in the project methodology, and it involved the drawing up of conclusions and recommending future work based on the outcomes and results of the project. These steps are discussed in more detail in the following sections.

3.1 Initial User Requirements

The first step, step #1 in Figure 3-1, in the system development process was to establish the initial requirements for the project. This process involved meeting with the stakeholders of the project in order to discuss desired outcomes. From these discussions initial requirements were developed, with the need for a pulse radar block for the Rhino being identified, and consequently the focus for this project was selected accordingly.

3.2 Background and Literature Review

Step #2 of the methodology was to conduct background research and a literature review. This process created aided in familiarization with the topics of pulse radar and FPGA-based design. While the literature review is shown in Figure 3-1 as a single process, it was actually a continuous process where literature was constantly being consulted throughout the project. The reason it is shown as a single block, is because this is the point at which the bulk of the work was done.

The initial literature review created a good understanding of the current technologies available in the relevant fields and provided a good starting point for further development of the project requirements. From the information gathered a solid knowledge base was developed from

which to further refine the system requirements, as well as aid in development later in the project. Having a good understanding of the hardware before beginning the design helped to understand what was possible from the system. Knowing the physical limits of the hardware available helped to ensure that the system was feasible on the hardware platform being used.

3.3 Expert Discussion

While the literature review provides a good understanding of the overall field, alone it doesn't provide enough information to properly specify the system. More guidance for the development of the requirements is found in step #3, where the system was discussed with an expert in the field. This gives a better understanding of what is most relevant to the users of the system. For this reason most of the user requirements were based heavily on expert input. The expert discussion was again not a single event, but rather a continuous process, with the bulk of the discussion taking place at this point in development. At each point in the development of the requirements this was repeated, to make sure the system being designed matched the system that was desired.

3.4 Requirement Review

The fourth step in the methodology model is to review the initial user requirements. With a better understanding of the system, new insights were obtained on what the user requirements should be, and thus could be re-worked to better suit the problem. This process was repeated until a final set of requirements was developed. The expert discussion was very closely linked to the requirement review process.

In this step parameters specified by the initial requirements were also investigated. Parameters that did not have forms that correspond well to hardware development, such as the Pulse Repetition Frequency (PRF), were translated into a more suitable parameters (in the

case of the PRF, instead a Pulse Repetition Interval which is more easily mapped to a delay function is used.) Units for parameters were also evaluated into units used in hardware, such as bits and bytes (in the case of the PRF, it was ultimately represented in terms of a number of bits required to store the goal for the counter.)

3.5 Rhino Platform Setup

Step #5 was to setup the Rhino platform, getting it to a point where development of applications could be run on the platform. As Rhino was a new platform this process was documented in order to help future users to easily configure the Rhino.

This process was actually started while Rhino was still under design, using an AM3517 development board to determine specifics about the board such as booting. This initial testing aimed to serve as a "proof of concept" for the ARM processor that was used in the Rhino platform.

Setting up the Rhino involved both the physical tasks such as wiring up power supplies and setting jumpers, as well as software tasks, such as configuring the boot procedure and developing software to program the FPGA.

It is important to note that while the Rhino platform has many interfaces through which access can be gained, it does not have any "human interface devices". This means that a computer is needed to enable the user to access the Rhino, and as such the first part of setting up the Rhino platform was to connect it to the Control PC. For this project a Pc running Ubuntu was used to control and interface with the Rhino.

3.6 ARM to FPGA Bus

Using the ARM to power up and program the FPGA, while being extremely useful, comprises only a small section of the ARM processor's desired functionality. Step #6 requires the design and development of gateware and software to control the communications bus between the ARM processor and the FPGA. Through this bus the ARM processor provides control over applications being run on the FPGA.

The Rhino platform provides a bus connecting the General Purpose Memory Controller (GPMC) on the ARM to pins on the FPGA. At this stage in development a software driver is created to allow users to write data onto the bus, without them requiring any knowledge of the structure of the bus.

Drivers running on the ARM allowing information to be written to the pins on the FPGA provide only one half of the communication bus desired. The other half of the bus is provided by gateware that runs on the FPGA to interpret this information on the pins and respond to it appropriately.

3.7 Gigabit Ethernet

Step #7 was to design and implement a gateware block that would allow the Rhino to output data over Gigabit Ethernet. This block was designed to be reusable in other Rhino projects to allow for easier development of applications on the Rhino. In this stage the Gigabit Ethernet block was developed as a stand-alone system, and tested as such. The Gigabit Ethernet was to be used as a method of testing the components of the radar block, and thus had to be completely functional before the radar block could be developed.

3.8 Radar Block Development

Development for the radar block was planned to be done in small components, with each component fulfilling one of the system functions; step #8 incorporates the design and development of each of the components making up the radar block.

3.9 Radar Block testing

In step #9, each individual component in the radar block was tested to see if it meets the performance requirements for the system. If a component does not meet the required performance benchmarks it is re-worked, until either it met the required level, or reached a point at which it could no longer be improved due to some limiting factor. The tests run in this step are described in Section 3.11

3.10 Full System Test

The penultimate step, step #10, involved testing the system as a whole. Here all the components of the radar block are connected to form the final system prototype. This prototype is then tested to determine how the system functions as a whole. The tests run in this step are described in Section 3.11.10

3.11 System Acceptance Test Procedure

In the steps explained above, various tests are carried out in order to ensure the system functions in an acceptable manner. The details of the tests performed out outlined in the following discussion, the tests are organised according to the system functions which they test. Tests that either evaluate multiple functions or cannot easily be related to a single function are grouped together at the end of this section.

3.11.1 Performance Tests for F1. Programming the FPGA

Programming Time Test

The goal of this test is to determine the time taken to correctly program the FPGA. This test determines both the ability to program the FPGA and the time taken to do so.

To test that the FPGA has been programmed correctly gateware that provides an output visible to the user is used. A simple gateware application that cycles the LEDs connected to the FPGA was programmed onto the FPGA, allowing visual confirmation that the FPGA has been programmed correctly.

The time taken to program the full FPGA image is done by comparing the timestamps at the start of the messages signifying the start of programming with those signifying the end. For this test it is required that the FPGA be programmed in under 5 minutes.

3.11.2 Performance Tests for F2. FPGA-ARM Data Bus

Transfer Integrity Test

To determine the correct functioning of communications between the ARM and the FPGA a simple application was used to write a series of data to the FPGA, then read back the data and compare it to the original data. If the data read from the system matches the input data exactly, the data transfer is deemed successful.

Reading the data back was done both through the Gigabit Ethernet and through the data bus. This was important because the memory on the FPGA was written 16 bits at a time on the bus side, and read 32 bits at a time on the radar block side.

For the purposes of the test it was assumed that the data which was to be transferred to the FPGA was correct. This assumption was easy to make as the file system for the Linux running

on the ARM processor was run through a Networked File System over TCP/IP, which should provide error free transmission[37].

Transfer Rate Test

This test was designed to determine that average time taken to upload configuration settings for the radar block over the FPGA-ARM bus. The test uses an application to repeatedly write configuration files to the FPGA, and record the length of time taken to write data. The application is given a data file, and the number of times it must write this file to the bus. This determines the speed of transfer that can be expected when using the bus for the radar application (or similar application).

The test was run for 100 byte, 1000 byte and 30000 byte files. The reason the files were written multiple times is to give a good indication of the average time for transfer.

The bus is required to be able to re-configure the radar block in less than one second, with the largest configuration file being 32Kb; an average transfer speed of at least 32Kb/s is required.

3.11.3 **Performance Tests for F3. Radar Block Parameters**

Register Value Test

To confirm that the correct values are being stored in each of the registers, they are printed out to the boards LEDs and the value shown on the LEDs is then compared with the desired value.

3.11.4 **Performance Tests for F4. Pulse Repetition**

Pulse Repetition Interval Range

The Pulse Repetition range was tested through calculation. The clock speed combined with the register size used to store the PRI, allow calculations to be performed to determine the range of intervals that can be used. Dividing the maximum PRI counter value by the Clock Frequency, gives the maximum delay created by the counter.

$$Maximum\ Interval = \frac{Max\ PRI\ Register\ Value}{Clock\ Frequency}$$

The system requires that the maximum PRI be at least one second long.

Pulse Repetition Interval Resolution

The resolution for the PRI was tested through calculation. The resolution of the PRI is equal to the inverse of the Clock Frequency.

$$Resolution = \frac{1}{Clock\ Frequency}$$

Performance requirements state that the resolution needs to be at least 1/1000 of the full range; this translates to a PRI resolution of 1µs.

Pulse Repetition Interval Accuracy

The accuracy of the PRI can again be determined through calculation. The accuracy of the pulse repetition function is going to be based on the accuracy of the clock that is driving it, which is available in the clocks datasheet. The jitter of the clock needs to be under the required jitter for the PRI, which is 1%.

Pulse Repetition Timing

This test aimed to show the system was operating according to the desired timing. An oscilloscope was used to monitor signals which were routed to output pins for the purposes of this test. Two signals were routed to IO pins, the first showed when data was being

transmitted, and the second showed when data was being received. From the output waveforms of these two signals it is possible to determine if the system has the desired PRI.

3.11.5 **Performance Tests for F5 Pulse Transmission**

Pulse Data Transmission test

In a complete radar system, it is possible to either check that the data coming out of the Digital to Analogue (DAC) converter is correct, or that the waveform radiating from the antenna is correct. Unfortunately for the purposes of this project there is no DAC, and the transmitted output data was being sent to an 'FMC mimic' block that would store the data for a few cycles and then return it to the receive block. For this reason, the blocks functionality was confirmed be examining the data returned by the receive block.

Pulse Transmission Speed.

The speed of transmission is tested through calculation, by multiplying the clock speed of the data transfer by the width of the bus.

$$Bus\ Speed = Bus\ Width\ \times Clock\ Speed$$

The performance requirements suggest that the optimal speed for output transmission should be around 25 GB/s, but also mention that this goal is unlikely to be achieved due to hardware limitations of the DACs. An acceptable speed would be any speed above 400Mb/s.

3.11.6 **Performance Tests for F6 Pulse Reception**

Pulse Data Reception Test

The pulse reception suffers from the same issue as pulse transmission, in that ADC's are not implemented in this project. In order to confirm the proper operation of the pulse reception function (and at the same time the pulse transmission function as the received data was exactly the same as the input data), data was transferred from the output using gateware designed to write the contents of a memory block to the network. The received data was compared to the

input data, if the pulse reception and transmission functions were working correctly, these two data sets should be identical.

Pulse Reception Speed.

The speed of data reception is tested through calculation, by multiplying the clock speed of the data transfer by the width of the bus.

$$Bus\ Speed = Bus\ Width \times Clock\ Speed$$

The performance requirements suggest that the optimal speed for input reception should be around 25 GB/s, but also mention that this goal is unlikely to be achieved due to hardware limitations of the ADC. An acceptable speed would be any speed above 400Mb/s.

3.11.7 Performance Tests for F7. Receive Delay

Receive Delay Interval Range

The Receive Delay range was tested through calculation. The clock speed combined with the register size used to store the delay, allow calculations to be performed to determine the range of intervals that can be used. Dividing the maximum delay counter value by the Clock Frequency, gives the maximum delay created by the counter.

$$Maximum\ Interval = \frac{Max\ Delay\ Register\ Value}{CLock\ Frequency}$$

The system requires that the maximum PRI be at least one second long.

Receive Delay Resolution

The resolution for the delay was tested through calculation. The resolution of the delay is equal to the inverse of the Clock Frequency.

$$Resolution = \frac{1}{Clock\ Frequency}$$

Performance requirements state that the delay resolution needs to be at least 5ns.

Receive Delay Accuracy

The accuracy of the delay can again be determined through calculation. The accuracy of the receive delay function is going to be based on the accuracy of the clock that is driving it, which is available in the clocks datasheet. The jitter of the clock needs to be under the required jitter for the delay, which is 1%.

Receive Delay Timing

This test aimed to show the system was operating according to the desired timing. An oscilloscope was used to monitor signals which were routed to output pins for the purposes of this test. Two signals were routed to IO pins, the first showed when data was being transmitted, and the second showed when data was being received. From the output waveforms of these two signals it is possible to determine if the system has the desired receive delay.

3.11.8 **Performance Tests for F8. Gigabit Ethernet**

Data Transfer Test

Testing of the Gigabit Ethernet was done with a piece of gateware that outputs a chunk of data over the network. A pc running WireShark (Figure 3-2) was then used to confirm that the correct data had been transferred.

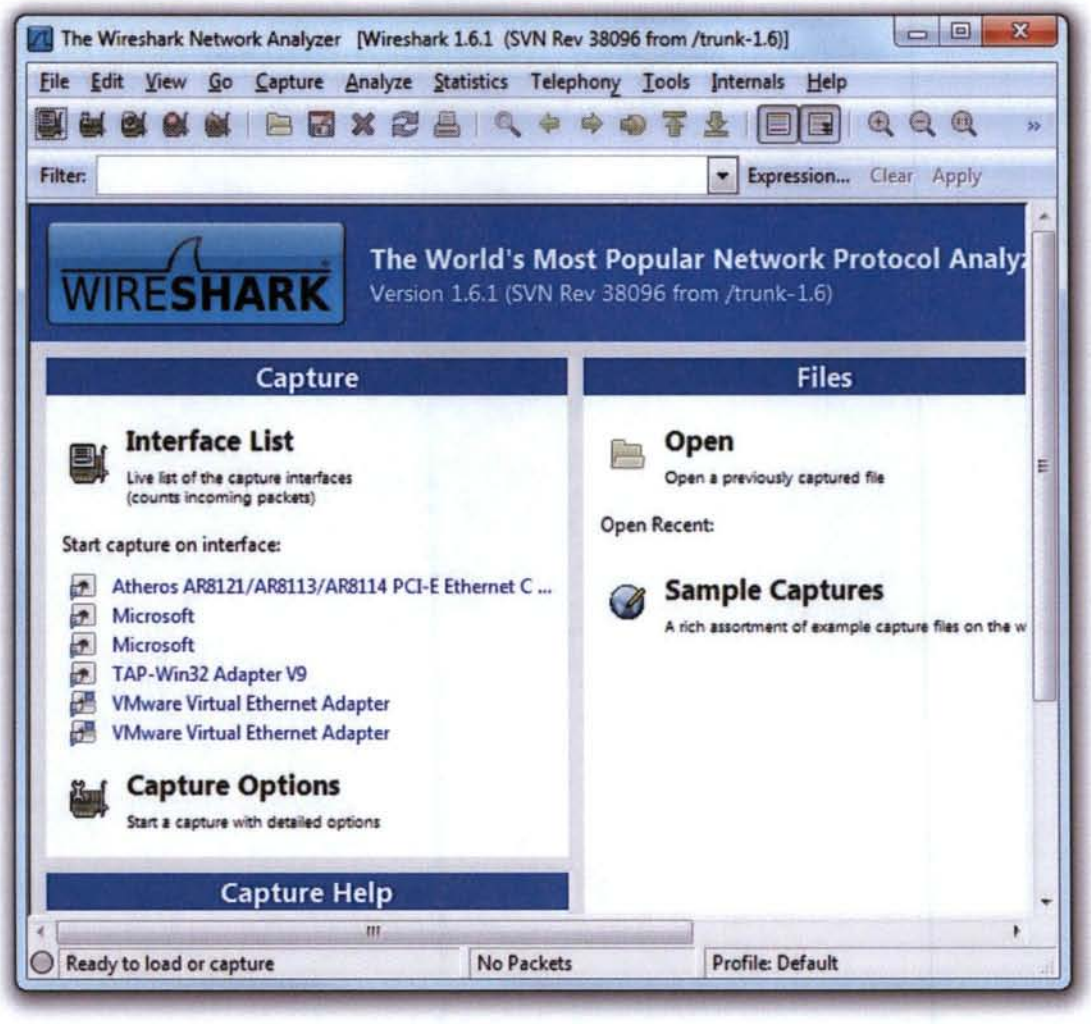

Figure 3-2: WireShark will be used to capture incoming data on the Gigabit Ethernet

Transfer Speed Test

The speed of the Gigabit Ethernet transfer is determined through examination of the packets in Wireshark. Every packet received has a timestamp with a microsecond resolution, recording the exact time it arrived. By examining the time stamps of collective packets the transfer speed of the Gigabit Ethernet can be determined. As mentioned in the performance requirements, the required data flow suggests the system required 25 GB/s data rate, which is obviously not possible for Gigabit Ethernet. Instead a transfer rate of 400Mb/s was the goal value for this test.

3.11.9 **Performance Tests for F9. Network Storage**

Data Storage Test

The network storage was tested by comparing the input data file to the output data file when the system is run using a chirp waveform. If no data loss exists between input and output, the speed of the network storage is acceptable.

3.11.10 **Full System Test**

Full System Verification Test (F3, F8)

The purpose of this test was to confirm the systems operation as a complete system. The system was tested with operating parameters at the limits of the specifications, and the output data was examined.

It was suspected that the speed of the Gigabit Ethernet block would act as a bottleneck in the system data flow, as it would not be sufficient to run the system with both the maximum PRF and maximum pulse size. For this reason the system was tested with the maximum pulse size, and various PRF values. The PRF values were adjusted in order to determine the maximum PRF possible for use with the full data size.

For these tests an 8096 sample chirp was used as our input waveform as seen in Figure 3-3. The pulse repetition interval was then varied between 0s and 1s. The delay between transmitting and receiving data was fixed due to the nature of the block mimicking the action of the ADC's and DAC's.

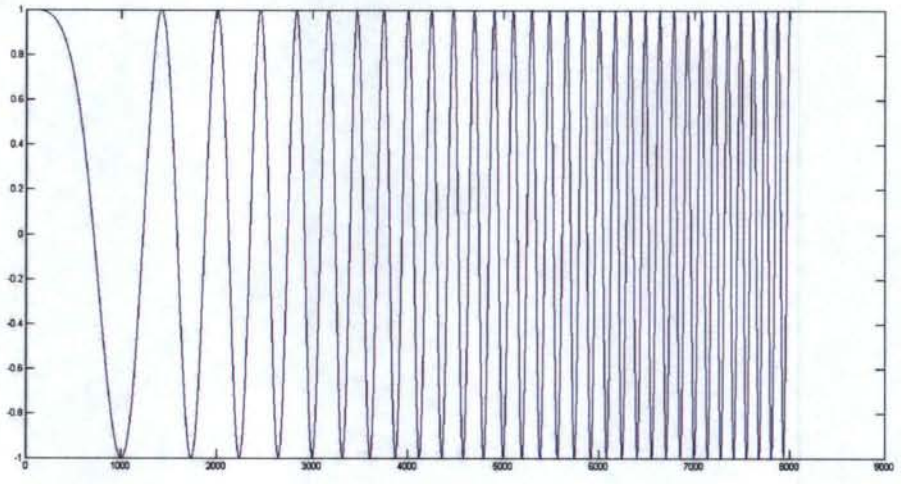

Figure 3-3: Chirp waveform used in system verification test.

It was expected that at some point the PRI would become too short, at this point the next set of data was to be transmitted before the previous data had finished transmitting. This was noticeable as the output data contained a header with information about which pulse it was generated by, and if the start of a new pulse would appear in a packet labelled for the previous pulse.

The point at which the PRI became too short for the output buffer to be flushed before the next series of data arrived showed the minimum PRI that could be managed when using the maximum pulse length.

3.12 **Conclusions**

Step #11 is the final step in the project methodology, and it involved the drawing up of conclusions and recommending future work based on the outcomes and results of the project.

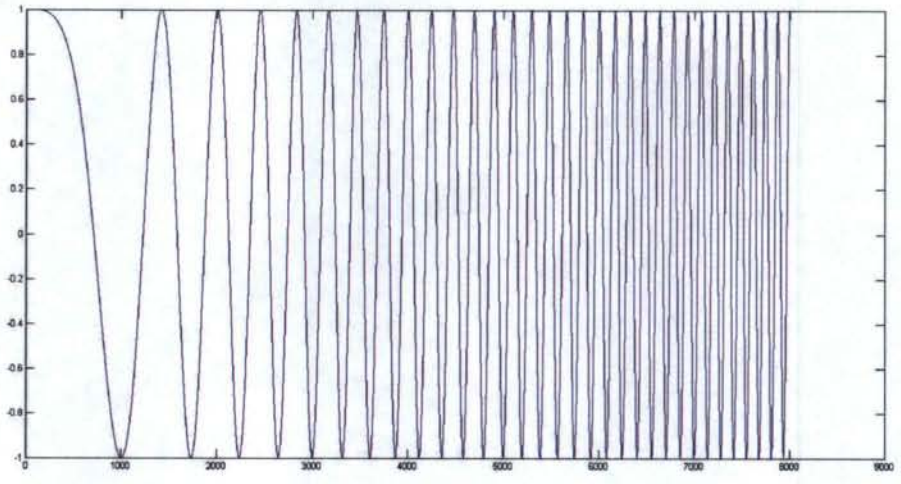

Figure 3-3: Chirp waveform used in system verification test.

It was expected that at some point the PRI would become too short, at this point the next set of data was to be transmitted before the previous data had finished transmitting. This was noticeable as the output data contained a header with information about which pulse it was generated by, and if the start of a new pulse would appear in a packet labelled for the previous pulse.

The point at which the PRI became too short for the output buffer to be flushed before the next series of data arrived showed the minimum PRI that could be managed when using the maximum pulse length.

3.12 **Conclusions**

Step #11 is the final step in the project methodology, and it involved the drawing up of conclusions and recommending future work based on the outcomes and results of the project.

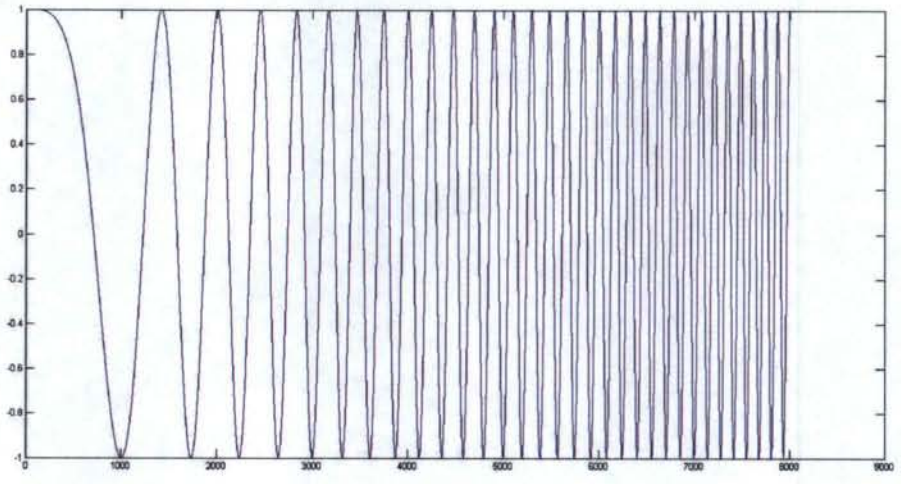

Figure 3-3: Chirp waveform used in system verification test.

It was expected that at some point the PRI would become too short, at this point the next set of data was to be transmitted before the previous data had finished transmitting. This was noticeable as the output data contained a header with information about which pulse it was generated by, and if the start of a new pulse would appear in a packet labelled for the previous pulse.

The point at which the PRI became too short for the output buffer to be flushed before the next series of data arrived showed the minimum PRI that could be managed when using the maximum pulse length.

3.12 **Conclusions**

Step #11 is the final step in the project methodology, and it involved the drawing up of conclusions and recommending future work based on the outcomes and results of the project.

4 System Design

This chapter gives an overview of the design process for the development of the Rhino System on which the radar block will be implemented. At the outset of this project, the Rhino platform was still in the early development stage and had no supporting software at all. Much work was required to get Rhino to a point where it could start running applications.

This chapter starts with a review of the user requirements that were presented in section 1.5. These requirements are reviewed and a final set of system requirements is presented in section 4.1. The parameters for the pulse radar block are discussed and details of the ranges of the radar block parameters are determined.

A high level overview of the system design is presented in Section 4.2, while section 4.3 covers the ARM processor, the setup and configuration of the ARM processor together with design of the programming interface that allows users to program the Spartan 6 FPGA from the ARM processor is discussed.

This chapter also covers the design of the supporting blocks for the radar system, such as the Gigabit Ethernet block.

4.1 Requirements Review

The requirements given in section 1.5 are not enough to build a system from. While they give a good starting point to work from, not enough detail is given. In order to create a set of specifications to build the system on these requirements need to be reviewed, and reworked.

4.1.1 Pulse Radar Delays (requirements U4,U5,U8,U9)

Implementing the pulse radar block requires two delays. The first is a delay to create the correct PRF, which will be done by setting a delay between the pulses (Pulse Repetition

Interval). The second comes from the fact that there is a delay between the transmitted signal and the return of the echo. A representation of this timing is shown Figure 4-1 Fast/Slow time diagram of Pulse Radar where there are two time axes, one showing pulse and echo with time moving slowly, and the other showing how this is repeated with time sped up.

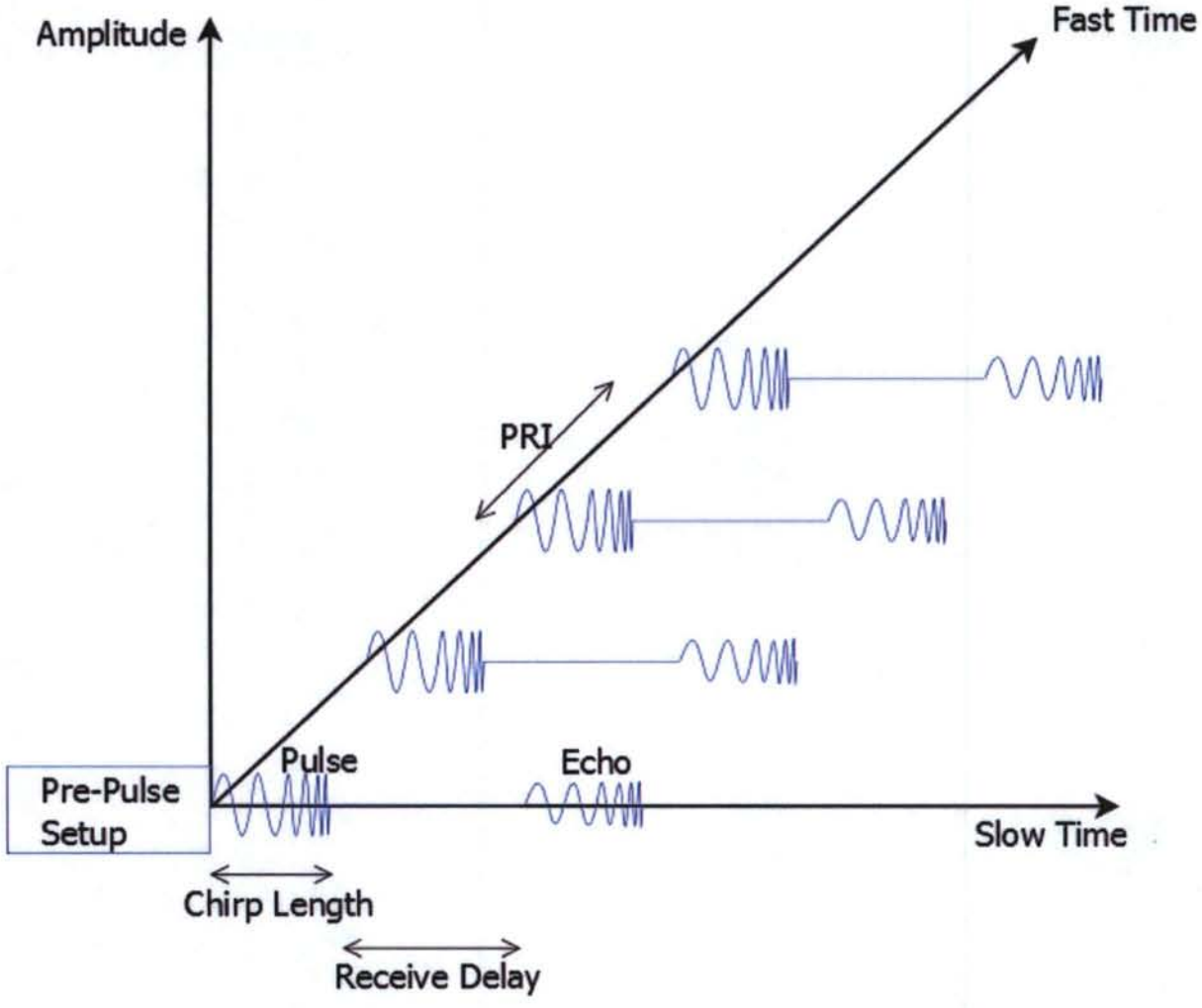

Figure 4-1 Fast/Slow time diagram of Pulse Radar

4.1.2 Parameter Control

The radar block has several parameters that need to be controlled by the user. The ARM processor needs to be used to control the FPGA, allowing the user to vary the required

parameters. In order to be able to use the ARM processor to control these parameters the user needs to be able to connect to the ARM processor.

In the review of the networking hardware of the ARM (section 2.4 it was discovered that there is a 100mb/s Ethernet connection to the ARM, as well as a serial connection over USB. One of these two interfaces needs to be used to control the ARM.

There also needs to be a method of communication between the ARM and the FPGA simple drivers installed into the Linux OS on the ARM processor to control the function of the radar block.

The following parameters in the radar block will need to be controlled:

- Pulse Repetition Interval (PRI) - Defines the time between pulses.
- Pulse Waveform - Complex, sampled, waveform to be transmitted.
- Pulse Width - The number of samples in the waveform.
- Delay - Time delay between transmitting and receiving the waveform.

In addition to these parameters, the user must also be able to start and stop the function of the radar block.

4.1.3 Parameter Ranges

Each of these parameters needs to have registers on the FPGA associated with it. These registers are used to store the values for use in the radar block. In order to determine the sizes of these registers, the range of each parameter is required.

To get typical real values that would be required from this system, an expert in radar systems was consulted. The resulting values that came from this consultation are shown and discussed below; while a summary of the register sizes are seen in Table 4-1.

Table 4-1: Minimum Register Widths

Requirement	Register	Minimum Size (bits)
U4	PRI Register	27
U6	Pulse Length Register	14
U8	Delay Length Register	27

U4. PRI Range:

The PRI needs to be able to vary between 0 and 1 second. With a the 100MHz clock running on the FPGA this translates to counting to 10^8, which requires a register of 27 bits wide.

U6. Pulse Length Range:

The pulse required to be transmitted needs to be stored in memory on the board, and a register needs to be stored to record the length of the pulse. The possible range of pulse lengths varies between the DAC clock length and 8096 complex, 16 bit samples. The lower limit has little relevance because the memory size is fixed at the upper limit. The complex samples will be represented by a real and imaginary part joined to form one 32bit value. This means

that 32 384 bytes of memory need to be available to store the pulse and the pulse length register needs to be 14 bits long.

U7. Received Pulse Size:

Returning pulses need to be stored in memory before being transmitted out over the gigabit Ethernet. The system needs to be able to store up to 8096 complex samples received, which again translates to 32 384 bytes of memory required to store incoming pulses.

U8. Delay Length:

Between the pulse transmission and receiving the echo there is a delay. The delay length needs to be variable up to the length of the PRI. The delay requires a resolution of 5ns, which requires a 200MHz clock; this means that it requires 27 bits.

U10,U11. Output Data Rate:

Data needs to be sent out of the Gigabit Ethernet at a rate which is fast enough to empty the receive buffer before the next pulse is sent. Because there is little control over the maximum speed available over the Gigabit Ethernet, and the 10 Gigabit Ethernet is outside the scope of this project, this will put a restriction on the minimum PRI for any given pulse length.

4.1.4 **[U2] FPGA-ARM Bus**

In order to control the radar block parameters a driver needs to be implemented in the Linux kernel running on the ARM processor that will facilitate communications to the FPGA. On the Rhino platform, as was seen in section 2.4.1, the ARM is connected to the FPGA through its General Purpose Memory Controller (GPMC). The Linux driver must map the virtual addresses

available in the GPMC to locations to which the user can write, so that data can be written to the FPGA.

For the communication to work a gateware block that can be implemented on the FPGA that will handle the communications incoming from the ARM is also required. This block needs to make the incoming data from the ARM available so that the user can map the various addresses being addressed by the arm to the appropriate registers or memory locations.

The bus needs to give users the ability to easily configure their own communication protocols without having to go through the complexities of setting up the GPMC on the ARM. This allows for the bus to be used in other applications.

4.1.5 **Pulse Radar Block (requirements U4,U5,U6,U7)**

A block diagram on the pulse radar implementation can be seen in Figure 4-2, where the flow of data between blocks is shown. The FPGA needs to get an input waveform (described in section 4.1.3 above) from the ARM processor. The input waveform must be stored in memory on the FPGA.

The transmitter must send out the waveform at the required PRF (which is determined by the PRI as described in section 4.1.3 above). To do this a PRI counter must be implemented to trigger the transmitter after the required value in the counter is reached. The PRI counter needs to be controlled by the user through start/stop registers. When triggered the transmitter should send the data out to the DAC (for the purposes of this work an actual DAC will not be used, signals a tested by loop-back between transmitted and received signals).

The trigger generated by PRI counter must also be sent to a delay block. When triggered the counter in the delay block needs to count to the value specified by the Delay register. When the counter reaches its goal it must trigger the receiver.

When the trigger is received by the receiver, the data coming from the ADC must be stored to memory (again in this case an Actual ADC will not be used, data will be looped back from the transmit output).

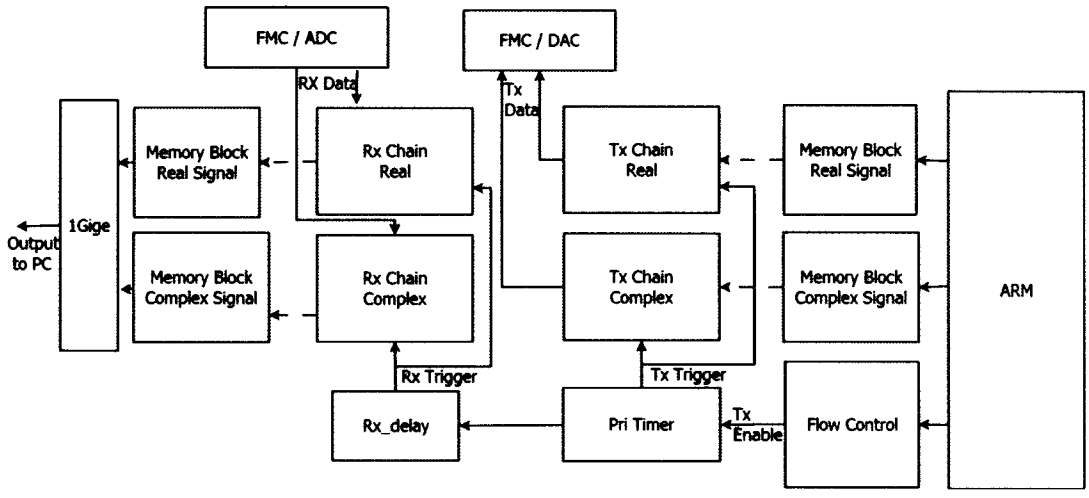

Figure 4-2 A Basic Example Block Diagram of a Radar Block Implementation

4.1.6 [F8] Gigabit Ethernet

The output data from the radar block needs to be stored in a Network-Attached Storage device (for this project the control pc will be used as the NAS). In order to get the data to the attached device, the data needs to be transmitted over the Gigabit Ethernet. The FPGA will need a control block that makes sure the data is transferred correctly to the storage device. The network needs to be able to transfer all of the received data to the NAS, thus clearing the output buffers, before the next pulse is received. The control pc needs software to save all data incoming from the Rhino.

4.2 High Level Design

At a high level the system being designed consists of 2 hardware devices, namely a control computer, and a Rhino board. The control computer is responsible for controlling the Rhino through the Ethernet and for storing the data output from the radar block, while the Rhino runs the radar block. On the Rhino there are two main processing elements. As discussed in chapter 2, the ARM processor is primarily intended for control purposes, while the Spartan 6 is intended to do the bulk of the work. Following this pattern, the Radar Block is entirely based on the FPGA, with the ARM being responsible for setting the parameters of the block.

4.2.1 System Connectivity

Figure 4-3 shows the various communication interfaces within the system and how they are connected. The ARM processor is connected to the control PC via 100Mb/s Ethernet. Two separate connections can be seen in the block diagram, one of these is a telnet connection which allows the user to access the Linux kernel on the ARM and the other is for the Networked File System (NFS) which is stored on the control pc (both of these connections share the same physical link). The use of the NFS allows the user to easily modify the file system of the Linux kernel running on the arm.Figure 4-3 shows a block diagram overview of the required system.

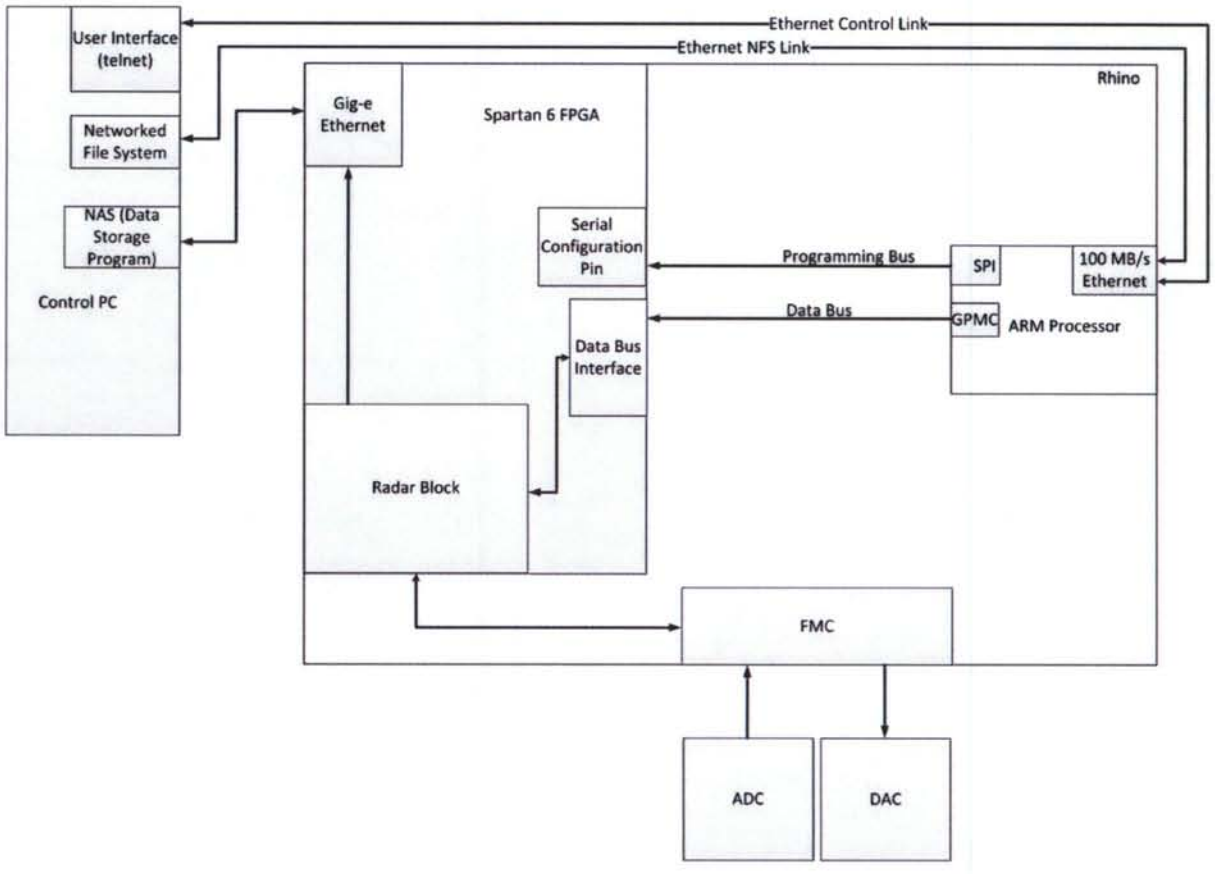

Figure 4-3: Overview of the System Connectivity

In Figure 4-3 two interfaces between the ARM processor and the FPGA are shown. The first of these interfaces is the programming bus; the second is the data bus. The programming bus is used to program the FPGA, while the Data bus is used to upload data to the FPGA and modify the radar block parameters. The FMC is used as a communications interface between the radar block and the ADCs/DACs while the gigabit Ethernet is used to transfer the output of the radar block to the control pc.

The FMC connectors are capable of connecting ADCs and DACs to the board, and while this is not implemented in this project (as mentioned in the scope and limitations section 1.7) the system is designed with this in mind.

4.2.2 System Functionality

As described in Section 1.5.3, the system has 9 core functions; in Figure 4-4: Block Diagram of System Functions functions and their interconnections are shown.

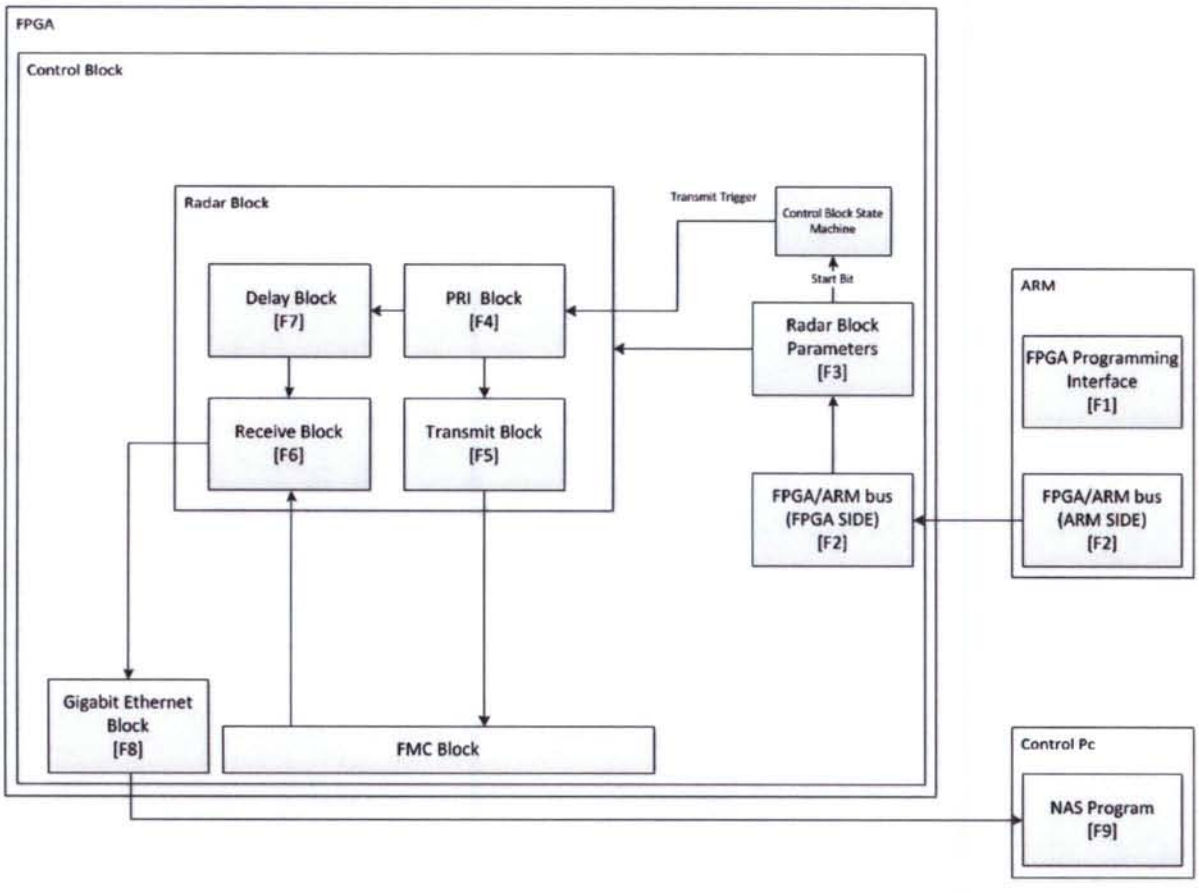

Figure 4-4: Block Diagram of System Functions

These functions provide a good high level description of the system. The FPGA programming interface (F1) is used to configure the FPGA with the gateware to be used. The user will then use the FPGA/ARM communications bus (F2) to upload the radar block parameters and waveform to the FPGA. The radar block parameters function (F3) then translates the data sent from the user and stores the parameter values in the appropriate registers.

When the start parameter is set, the PRI counter (F4) is started. The PRI counter triggers the delay counter (F7) and the transmit block (F5). The transmit block sends data out to the FMC block (replaced with a temporary block in this project). The delay counter will trigger the receive block (F6) which will receive data from the FMC block.

Once data has been received by the receive block, it is outputted to the Network Attached Storage (F9), by the Gigabit Ethernet block (F8).

4.3 **ARM Configuration**

The ARM processor on the Rhino controls all of the functionality of the board; it acts as the gateway into the Rhino. Without a way to access to the features of the Rhino provided by the ARM it is little more than an expensive doorstop, which means that communication with the ARM processor is vital.

The FTDI chip on the Rhino gives the first bit of connectivity to the ARM processor, which allows to the user to connect through a serial connection (as seen in Figure 4-5) over the USB to configure the ARM. The ARM processor has several boot options available; the first task in getting the Rhino configured was to decide which of these options best suits the needs of the user. These options were then reviewed to determine which of the options provide the functionality required, as well as which of the options make development on the platform as hassle free as possible.

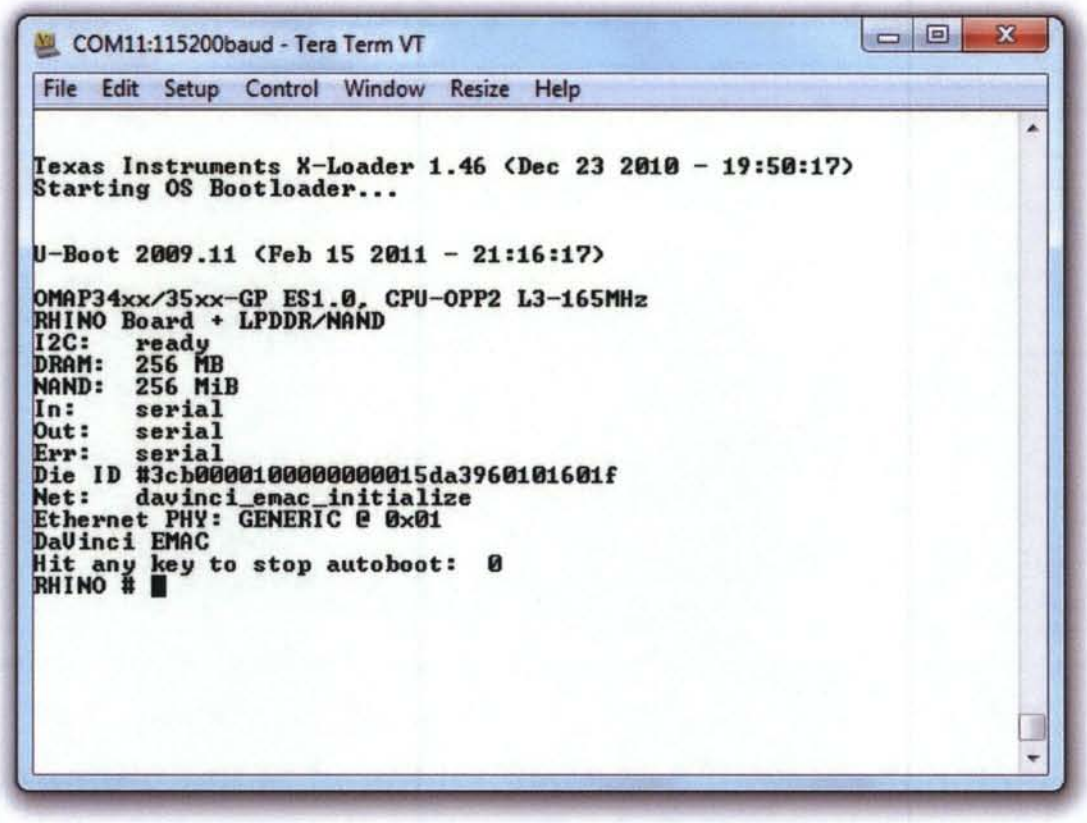

Figure 4-5: Terminal Client logged into U-Boot on the Rhino via FTDI serial link.

4.3.1 Arm Boot Configuration

As mentioned in section 2.4 the ARM processor has several boot options available to the user. Initial booting of the ARM on the Rhino requires the use of an SD card to provide the X-Loader, u-boot and Linux kernel. Once these have saved to the NAND flash the SD card is no longer required for booting.

In order to make development and usage of the system as simple as possible an NFS (Networked File System) was selected for the Linux kernel. Having the file system for the ARM stored on the control pc allows for easy modification of files on the ARM file system. This is

considerably simpler than the alternative of storing the file system on the NAND flash and using a protocol such as TFTP to transfer files.

With the ARM configured, a method for running applications on the FPGA was the next priority. This is due to the fact that while the ARM processor is a great tool, it is not the optimal device on which to be running applications. The Spartan 6 FPGA was incorporated onto the Rhino to do all of the "heavy lifting", and so that is where it is best to execute applications.

4.3.2 [F1] U-Boot Standalone Programmer

Programming the FPGA through the JTAG interface is inconvenient and requires a JTAG programmer. Having the ARM processor on the platform opens up a much more elegant and convenient way to program the FPGA. A connection between the serial programming pin on the Spartan 6 and an IO pin on the ARM provides a means to program the FPGA without requiring a JTAG programmer, or any other external connections. In order to allow easy development on the Rhino before a Linux Kernel was ported to Rhino a programmer was designed for u-boot. The desired process for the u-boot programmer can be seen in Figure 4-6: Flow Diagram of FPGA Programming Process through Uboot.

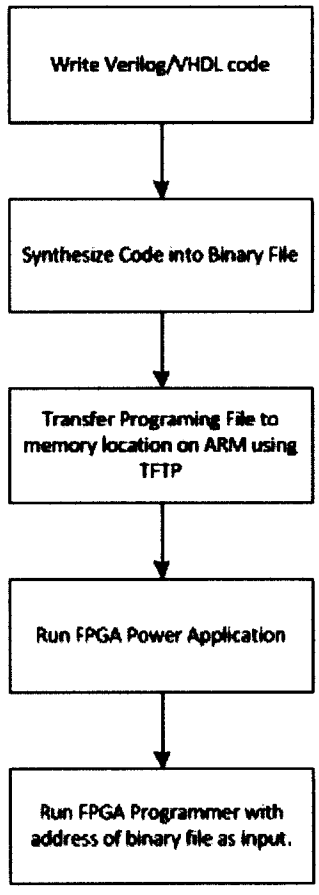

Figure 4-6: Flow Diagram of FPGA Programming Process through Uboot

U-boot is capable of small standalone programs by using the go command and giving the starting address of the application to be run. The FPGA programmer was to be coded in c, and needed to be cross-complied to be run on the ARM processor. Uboot was designed as a boot loader, and not as an operating system, and as such is not perfectly suited to the task of running applications. This adds some difficulties when designing standalone programs for u-boot. The first of these difficulties is that it does not handle applications with multiple functions particularly well. This is because of the way programs are executed. Programs are executed by telling uboot to run the application at a specific memory address, which works well if there is only one function as the entry point to the program is at the start of the memory area the program is saved in. When multiple functions are added to the function the main function is no longer necessarily at the start of the memory block the program is stored in,

creating difficulties when attempting to run the code. This combined with the lack of libraries available by default in u-boot makes programming more difficult.

In order to keep the program as reliable and simple to use as possible, the FPGA-programmer will be coded in a single function, avoiding the user having to calculate the correct entry point for running the program.

The second challenge in u-boot programming is the lack of a file system. Unlike an operating system, u-boot does not have a user friendly file system structure available, making tasks such as copying files slightly more complicated. Files are accessed through their actual address in memory, this makes the user's file input interface difficult and non-intuitive. To get around this a set memory location was decided upon in advance for both the application and the programming file, and described in the user guide for the programmer. This means that as long as the user stores the application in the correct location, there is no need for the user to worry about the lack of a proper file system. The third step in Figure 4-6 is copying the binary file to the ARM processor, for this step a pre-defined memory location is used.

Programming the FPGA

The programmer application makes use of the serial programming port available on the Spartan 6, which is connected to the SPI of the ARM through a level translator. In order to program the FPGA through the serial pin there is a set protocol that needs to be followed. There are two options for serial configuration, slave serial and master-serial. The programmer application uses the slave-serial configuration mode, which was designed for configuration on an FPGA from an external processor, as the ARM is the control processor and so it should be the master.

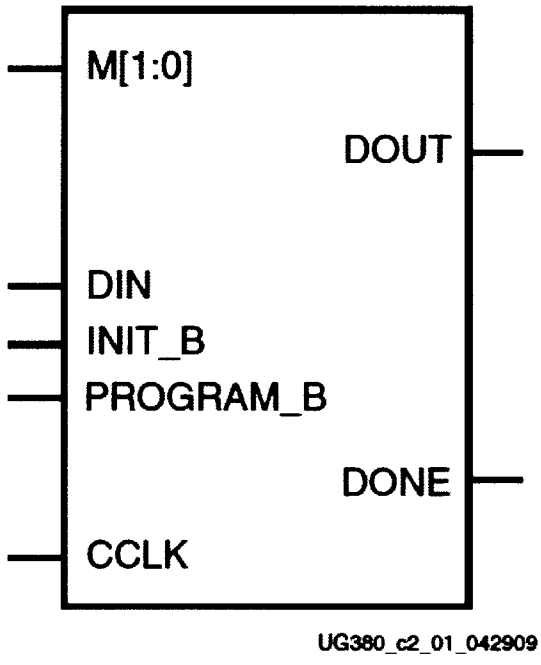

Figure 4-7: Serial Programming Interface for Spartan 6[38]

The serial programming link has five input lines and two output lines as seen in Figure 4-8. PROGRAM_B is an active low asynchronous full-chip reset [38]. INIT_B is used to indicate a CRC error [38]. CCLK is a clock line for clocking data. DIN is the serial data input that is used to transfer configuration data to the FPGA. DOUT is a serial data output for configuring daisy chained devices; this pin will not be used as there is only one FPGA. DONE is used to indicate that the FPGA has been programmed correctly. A table of these pins and there uses can be seen in Table 4-2: Configuration Pins of Spartan 6 Serial Configuration Table 4-2.

Table 4-2: Configuration Pins of Spartan 6 Serial Configuration [38]

Pin Name	Type	Dedicated or Dual-Purpose	Description
M[1:0]	Input	Dual-Purpose	Mode Pins – determine configuration mode
CCLK	Input or Output	Dual-Purpose	Configuration clock source
DIN	Input	Dual-Purpose	Serial configuration data input, synchronous to rising CCLK edge.
DOUT	Output	Dual-Purpose	Serial data output for downstream daisy-chained devices. (not used for Rhino at this stage)
DONE	Bidirectional, Open-Drain, or Active	Dedicated	Active-High signal indicating configuration is complete: 0 = FPGA not configured 1 = FPGA configured
INIT_B	Input or	Dual-	Before the Mode pins

	Output, Open-Drain	Purpose	are sampled, INIT_B is an input that can be held Low to delay configuration. After the Mode pins are sampled, INIT_B is an open-drain active-Low output indicating whether a CRC error occurred during configuration: 0 = CRC error 1 = No CRC error
PROGRAM_B	Input	Dedicated	Active-Low asynchronous full-chip reset.

The serial programming timing specification, as shown in Figure 4-8, needs to be followed correctly to program the FPGA. In order to initiate programming of the Spartan 6 the PROGRAM_B line is pulled low. This resets the chip preparing it to be programmed. After PROG_B is pulled low INIT_B is pulsed low to signal that the chip is ready to be programmed. PROG_B must be pulled high, and once the FPGA pulls INIT_B high again data transfer to the chip can begin.

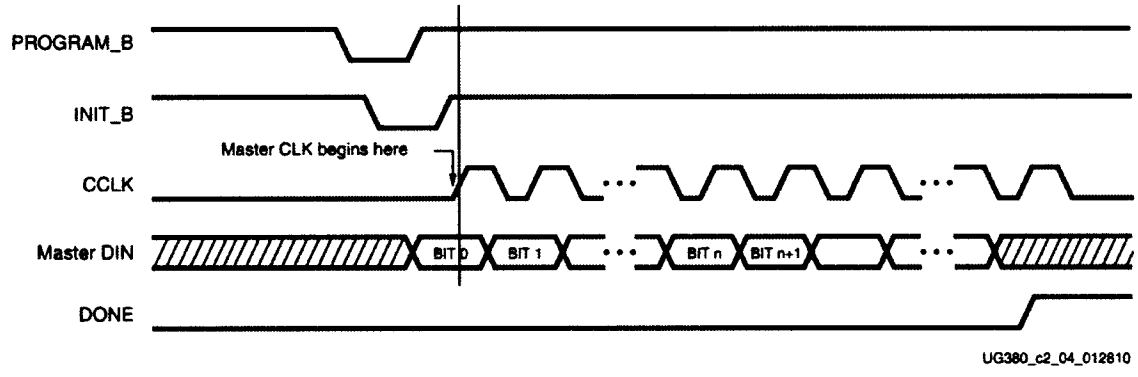

Figure 4-8: Timing Diagram of Serial Programming for Spartan 6 [38]

The level translator between the ARM and the FPGA means that a direction line was required to toggle INIT_B between an input and an output. In addition to setting INIT_B as an input on the ARM, the INIT_B_DIR line also needs to pulled low to toggle the direction of the line in the level translator. Because all the other lines are only used in one direction there is no need to have a direction line for any of them.

Once the INIT_B line is asserted, the ARM must initiate the data transfer of the configuration file to the FPGA. The configuration file needs to be a binary configuration file for the Spartan 6 (the file should have the '.bin' extension and not the default '.bit' extension).

Data transfer to the FPGA is done using the SPI interface on the ARM. The GPIO pin on the ARM that is connected to the DIN pin on the FPGA (through the level translator) is set to be used as an SPI pin in the u-boot configuration files. The application must configure the SPI for the transfer. The SPI was configured to use the maximum clock rate of 48 MHz [15]. A binary

programming file is 4122Kb; combined with a 48MHz clock this gives a minimum programming time of 0.7s.

Data is then read from memory and transferred to the SPI buffers to be clocked into the FPGA. Because there is no need to read data back from the FPGA (the data line is unidirectional anyway) it is possible to transfer 16 bits at a time to the SPI buffer, instead of reading 8 and writing 8.

Once all the data has been clocked into the FPGA the DONE pin is checked to confirm that the FPGA has been properly programmed, if it has not the user is informed through an error message. If at any time during the data transfer INIT_B is pulled low by the FPGA, then a CRC error has occurred, and the programming must be cancelled and an error reported. The programming process described above is summarised in Figure 4-9.

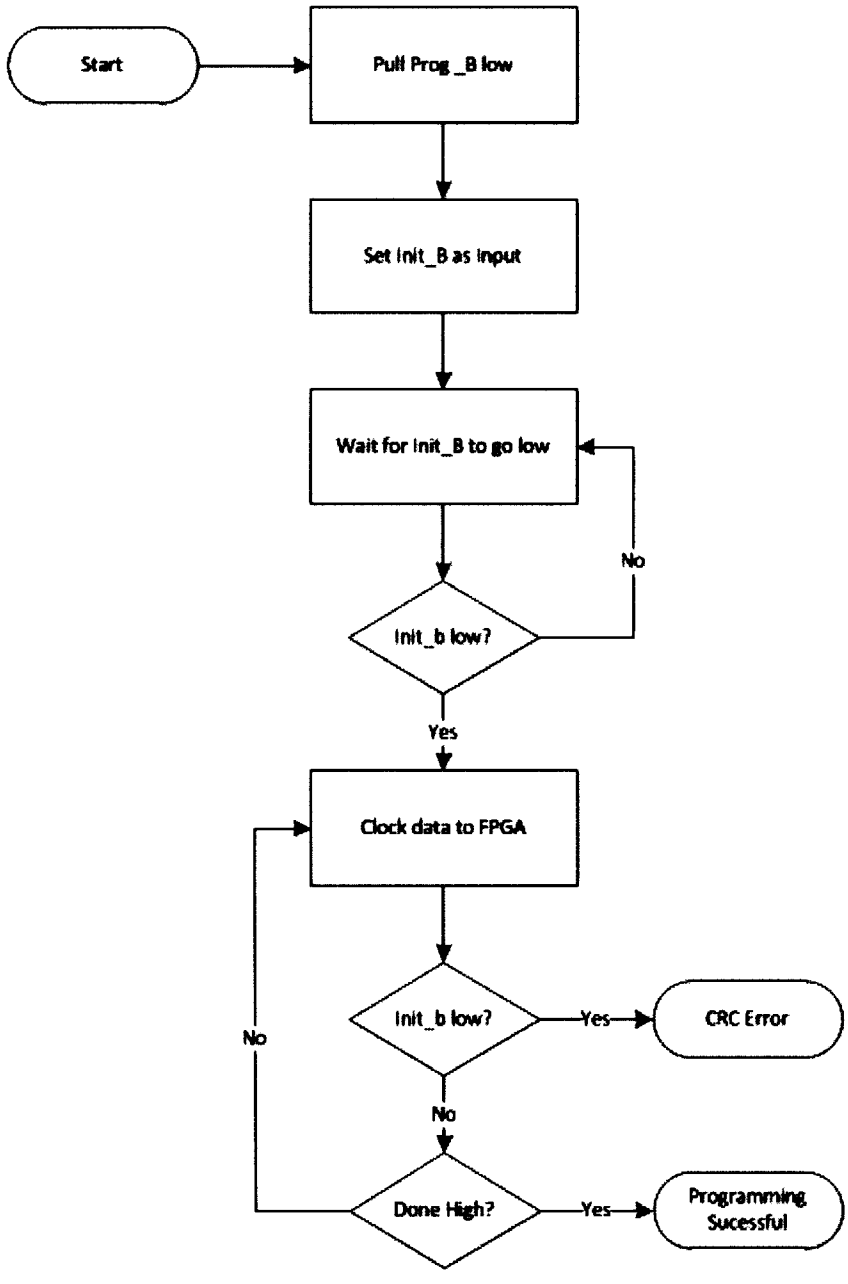

Figure 4-9: Flow Chart of the FPGA Programmer Operation.

Once programmed the FPGA will immediately start executing the configured gateware, and the red FPGA_NOT_CONFIG LED will turn off. It is important to note that before the FPGA can be configured its power supplies must first be enabled. To simplify the programming of the FPGA, enabling the power supplies was added to the programmer.

4.3.3 [F1] Linux Programmer Port

Programming the FPGA from u-boot is a useful tool, but it is not appropriate for general use. U-boot was not designed as an operating system and using it as such does not provide a good user experience. A Linux based operating system is the go-to choice for the Rhino board. The flow diagram for programming the FPGA from Linux can be seen in Figure 4-10.

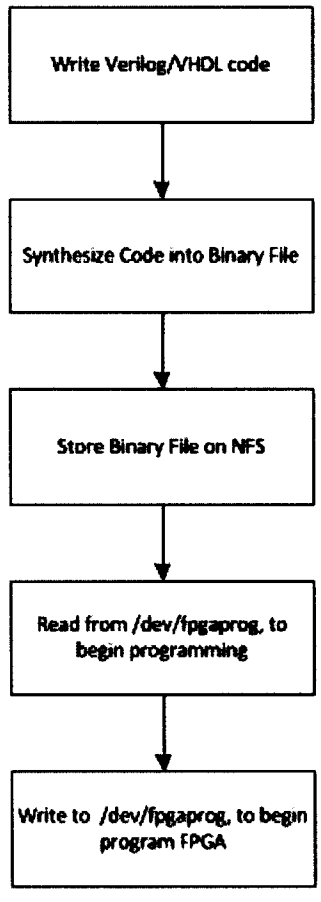

Figure 4-10: Flow Diagram for Programming FPGA from Linux

Programming the FPGA through Linux requires an identical procedure to the one described in section 4.3.2 above; however using Linux provides additional features and some additional challenges.

Because the programmer needs to read and write to the GPIO pins part of the programmer needs to be written as a driver in kernel-space. A user-space application is also required to give the user a way to access the programmer. In order to reduce the complexity of the system, it was decided to make use of existing Linux functions as the user-space software. This leaves the core functionality in a char driver running in kernel-space. The driver will create a file in the /dev directory in the Linux file system, which when written to, will program the FPGA with the file written.

Before the FPGA can be used, the ARM needs to enable its power supplies. In u-boot a separate utility was used to enable the power supplies. To make programming the FPGA simpler and to avoid confusion, code to enable the power supplies is added at the start of the Linux programmer. Enabling the power supplies requires pulling VCCINT_EN, VCCO_AUX_EN and VCCMGT_EN high. These pins are connected the ARMs GPIO pins 99 to 101.

The driver was split into two sections, one to get the FPGA ready for data transfer, and the other to actually transfer the data over the SPI interface. This allows for one section to power on the FPGA, setup the SPI, and prepare the FPGA for data transfer, and the other section to copy the data to the FPGA. The split in these two functions allows the user to reset the FPGA and prepare it for programming, without needing to program it immediately.

To facilitate ease of use, the driver automatically creates a file in the /dev/ directory that will allow the user to access its features using standard Linux file handling, such as the "cat" and "dd" functions.

To program the FPGA with the programmer module inserted into the kernel, the user first reads from the programmer driver, using a command such as "cat /dev/fpgaprog". This will setup the SPI, power on the FPGA and prepare if for programming. The user will receive a notification that the FPGA is ready to be programmed on the appropriate SPI number. The user can then use the cat command to copy the binary configuration file to the FPGA (cat

congifgfile.bin > /dev/fpgaprog). The FPGA will then be programmed with the supplied configuration file and the status of the operation will be returned.

4.3.4 [F2] GPMC Bus – Linux

The data bus between the ARM and the processor is the primary means of controlling the FPGA. As described in 2.4.1 the ARM is connected to the FPGA through its GPMC, because of this the FPGA is seen by the ARM as if it was a standard memory device. The FPGA will have gateware running on it to translate the signals coming from the ARM; this is described in 4.4.2 below.

The bus being developed consists of two parts. The first part is a tool that will allow users to easily create their own bus driver, customized to their own needs; the second is an implementation of the first specifically for the radar block. The user interface for controlling the bus implementation will be same as is used in the serial programmer. A char driver will allow users to read and write data to the bus as if it were a file in the file system.

The GPMC has a 16 bit data bus and 10 bit address bus, as discussed in 2.4.1 both the address and the data bus can be used for addressing, giving an addressable space of 2^27bytes. The GPMC was designed to interface with multiple memory devices, and as such has several select lines to control which device is being written to. Normally these lines would be wired to the enable pins of the memory devices with which they are associated. In this case however all multiple select lines go to the FPGA, further increasing the address space.

The ARM reserves address spaces for each block of memory mapped to the GPMC as seen in Table 4-3. When an address in this space is written to or read from, the GPMC asserts the correct select line and performs the required action. In order to make the bus as easy to customize as possible, it was decided that the addressing on the ARM should match the addressing on the FPGA. In Table 4-3 it can be seen that the CS lines connected to the FPGA do not constitute a continuous space in the ARM address space, and the addresses do not start at

address zero. On the FPGA it makes sense to start addressing from zero and make the address space continuous. In order to make the two address spaces the same, the driver software will translate the address on the user side (which will match the addressing on the FPGA) into the address required to be written to on the GPMC.

Table 4-3: Address information for CS lines on GPMC bus

CS line	Start Address	End Address
CS1	0x08000000	0x0FFFFFFF
CS2	0x10000000	0x17FFFFFF
CS3	0x18000000	0x1FFFFFFF
CS4	0x20000000	0x27FFFFFF
CS5	0x28000000	0x2FFFFFFF
CS6	0x38000000	0x3FFFFFFF
CS7	Unused	Unused

For the driver to be able to write to these sections of memory they must first be remapped (using ioremap) so that the physical addresses are available to the driver. Once the memory has been made available, the driver can write to the GPMC and the data written will be transmitted on the bus lines. The GPMC handles all the timing for the connection automatically. Reading from the GPMC follows the same process as writing to it.

It is important to note that in order to use the bus efficiently at least 2 bytes should be written or read at a time; otherwise only half the bus is being used.

As with the FPGA programmer, the ARM-FPGA bus can be written to or read from by using standard Linux tools such as "cat". The driver automatically generates a file in the /dev folder that can be accessed by the user.

The code generated for this driver writes all the input data starting from beginning of the addressable space. For many applications multiple instances of this code will need to be used

to allow writing to different parts of memory. For this project, all the registers are written to and read from at the same time, as the amount of data is fairly small, and the time taken to read and write all the data is not significant.

4.4 Supporting Gateware

This section discusses gateware that was created to be run on the FPGA either to support the operation of the radar block, or to aid in developing the system

4.4.1 Serial Debugging

To enable easier debugging, a debug board containing an RS232 level converter can be connected to Rhino (While this is not required for the functionality of the system it is included as it can greatly aid the debugging process). Serial communications through the level translator required gateware to control process.

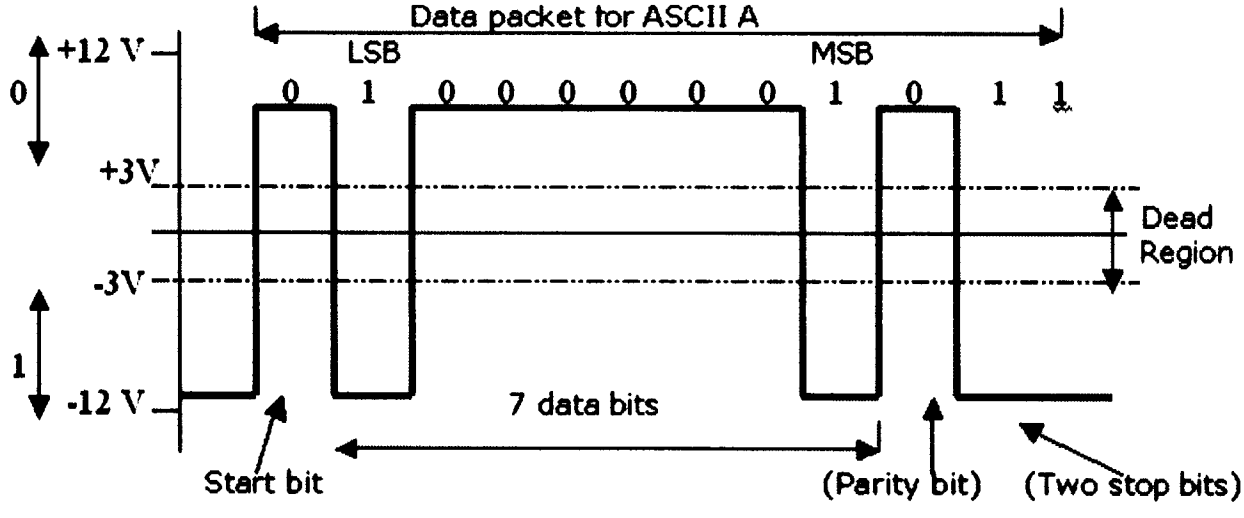

Figure 4-11 Timing Diagram for RS232 [39]

The gateware for the serial bus is a state machine which will execute the timing shown in Figure 4-11 above. The transmitter will pull the line low to begin a transmission of a byte. After putting the 7 data bits and parity bit on the line, the line is held high for two clock cycles which forms the stop bits.

The receiver follows the same timing diagram, except that it waits for a start bit before receiving the byte.

4.4.2 [F2] FPGA-ARM Bus – Spartan 6

The first half of the bus between the FPGA and the ARM processor was described in section 4.3.4 above. The data coming in from the GPMC on the ARM uses the timing specified by the GPMC settings on the ARM controller. Timing diagrams for read and write operations are shown in Figure 4-12 and Figure 4-13.

The gateware for the data bus needs to take the incoming signals from the ARM processor and output them in a form that is easy to understand and use. Because the ARM treats the FPGA as a memory device, it makes logical sense for the output from the bus gateware block to be in a form compatible with the memory devices used on the FPGA.

Wishbone is the recommended bus standard for use on the Rhino, and as such the output of the bus is designed to be wishbone compatible. This allows the bus to be used with any other wishbone compatible block. In order to do this the address needs to be translated and buffered as a single register.

The FPGA has multiple CS lines running to it from the ARM; each CS line represents a different block in the virtual memory space of the ARM. These CS lines must be included when creating the buffered address to be output.

Timing

In Figure 4-12 a timing diagram for the write operation on the GPMC can be seen. The FCLK line in this diagram is internal to the GPMC, while the CLK line is connected to the FPGA and is used for the clocking of the data. The start of the write is signalled to the FPGA by the pulling

one of the cable select lines (shown in the Timing diagram as nCS to represent active low line) and the Address Valid line (nADV) low.

Pulling the nADV line low indicates to the FPGA that the information on the data and address busses represents a valid address. The FPGA needs to concatenate the values on these two lines and buffer them. This buffered address is not the final address for outputting. Because the ARM considers each CS line as being connected to a different memory space, the addresses are repeated. This means that if the ARM was to write to the full addressable space and just the address on the FPGA was used data would be overwritten multiple times.

In order to prevent overwriting of data there are two simple options to be used. The first is output the CS lines and connect the output to multiple memory blocks, using the CS lines as enable lines on the respective blocks. The other option is use the information on the CS lines to recreate the original address written to on the ARM processor.
To increase the usability of this bus it was decided in section 4.3.4 that the addressing on the ARM should exactly match the addressing on the FPGA. Because the user specifies an address when writing to the FPGA and not a CS line, the output should match this; and thus the second option will be used.

An additional 3 bits is added to the beginning of the address based on which CS line has been asserted. The final address is then buffered for output.

Then next stage in the write is the data transfer. The GPMC will write the data to the bus, unset the valid address flag (nADV) and pull the write enable line low (new). When the write enable line is active, the FPGA bus block will buffer the output data and set its own write enable line. In order to do this a tri-state buffer is required for the pins of the data bus. This is because the same pins will be later used to write data out. When the write enable line is active, the tri-state buffer needs to set the data bus to be used as a high impedance input. At all other times the buffer will be set as an output.

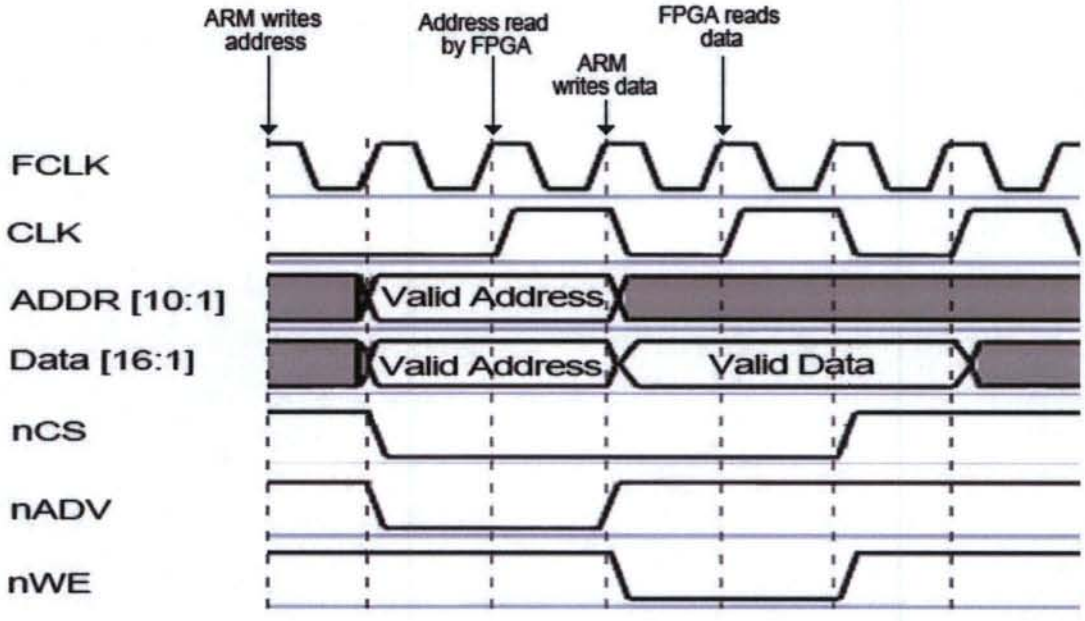

Figure 4-12: Timing diagram for write operation on GPMC bus. [15]

The read process is fairly similar to the write process. The transfer is again controlled and clocked by the GPMC. The transfer is again initiated by writing the address to the data and address busses, and setting the nCS and nADV lines low. The FPGA will read the address from the bus in the same fashion as it did for the write operation above.

In the write operation, at this point the write enable line would be activated, however for the read operation, the output enable line is pulled low (nOE). When the FPGA bus block detects the output enable line being asserted, it puts the data stored at the requested address on the bus.

On the next falling edge of the clock the GPMC will read the data off the bus and return the cable select and output enable lines back to their default high values.

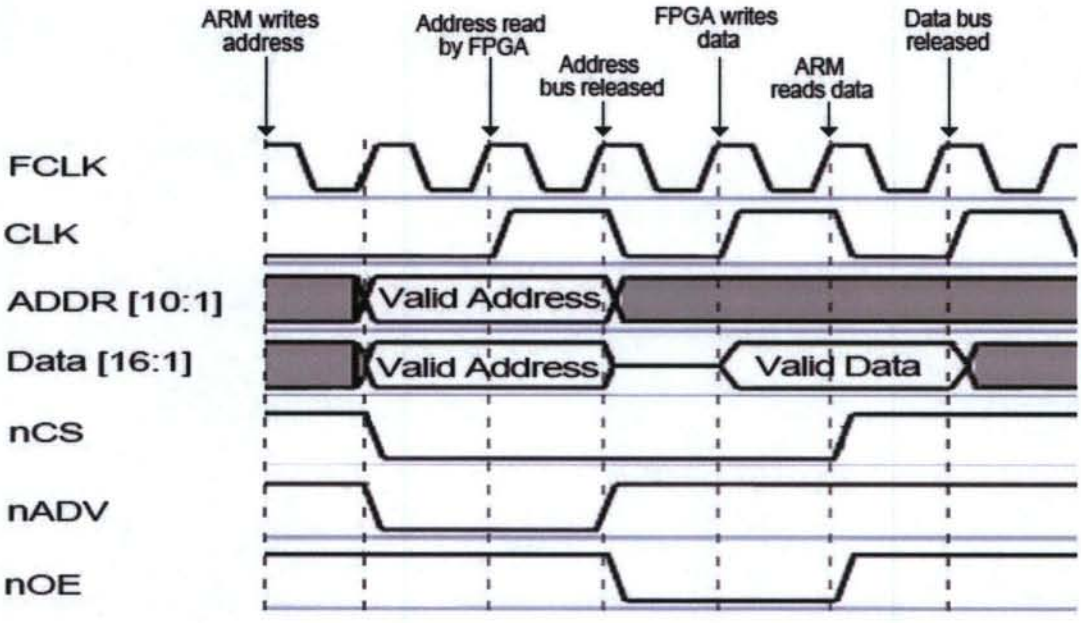

Figure 4-13: Timing diagram for read operation on GPMC bus. [14]

4.4.3 [F8] Gigabit Ethernet

The design for the Gigabit Ethernet for this project revolves largely around the time available for development. As a new platform there is much interest in getting applications running as soon as possible to show what the platform is capable of. For this reason one of the design requirements for the gigabit Ethernet is that it be implementable as quickly as possible.

The simplest options for getting the gigabit Ethernet working are to either use a soft-core processor (such as a Microblaze) to implement at TCP/IP stack, or to use gateware to control UDP communications. Because of a preference towards a design not using a soft-core processor, the UDP option was selected. It should also be noted that for this application the

transmission of data is only required in one direction, so the network block being designed is only capable of transmitting data.

The Rhino has a PHY chip which connects the FPGA and the Gigabit Ethernet connection. In order to use this, a MAC block is required to control the networking. Due to limited time implementation a new MAC was not a viable option, and considering the availability of compatible open source MAC cores is a waste of resources.

The MAC core used in the USRP is compatible with the Rhino hardware, and is open source. For these reasons it was selected as a starting point for the implementation of the gigabit Ethernet connection.

With time being the primary constraint for development of the Ethernet block, the design was kept as simple as possible. A predefined packet header is coded into the UDP control block that simplifies packet creation. Data is input into the packet shell as needed.

Data to be output by the Gigabit Ethernet is stored in a FIFO memory block. A state machine is used to control the configuration of the UDP as well as the transmission of packets to the MAC layers. Once the state machine reaches the point where the Ethernet is ready to send data, it waits for enough data to be stored in the FIFO to fill a packet. Once there is enough data, it transfers one packet of data, and then waits for enough data to be available again.

There is a flag available that will trigger the block to send a packet that is not yet completed. The packet is padded with zeros in order to fill the full packet size. When the last byte of data for a specific pulse is received this flag is set to allow the output buffers to be completely emptied before the next pulse arrives.

The pre-defined packet means that the source-address, destination-address and packet size are all configured at compile time, and must be adjusted to suit the required implementation. The

size of the packets will vary the rate at which data can be transmitted, because there is a set overhead in the transmission of each packet to store Ethernet frame information.

While the Gigabit Ethernet block was designed to be useable by multiple applications, it was designed with the radar block in mind and is tailored to best suit the operational requirements of the radar block. The design of the radar block will be discussed in the following chapter.

5 Radar Block Design and Implementation

The Radar block is one of the primary outputs of this project, in this chapter the design process for the Radar block will be viewed and discussed. The radar block is intended to be used as a stepping stone for the development of radar systems on the Rhino. While unfortunately actual ADCs and DACs will not be available for this project, it should still serve as a good proof of concept

The aspects of the Radar block, such as the parameters that it requires, the timing and control of the system will be described. The process of setting the parameters, starting and stopping the radar block and reasons for the selections of these processes will all be discussed.

The final output of the block is what is most relevant to users, and the way this output is handled will be discussed in the final section of this chapter.

5.1 High Level Design of Radar Block Implementation

The Radar Block design can be split into two distinctive sections. The first is the design of the actual Radar Block, and the second is the design of a control block that is used to link the radar block with other components, and control the operation of the radar block. Both the radar block and the control block are primarily containers that hold other functions as shown in Figure 5-1.

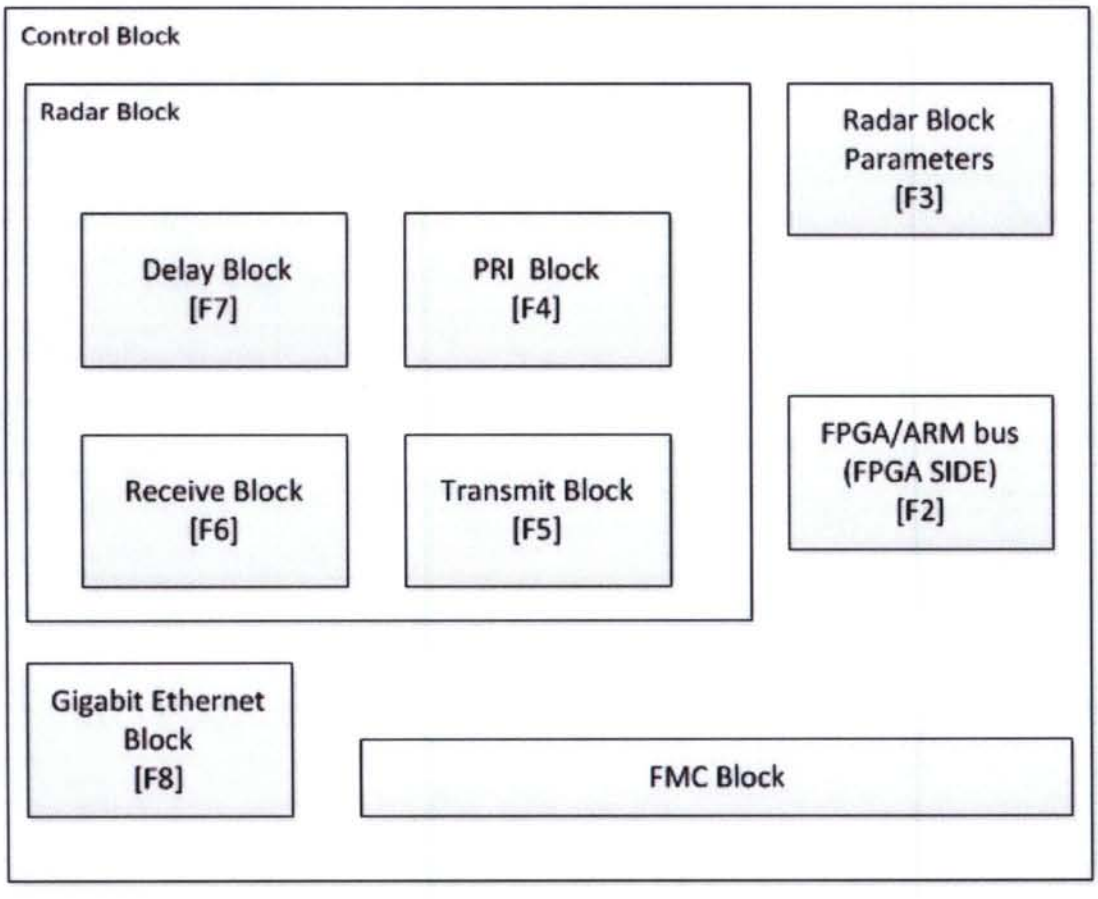

Figure 5-1: Diagram showing components of the Control Block

The Control Block instantiates all the components that will make up the radar block implementation, these are the Radar Block, the parameters block, the Bus block, the Ethernet block and the FMC block. The control block will also handle the interconnection of these blocks.

The radar block houses the four primary radar functions: Transmit Receive, PRI and Delay Blocks. The PRI and Delay blocks control the timing of the radar block, while the transmit and receive blocks perform the Input/output role.

5.2 [F3] Radar Block Parameters

The parameters in the radar block need to be updated from the ARM processor over the GPMC bus. All the parameters in the block are changed in one read. The data structure used for this transmission is shown in Figure 5-2. The data is ordered so that the parameters that are smallest to change are sent first, so that they can be updated without sending the full data frame. Note that even though the pulse length is smaller than PRI and delay it is stored after them; this is because the pulse length is always changed together with the much larger wave data parameter.

Start/ stop	PRI	Delay	Wave Length	Wave Data
2 bytes	4 bytes	4 bytes	2 bytes	4 – 32 384 bytes

Figure 5-2: Parameter update data frame

Start/Stop

The first byte of the control frame is the start/stop frame. It is padded to 2 full bytes in length rather than a single bit, because that is the size of the data bus, and it is important that the start bit can be changed without having to send any other parameters. If only a single word is sent over the data bus, it will only modify the start/stop bit, allowing the user to start and stop the system without re-sending parameters. The additional bits in this packet will be used in future to store additional parameters specific to the front-end that is being used.

Pulse Repetition Interval (PRI)

The Pulse Repetition Interval (PRI) is used to set the Pulse Repetition Frequency (PRF) of the radar. To implement this interval a counter is used, that will count to a value stored in the PRI register. The value in the PRI register is updated when the data frame sent over the ARM-FPGA

bus is longer than 2 bytes, the next 4 bytes sent after the stop start bytes set the PRI. In section 4.1.3 it was shown that to implement the desired pulse interval a register 26 bits wide is required. This is again rounded to fit the size of the data bus, and is sent as two 16bit words to fill a single 32bit register.

When changing the PRI the interval between the last pulse sent before the change, and the next pulse may be unpredictable. This is because the timer may have already counted past the new PRI value, in which case the pulse is sent immediately, or it may have not reached the new PRI value, in which case the pulse is sent with the correct delay between pulses. It is suggested that the system be stopped when changing the PRI to avoid this.

Delay

The next 4 bytes in the data frame represent the delay between transmission of a pulse, and the activation of the receiver. The delay is implemented in a fashion which is identical to that of the PRI.

Pulse Width

The pulse width is needed in order to make sure that the block sends the correct number of samples to the ADC. In the transmit block, samples are sent from sequential addresses in memory, until the pulse width is reached. This prevents the transmitter from sending undesired data. The width is stored in a 16 bit register and is updated by the two bytes of data in the data frame that follow the delay. If the pulse width is not entered correctly it will impair the functionality of the radar, as the transmitter will always transmit a number of bytes equal to the pulse width regardless of the actual length of the pulse.

Pulse Waveform

The final piece of the data frame is the pulse that is to be sent, with a variable length of between 4 and 32384 bytes. It is stored as a series of 32 bit numbers, the first 2 bytes representing the real part of the sample, and the second two representing the complex part. Within the radar block the samples are treated as a single 32bit number. When ADCs/DACs are used (as will be the case in an actual implementation of the radar block) the 32bit output from the radar block will be split into two 16bit numbers, and sent to different channels. This 32bit format for the waveform data was selected, because the upper limit on the required pulse size was given as 8096 16bit complex samples and it is easier to handle a single 32bit data word than two 16bit words. Because many ADCs/DACs are not capable of handling 16 bit numbers, it is likely that a few of these bits will be unused; this should not affect the operation of the system.

The actual pulse used does not impact the functionality of the radar block. This means that a radar can use any pulse that can be represented by 32384 bytes of data, and is not restricted to a specific pulse.

5.3 [F4] Pulse Repetition

The PRI block is responsible for repetitively generating transmit triggers, in order to generate the required PRF. The PRI register value is used as a goal for a counter, when the goal is reached, the correct duration of time will have passed and a transmit trigger will be generated. Figure 5-3 shows the state machine for PRI operation.

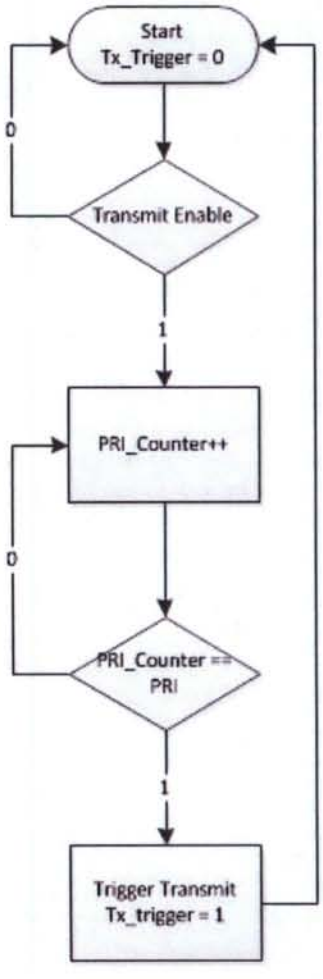

Figure 5-3: State Machine for PRI operation

5.4 [F5] Pulse Transmission

The pulse transmission function is responsible for ensuring that the pulse is transmitted to the DACs. When triggered, the function will transfer one pulse to the FMC block which controls the DACs; a simple state machine for this operation is shown in Figure 5-4. This function is clocked at a rate equal to that used by the DAC, for this project a 100MHz clock is used as implementing DACs is outside the scope of this project.

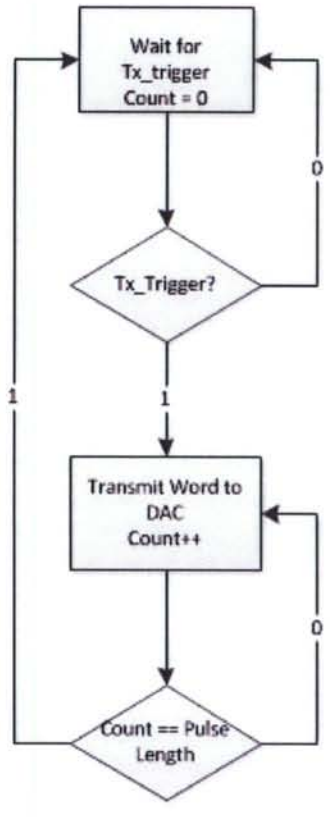

Figure 5-4: State Machine for Pulse Transmission Function

5.5 [F6] Pulse Reception

The pulse reception function is responsible for capturing incoming data from the ADCs. When triggered, the function will capture one pulse incoming from the FMC block that controls the ADCs; a simple state machine for this operation is shown in Figure 5-4. This function is clocked at a rate equal to that used by the ADC, for this project a 100MHz clock is used as implementing ADCs is outside the scope of this project.

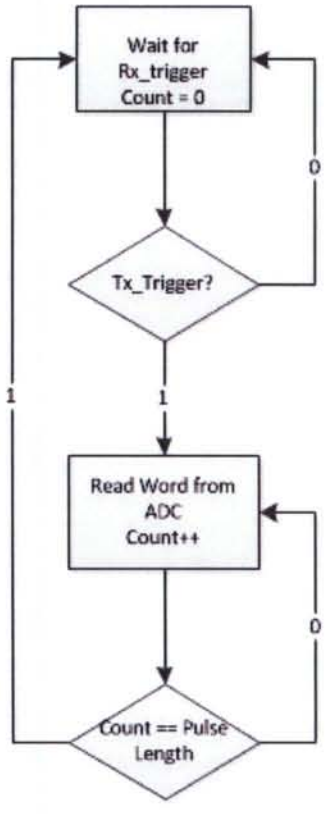

Figure 5-5: State Machine for Pulse Reception Function

5.6 [F7] Receive Delay

The delay block is responsible for delaying the operation of the receive block by a desired time period after the start of pulse transmission. The Delay register value is used as a goal for a counter, when the goal is reached, the correct duration of time will have passed and a transmit trigger will be generated. Figure 5-6: State Machine for Receive Delay Function shows the state machine for receive delay operation.

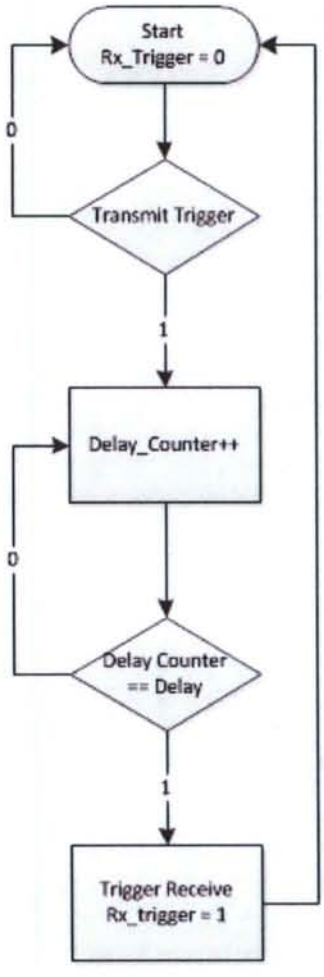

Figure 5-6: State Machine for Receive Delay Function

5.7 Timing Aspects

The systems primary clock is a 100MHz clock, which is generated on the Rhino, and sent to the FPGA through a differential clock input. This clock is buffered, and used as the clock for most of the functionality of the block. The 100MHz clock is used to implement the PRI, as well as to control the main radar block state machine. The receive delay requires a 200MHz clock to function with the correct resolution.

Because ADCs/DACs are not available for this project, the 100MHZ clock will also be used to control the timing of the receive and transmit block. In an implementation using ADCs/DACs either a clock provided by the DAC would be used, or the 100MHZ clock would need to be multiplied to match the required clock speed. The same is true for the ADC.

The system is started and stopped by the start/stop bit. When the start bit is received the counters in the PRI and delay blocks are reset. The PRI counter starts counting as soon as transmitting is enabled; this gives time for any pre-pulse setup that is required for the frontend.

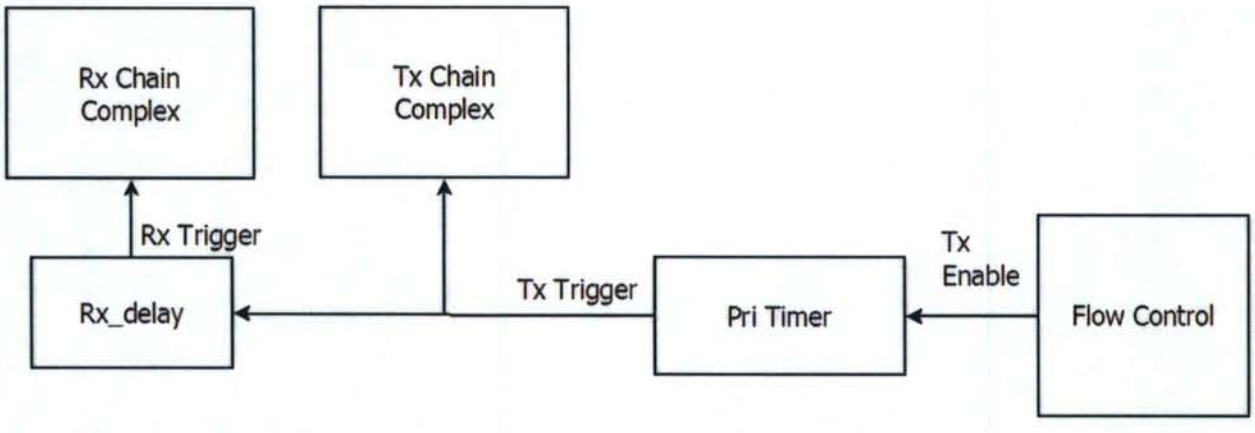

Figure 5-7: Block diagram of receive and transmit triggers

The PRI delay block counts at 100MHz until it gets to the value in the PRI register, at which point it triggers the transmitter module. This trigger is also sent to the receive delay block as seen in Figure 5-7: Block diagram of receive and transmit triggers above, and triggers a timer to start the receive delay. Again counting up at 100MHz the delay block counts to the required delay as specified in the delay register.

Data received is buffered and sent to the FIFO (First In First Out) to be written out to the Ethernet. Because it is likely that a clock speed other than the 100MHz system clock is being used for the receiver, the input clocking on the FIFO needs to have 2 clock inputs. One clock is used for writing to the FIFO, and the other for reading from it

The clocking on the gigabit Ethernet is described in section 4.4.3 above.

5.8 System Control and Connectivity

In order to link the various functions of the system together, the idea of a system container block is introduced. The system control block is simply a wrapper that is used to thread all the other blocks together. In the control block, the system inputs and outputs are declared and all the individual blocks that will make the radar block are wired up. The interconnections of the various modules can be seen in Figure 5-8 below.

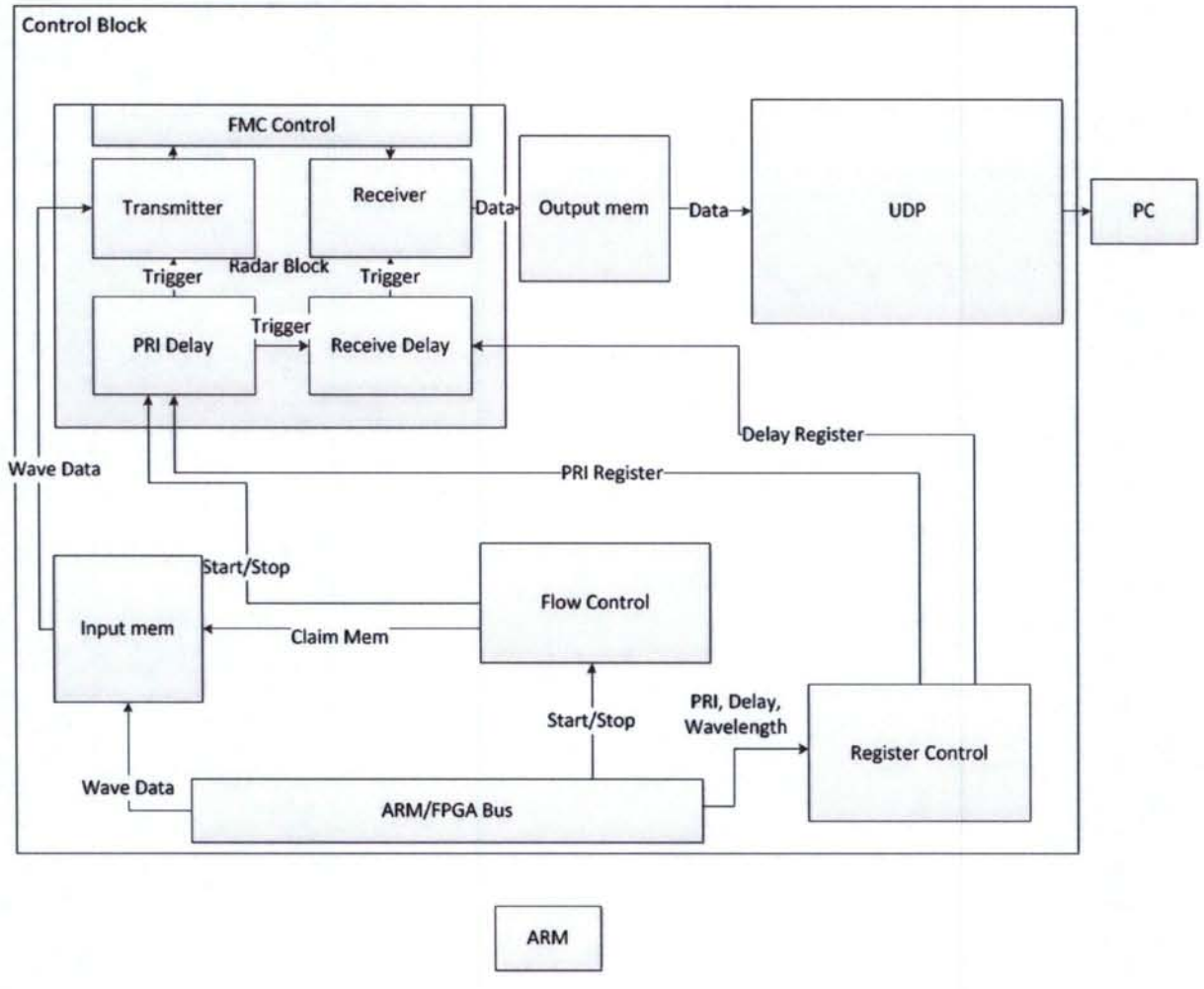

Figure 5-8: Block diagram showing interconnection of modules within the control block

It is in the control block that the primary control state machine for the radar is found. A block diagram for this state machine can be seen in Figure 5-9 below. The purpose of this state machine is to control the starting and stopping of the radar block. When the start bit is received from the ARM via the ARM-FPGA bus, the PRI timer is enabled, by pulling the enable transmitting line high. This line is wired to the PRI block, which will start counting from zero to the goal PRI as soon as transmitting is enabled. When transmitting is disabled, the PRI counter will not function, thus stopping new pulses from being sent.

When transmitting is enabled, the wave memory is claimed by the FPGA, preventing the arm from altering this memory while transmission is enabled. This prevents any errors that could occur from changing parameters mid pulse. In order to change parameters, the stop bit must first be sent, after which the bus will be released by the control block.

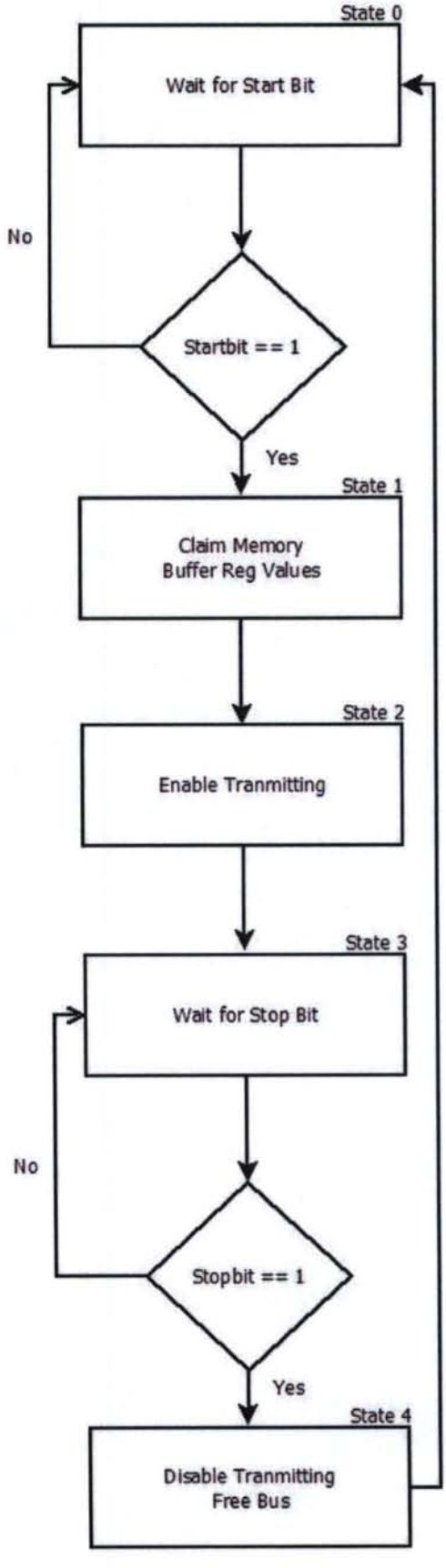

Figure 5-9: Control State Machine for the radar block

5.8.1 Control Block Connectivity

As mentioned previously, the control block links all the pieces of the radar configuration together. It not only links the blocks together, it also connects the blocks to the correct pins on the FPGA, and declares all the inputs and outputs of the system. Figure 5-10 shows all the inputs (left) and outputs (right) of the control block.

The inputs labelled "FPGA_PROC_BUS" are all inputs for the FPGA-ARM bus. The functionality of this bus is described in section 4.1.4. The FPGA_PROC_BUS_D is both an input and an output.

The next group of inputs and outputs are the "GIGE" pins. These pins are all connected to the PHY chip for the gigabit Ethernet, and the control block connects these all to the UDP block.

SYS_CLK_N and SYS_CLK_P are the two 100 MHz differential clock pins. These are buffered to create the 100 MHz clock that is used for clocking in most of the modules used.

The GPIO and LED inputs are used primarily for debugging. They give simple inputs and outputs to the system that do not require any of the other gateware in the block to be functioning (i.e. if the ARM-FPGA is not functioning to send input, the GPIO can be used)

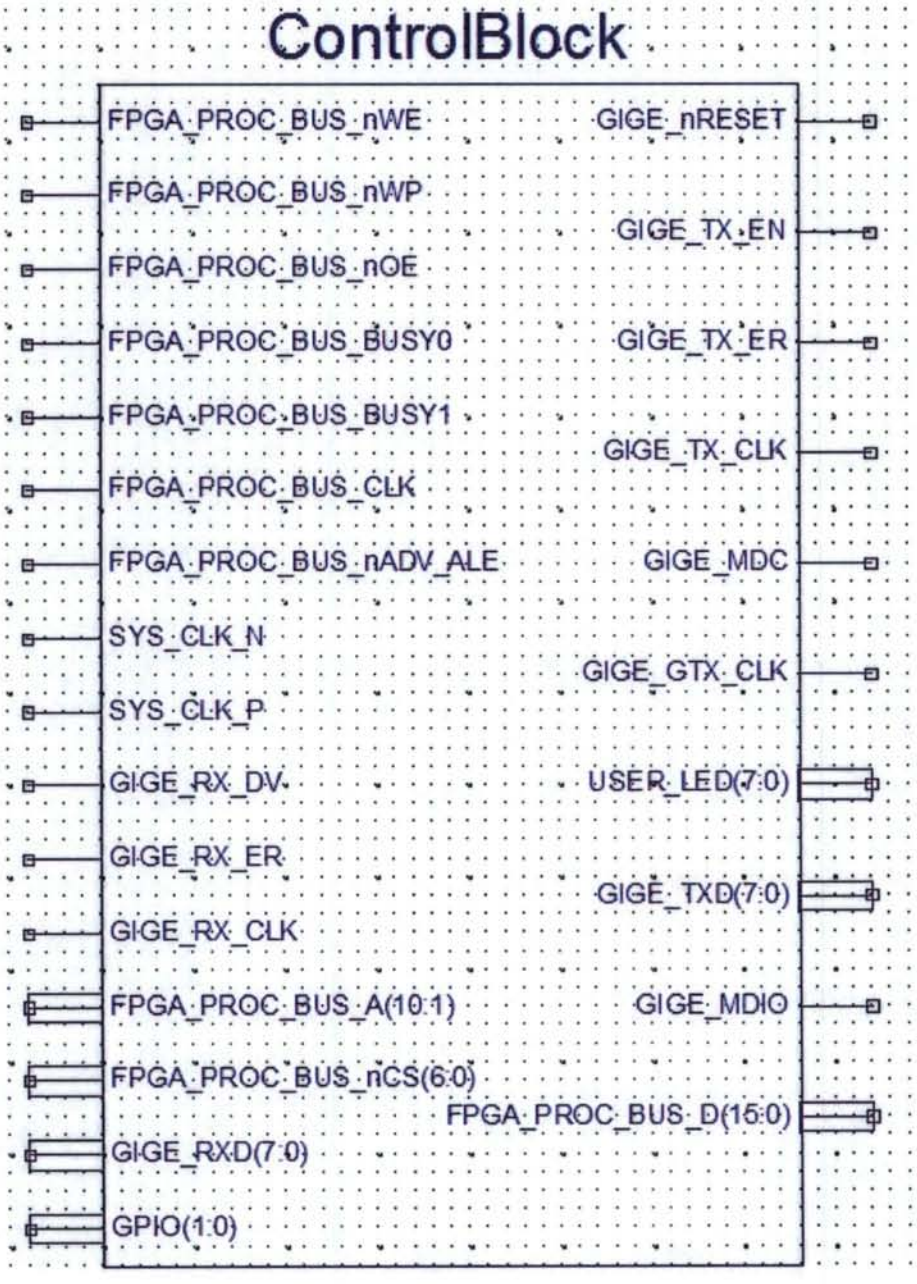

Figure 5-10: Diagram showing inputs and outputs of the control block (inputs on left, outputs on right)

In the control block the radar module, UDP module, memory blocks and FPGA-ARM module are all linked together. The connections made to all of these blocks are described in the sections that follow.

5.8.2 Radar Block Connectivity

This module controls all of the core functionality of the radar system. It is implemented separately so that modules such as the ARM/FPGA bus and the UDP module can be easily reused, without having to remove un-needed code.

The radar block itself has several inputs and outputs that need to be connected. In Figure 5-11 the inputs and outputs of the radar block are shown.

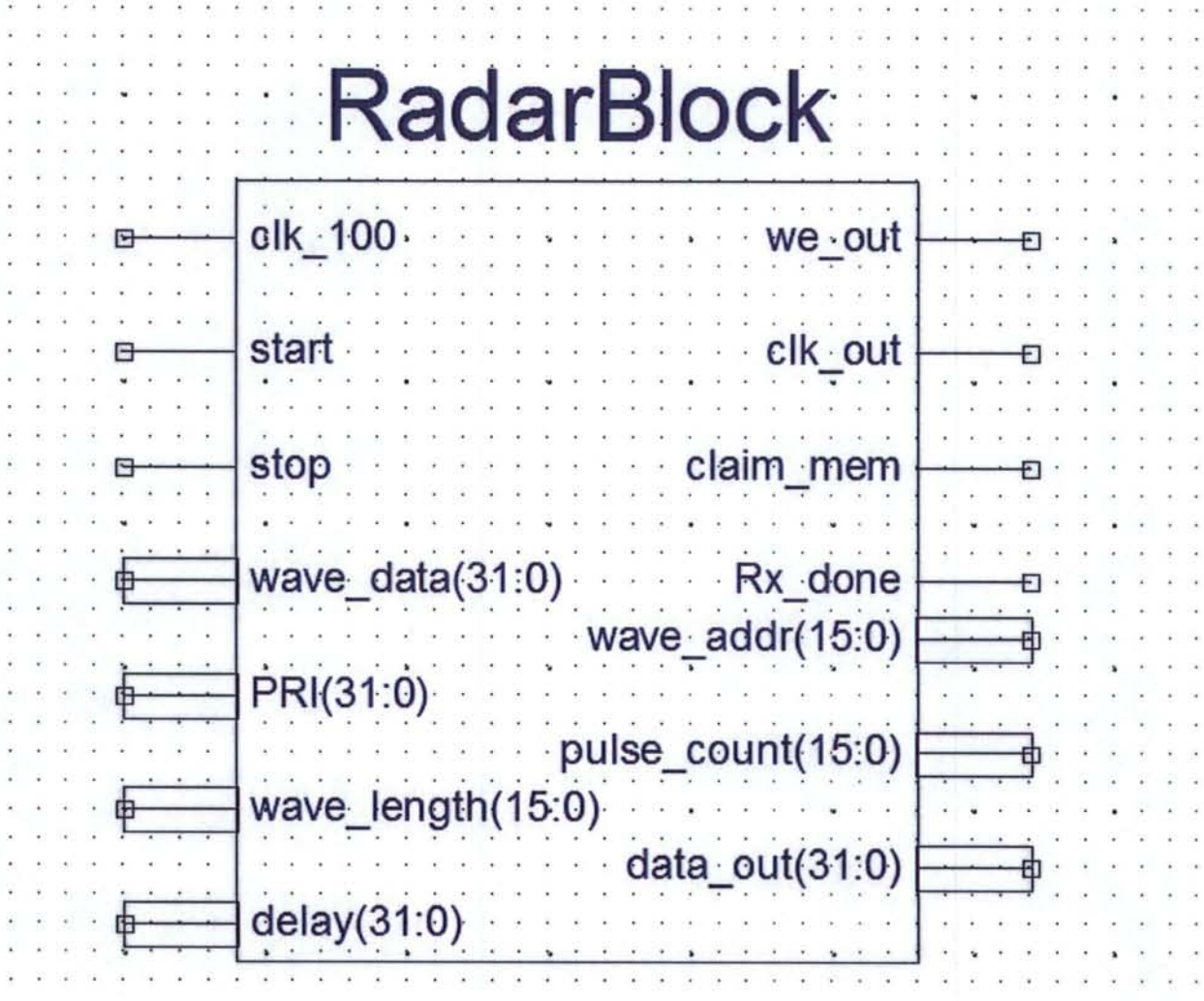

Figure 5-11: block diagram showing Inputs and outputs for radar block. (Inputs on left, outputs on right)

Inputs

The incoming clk_100 signal is linked to the 100MHz system clock. The start and stop bits are registers in the control block that are varied by the user through the FPGA/ARM bus. Wave data is a 32 bit bus that connects to the data output of the memory where the waveform is stored. PRI is a 32 bit input that is linked to the PRI register in the control block which is also controlled by the user through the ARM/FPGA bus. The wave_length and delay variables are also linked to registers in the control block, and are updated in the same way.

Outputs

The we_out output is connected to the output memory. This is used to signal to the memory when data that has been received is ready to be written to memory. The clk_out is also connected to the output memory. The reason a separate clock is required, is because the ADC may be clocking in data at a rate other than 100MHz as explained in section 5.7 on timing. The data_out bits are also connected to the output memory, and carry the data to be written.

The claim_mem output is used to make sure that the input memory where the wave data is stored cannot be accessed by the ARM while transmission is enabled, ensuring that the correct data is transmitted. As soon as transmission is started via the start bit, the claim_mem bit goes high. This claim_mem bit toggles the tri-state buffer on the FGPA/ARM bus module that controls the data bus, and sets the bus to output data to the ARM. This effectively makes the FPGA/ARM bus read-only, for the duration of the transfer.

The Rx_Done signal is used to indicate that a full pulse has been received. This is connected to UDP FIFO. Its purpose is to make sure that there is not an incomplete packet waiting for more data before transmission when the next pulse is received. When the Rx_Done flag is asserted, the final data in the output buffers is cleared.

The Wave_addr bus is connected to the input memory. This controls the address in the input memory from which the wave data is being read.

The final output is the pulse_count ouput. This output is connected to a 16bit counter inside the radar block and to the networking block on the outside. It counts the number of pulses sent and is used to append the pulse number to the data packets.

5.8.3 **Wishbone Memory Blocks**

Block Ram was selected as the primary storage for the wave data in the radar block. This was selected because of its availability and simplicity. Alternatively the DDR3 RAM could have been used to store the data, but there is currently no gateware for controlling the DDR3 and it adds additional complexity. The spiral model approach taken for the development of the project leads to the conclusion that block RAM should be used for the initial implementation, with the possibility of changing to DDR3 at a later stage.

In order to make sure that the Block Ram and DDR3 ram could be easily switched, it is important to have a connection bus between the radar block and the RAM that is standardised and compatible with both the DDR3 and the block RAM. As discussed in section 2.7 the interface chosen for this interconnection of blocks on Rhino is Wishbone. A simple wishbone wrapper was created to make the block RAM wishbone compatible.

Using the wishbone interface means that the upgrade from block RAM to DDR3 is trivial, and requires only a simple substitution of modules.

The block RAM used for this project is created by the Xilinx core generator. It can be customized to have the required read and write bit-widths, and the correct amount of storage

space. It is also possible to create dual port ram, which is used for the input memory, as there are different clocking speeds required on either side of the memory. The data coming in from the ARM is clocked by the clock generated by the GPMC; while data going out to the radar block is clocked by the 100MHz system clock.

5.9 [F9] Network Attached Storage

The Network Attached Storage for this project is implemented by a software program running on the control Pc. The data outputted from the radar block is sent out over the gigabit Ethernet to a waiting Pc. The Pc needs to know what to do with this information. In order to handle the storage of information on the Pc, an application is run to store the packets.

The application was designed to run on a windows based machine, and used win-socket to create a UDP socket. The data written to the pc is handled by this program, and stored in a file labelled by the data and time of the first incoming pulse.

The program was developed for windows because only available machine with gigabit networking at the time was running on Windows.

6 Results

This chapter documents the results of the tests carried out on the system, both throughout its development, and after completion of the system.

Tests are sorted according to the system functionality they are testing, in the same format in which they were described in the methodology.

The tests are only described briefly in this chapter, for more details on how the tests were conducted please see Section 3.11.

6.1 [F1]Programming FPGA

Programming Time Test

The goal of the programming time test was to determine the time taken to program the FPGA. The time taken to program the FPGA was monitored by comparing the timestamps at the beginning and ending of programming. The time taken to program the FPGA was 53s.

The FPGA programmer successfully programmed the FPGA from the ARM processor using a compiled binary file. Programming the FPGA only required the user to copy the programming file to the correct directory using the built in "cat" or "dd" commands in Linux. The programming was confirmed by the scrolling LEDs shown in Figure 6-1: Photo of FPGA LEDs during operation of "Rhino_Blinky" an LED scrolling gateware application.

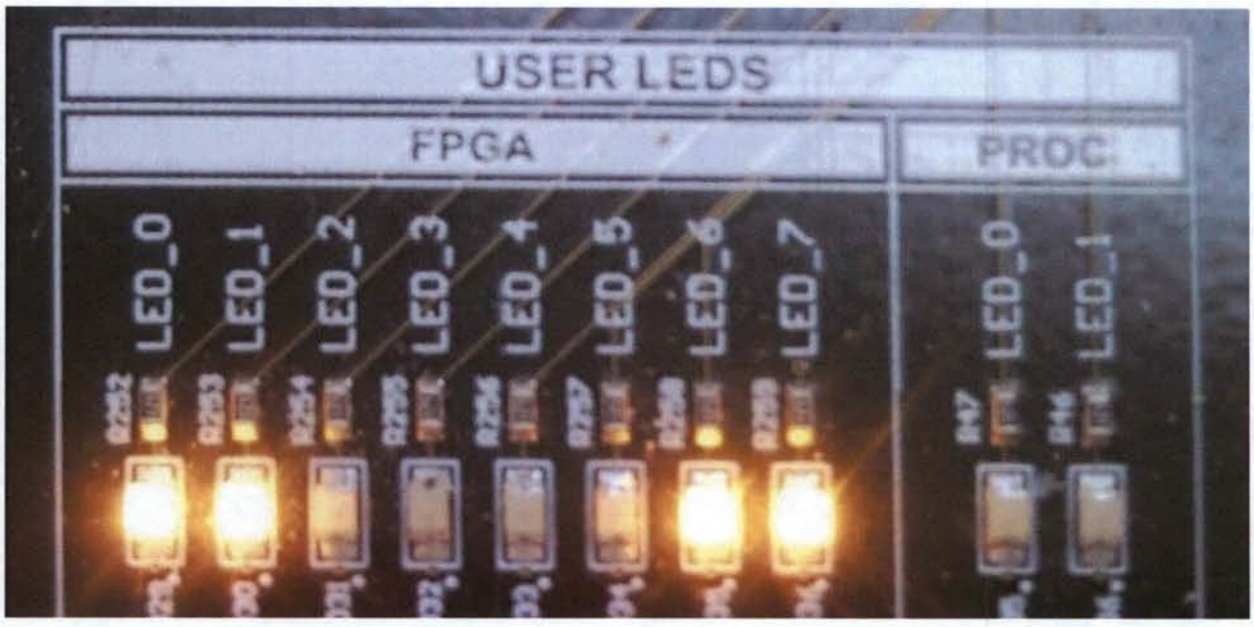

Figure 6-1: Photo of FPGA LEDs during operation of "Rhino_Blinky"

6.2 [F2] FPGA-ARM Data Bus

Transfer Integrity Test

The goal of the transfer integrity test was to confirm that the data was being correctly written to the FPGA by copying a block of data to the FPGA and then reading it back.
Running this test showed that all data written to the FPGA was transferred correctly, and all data read back matched the input data exactly.

Transfer Rate Test

The speed of the bus was tested by using and application repeatedly writing files to the FPGA, and recording the time taken to complete the transaction. The results of these tests can be seen in Table 6-1.

Table 6-1: Table showing data transfer results in transfer of data from ARM-FPGA

File Size (bytes)	No. of Iterations	Total Time Taken (s)	Data Transfer Rate (Mbits/s)
100	100000	1.57	50.96
100	1000000	16.24	49.26
1000	100000	15.26	52.42
1000	1000000	154.5	51.78
30000	10000	6.28	47.77
30000	100000	62.85	47.73
		Average:	49.99

This test used an application to copy the data from a file stored on the ARM's file system (which was stored on the control PC) to the FPGA. The time taken for the transfer shown in Table 6-1 above represents the total running time of the transfer, including copying the data into memory from the control PC over the network.

As can be seen in Table 6-1, while the data transfer rate varied based on the size of the input file, the average data transfer rate was in the region of 50Mbit/s.

6.3 [F3] Radar Block Parameters

Register Value Test

In order to confirm the correct values in the registers, each value was output on the boards LEDs. Each register was tested individually and all the registers were confirmed to be storing the correct values. In Figure 6-2 the LEDs showing the value 10010110 (150) for the pulse length register is shown.

Figure 6-2: LEDs indicating a value of 300 on the pulse length register.

6.4 [F4] Pulse Repetition

Pulse Repetition Interval Range

The Pulse Repetition range was tested through calculation. The clock speed combined with the register size used to store the PRI, can be used to perform calculations to determine the range of intervals that can be used.

The PRI counter used a 32bit counter with a 100MHz clock; the equations for calculating the maximum PRI interval are shown below.

$$Maximum\ Interval = \frac{Max\ PRI\ Register\ Value}{Clock\ Frequency}$$

$$= \frac{\mathbf{4\,294\,967\,295}}{100 \times 10^6}$$

$$= 4295s$$

From the calculations above, the maximum interval supported by the PRI counter is 4295s, which is well above the required 1s.

Pulse Repetition Interval Resolution

The resolution for the PRI was tested through calculation. The resolution of the PRI is equal to the inverse of the Clock Frequency.

$$Resolution = \frac{1}{Clock\ Frequency}$$
$$= \frac{1}{100 \times 10^6}$$
$$= 10ns$$

The actual resolution of the PRI of 10ns is well above the required spec of 1µs.

Pulse Repetition Interval Accuracy

The accuracy of the PRI can again be determined through calculation. The accuracy of the pulse repetition function is going to be based on the accuracy of the clock that is driving it, which is available in the clock's datasheet.
The PRI was driven by a 100MHz Clock, which has a period of 10ns. The jitter of the differential input clock used is 1ps [40], which after passing through the Xilinx clock core, comes out at 200ps. The output clock's jitter is less than 1% of the period, and therefore falls within specified limits.

Pulse Repetition Timing

This test aimed to show the system was operating according to the desired timing. An Oscilloscope was used to monitor signals which were routed to output pins for the purposes of this test. Two signals were routed to IO pins, the one shows when data was being transmitted, and the other shows when data was being received (A high value indicates data transmission). In Figure 6-3 two signals can be seen yellow signal represents data being received, while the green signal represents data being transmitted. The interval between the start of transmission of two pulses is shown here to be 500µs. The measured values matched the input PRIs for all tested values, showing the PRI was functioning correctly.

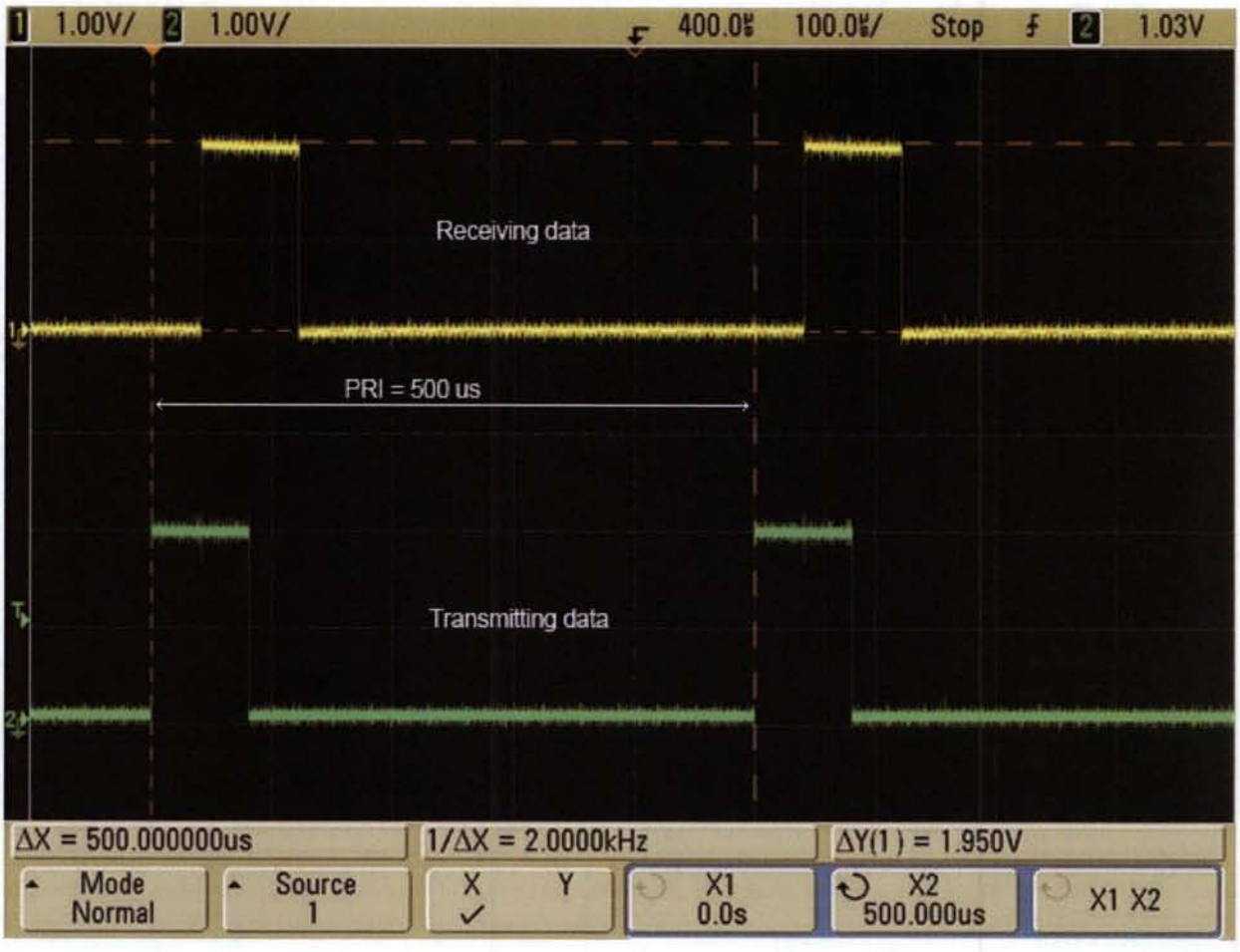

Figure 6-3: Scope output showing PRI of 500µs

6.5 [F5] Pulse Transmission

Pulse Data Transmission test

As described in section 3.11.5 the transmit functionality was confirmed by examining the data returned by the receive block. When compared to the input data, it showed to be an exact match indicating a fully functional system.

Pulse Transmission Speed.

The speed of transmission is tested through calculation, by multiplying the clock speed of the data transfer by the width of the bus.

$$\begin{aligned} Bus\ Speed &= Bus\ Width\ \times Clock\ Speed \\ &= 32bits\ \times 100Mhz \\ &= 400MB/s \end{aligned}$$

As expected the pulse transmission speed does not meet the ideal requirement of 25Gb/s, this is due to the fact that it was designed for use with DACs running at 200 mega samples per second. It does however meet the 400MB/s speed which was deemed acceptable.

6.6 [F6] Pulse Reception

Pulse Data Reception Test

Pulse data reception was confirmed by examining the output of gateware designed to write the received data to the network. The received data was compared to the input data and the two data sets were seen to be identical indicating full functionality.

Pulse Reception Speed.

The speed of transmission is tested through calculation, by multiplying the clock speed of the data transfer by the width of the bus.

$$\begin{aligned} Bus\ Speed &= Bus\ Width\ \times Clock\ Speed \\ &= 32bits\ \times 100Mhz \\ &= 400MB/s \end{aligned}$$

As expected does not meet the ideal requirement of 25Gb/s, because it was designed for use with ADCs that are built to handle a rate of 200 mega samples per second. The system was designed to be modified to allow faster ADCs if required. It does however meet the 400MB/s speed which was deemed acceptable.

6.7 [F7] Receive Delay

Receive Delay Interval Range

The receive delay range was also tested through calculation. The clock speed combined with the register size used to store the delay, allows calculations to be performed to determine the range of intervals that can be used.

The delay counter used a 32bit counter with a 200MHz clock; the equations for calculating the maximum delay interval are shown below.

$$Maximum\ Interval = \frac{Max\ PRI\ Register\ Value}{Clock\ Frequency}$$

$$= \frac{\mathbf{4\ 294\ 967\ 295}}{200 \times 10^6}$$

$$= 2148s$$

From the calculations above, the maximum interval supported by the delay counter is 2148s, which is well above the required 1s.

Receive Delay Resolution

The resolution for the receive delay was tested through calculation. The resolution of the delay is equal to the inverse of the Clock Frequency.

$$Resolution = \frac{1}{Clock\ Frequency}$$

$$= \frac{1}{200 \times 10^6}$$

$$= 5ns$$

The actual resolution of the receive delay is 5ns which exactly meets the specified requirement.

Receive Delay Accuracy

The accuracy of the receive delay was determined through calculation and was based on the accuracy of the clock that drives it.

The delay was driven by a 200MHz Clock, which has a period of 50ns.The jitter of the differential input clock used is 1ps [40], which after passing through the Xilinx clock core, comes out at 200ps. The output clock's jitter is less than 1% of the period, and therefore falls within specified limits.

Receive Delay Timing

This test aimed to show the system was operating according to the desired timing. An Oscilloscope was used to monitor signals which were routed to output pins for the purposes of this test. Two signals were routed to IO pins, the one shows when data was being transmitted, and the other shows when data was being received (A high value indicates data transmission).

In Figure 6-4 two signals can be seen yellow signal represents data being received, while the green signal represents data being transmitted. The delay between the beginning of data transmission and data reception is shown to be 40µs. The measured delays matched the input delays for all tested values.

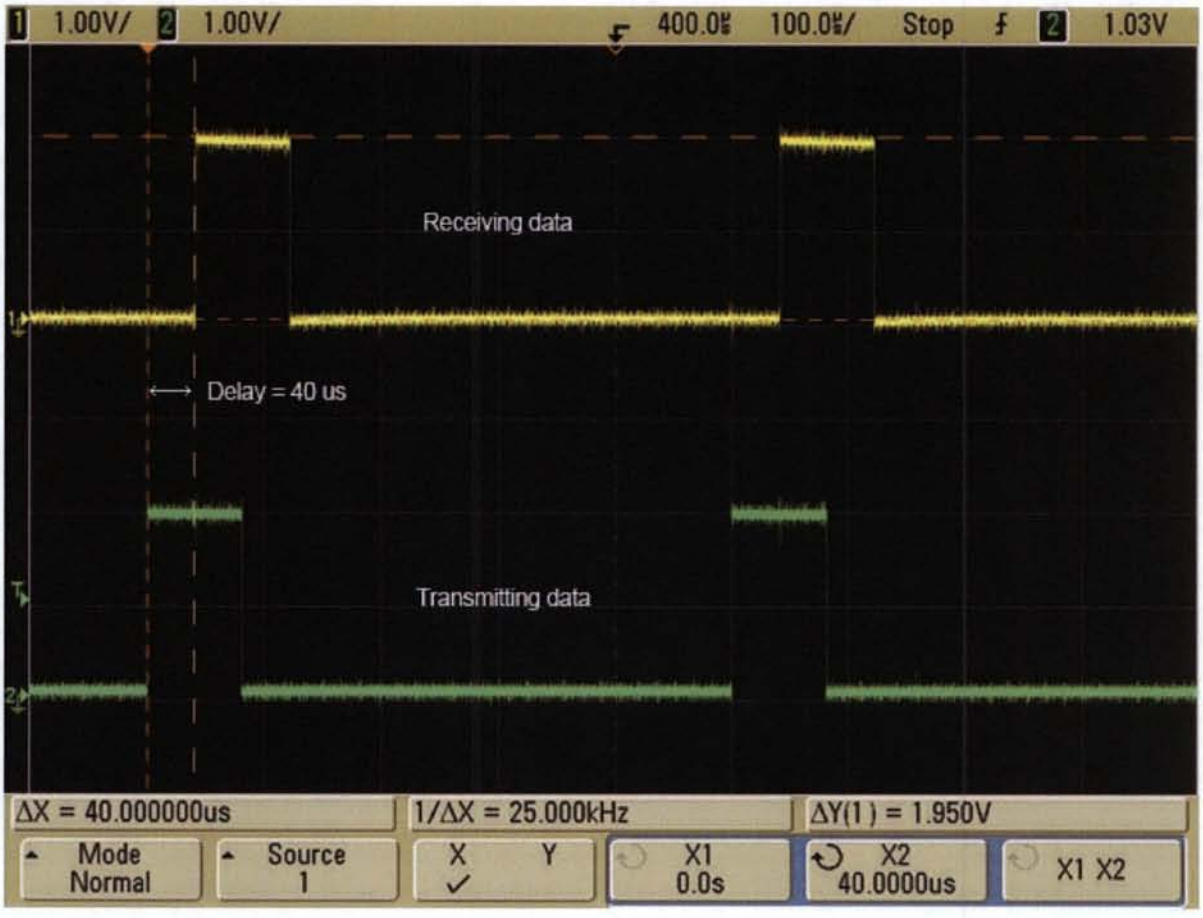

Figure 6-4: Scope output showing receive delay of 40µs

6.7.1 [F8] Gigabit Ethernet

Data Transfer Test

The gigabit Ethernet connection was tested by sending packets of known data over the network, and receiving them using Wireshark. In Figure 6-5 a screen capture of one of the packets received in Wireshark can be seen. The packet arrived complete, and with no errors.

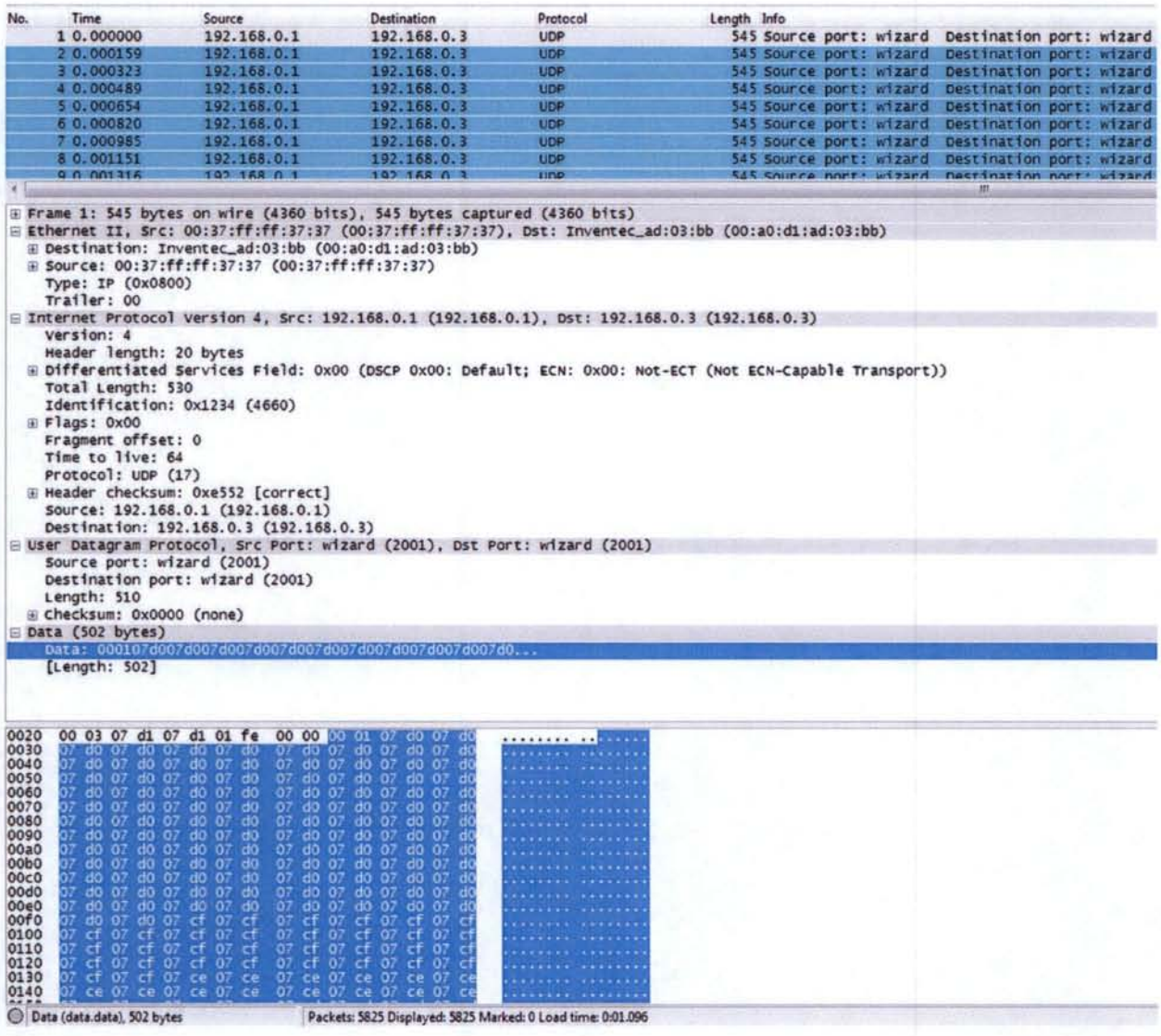

Figure 6-5: Screen Capture of Wireshark, showing a packet received.

In Figure 6-5 above the blue background behind the packets received at the very top of the image show that the packet has no errors in the frame (this does not check the data other than confirming the checksum). The source and destination ports and IPs are all as specified in the gateware.

The same result was found for all other packets sent in the data transfer, thus verifying the proper functionality of the Gigabit Ethernet data transfer.

Transfer Speed Test

The speed of the Gigabit Ethernet transfer is determined through examination of the packets in Wireshark. Every packet received has a timestamp, recording the exact time it arrived. In Figure 6-6 a screen capture is seen, showing the times at which packets arrived. Highlighted in blue the time between individual frames is 165µs, while in red the time taken to transmit all 50 frames is 8.093ms. The data payload for each packet is highlighted in green, and is 502 bytes.

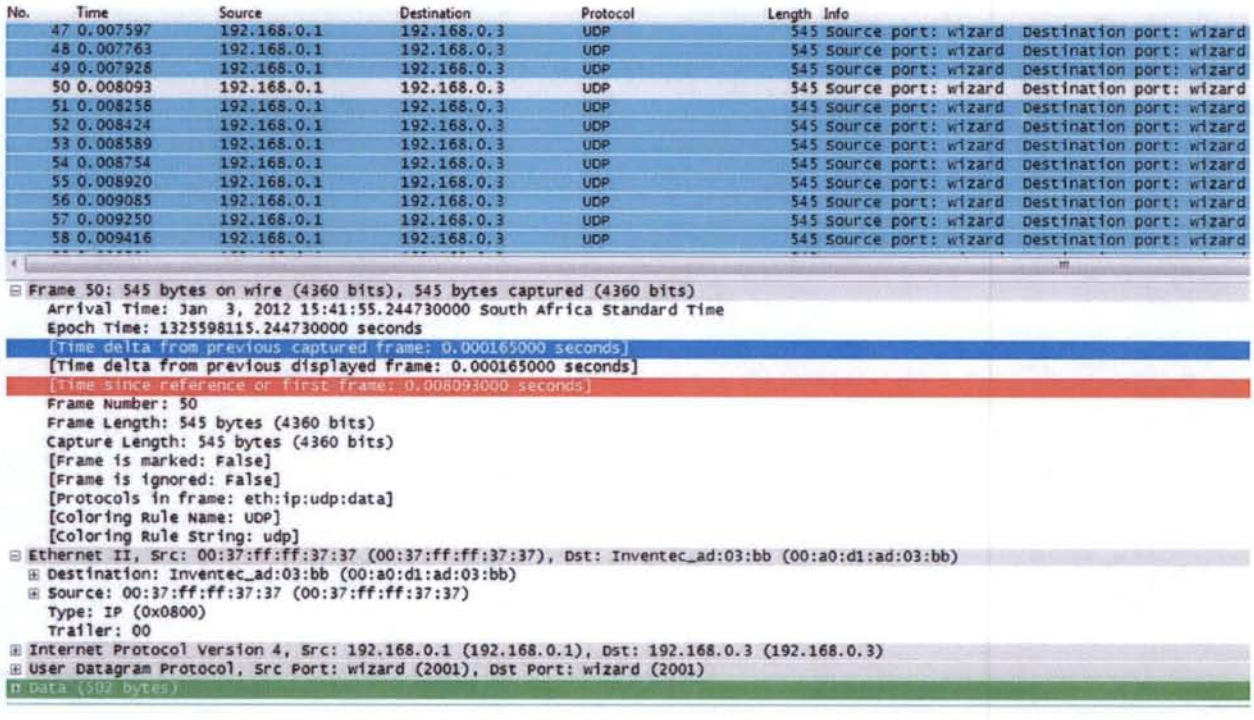

Figure 6-6: Screen capture of Wireshark output, showing time taken for packet transfer over Gigabit Ethernet

Transmitting 50 frames containing 502 bytes of data each in 8.093ms multiplies out to a data transfer rate of 24.8Mb/s, which is considerably lower than either the ideal or desired transfer rates.

6.7.2 [F9] Network Storage

Data Storage Test

The network storage was tested by comparing the input data file to the output data file. No data loss existed between input and output files, which implies that data is being stored at a

sufficiently fast rate thus avoiding data loss. Figure 6-7 shows a comparison of the input and output data waves.

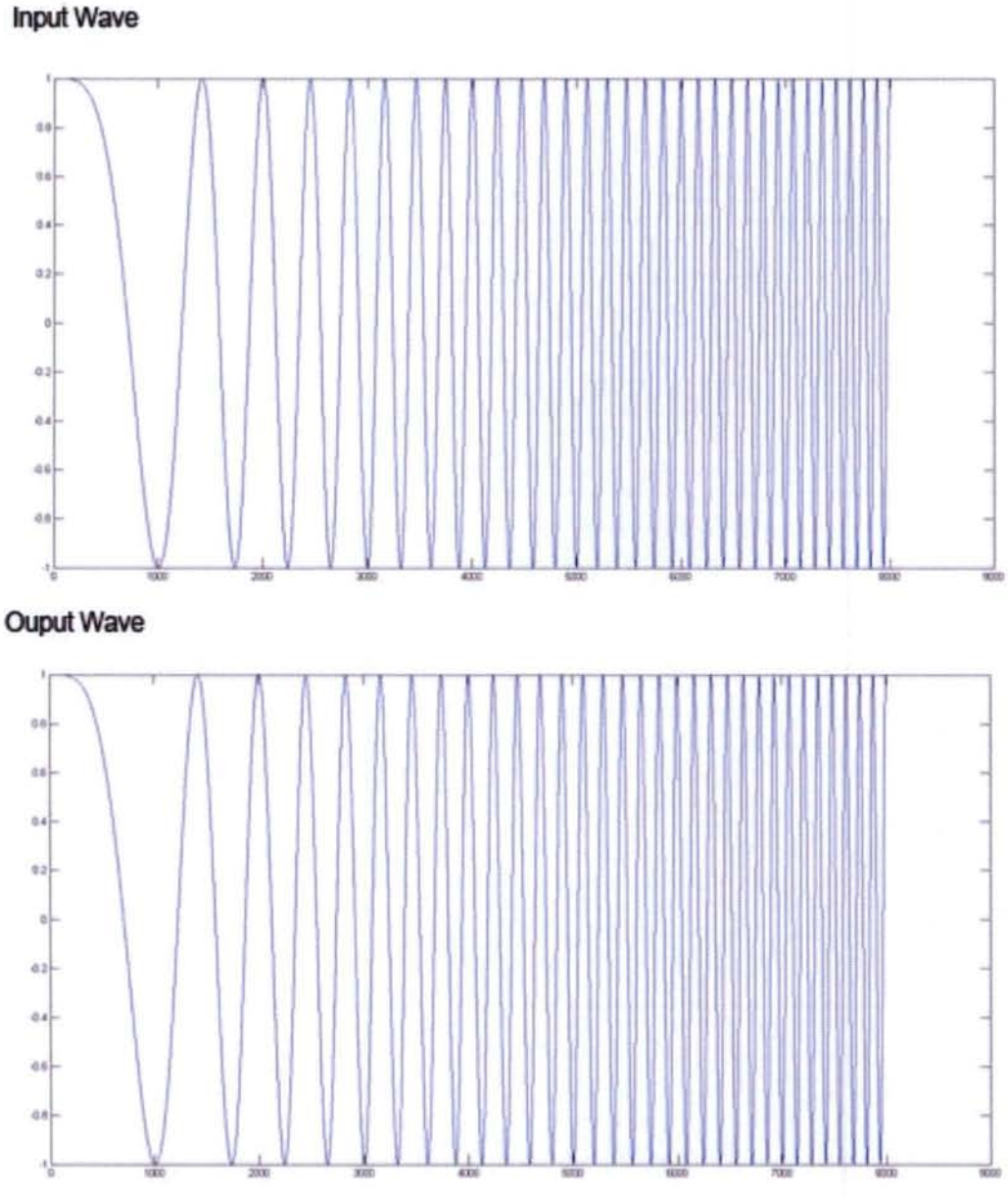

Figure 6-7: Comparison of Input and Output Waves.

6.7.3 **Other System Tests**

Full System Verification Test (F3, F8)

The purpose of this test was to confirm the systems operation as a complete system. The system was tested with operating parameters at the limits of the specifications, and the output data was examined.

Figure 6-8 shows the final packet of data containing the first output pulse for two different PRI values, 0x1025aa(10.58218ms) and 0x1025af(10.58218ms). In both the left and right image the pulse number is highlighted in purple, and the final bytes of the first pulse are highlighted in green. On the left, highlighted in red, the first samples of the second pulse can be seen at the end of the frame; while on the right these are not present and only padding zeros are present. This occurs when the second pulse is received before the first pulse has finished transmitting and can cause confusion in the output. It also means that data is being received faster than it can be output over the gigabit Ethernet.

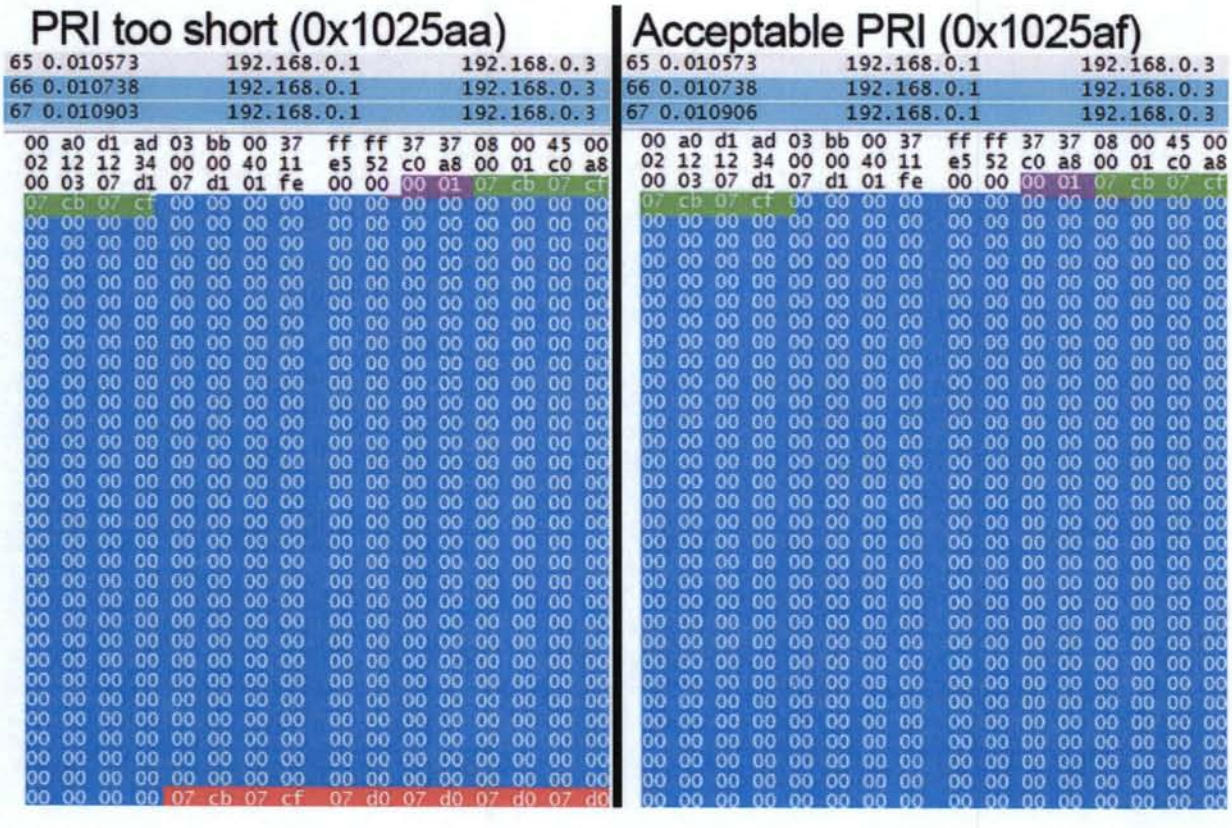

Figure 6-8: Comparison between output from acceptable and short PRI values.

This test confirms suspicions that the gigabit Ethernet output does not output data fast enough to support the full range of pulse repetition frequencies at the maximum pulse length.

The maximum PRF that can be used for any give pulse size is based on the total output data rate.

The shortest PRI that can be used with an input pulse of 32128 bytes is approximately 10.58ms; simple multiplication gives a maximum data output rate of 25Mb/s.

In order to illustrate this point, the test is repeated using wave sizes that are easily displayed graphically. Using an input wave 930 samples long, as seen in Figure 6-9, the test is repeated

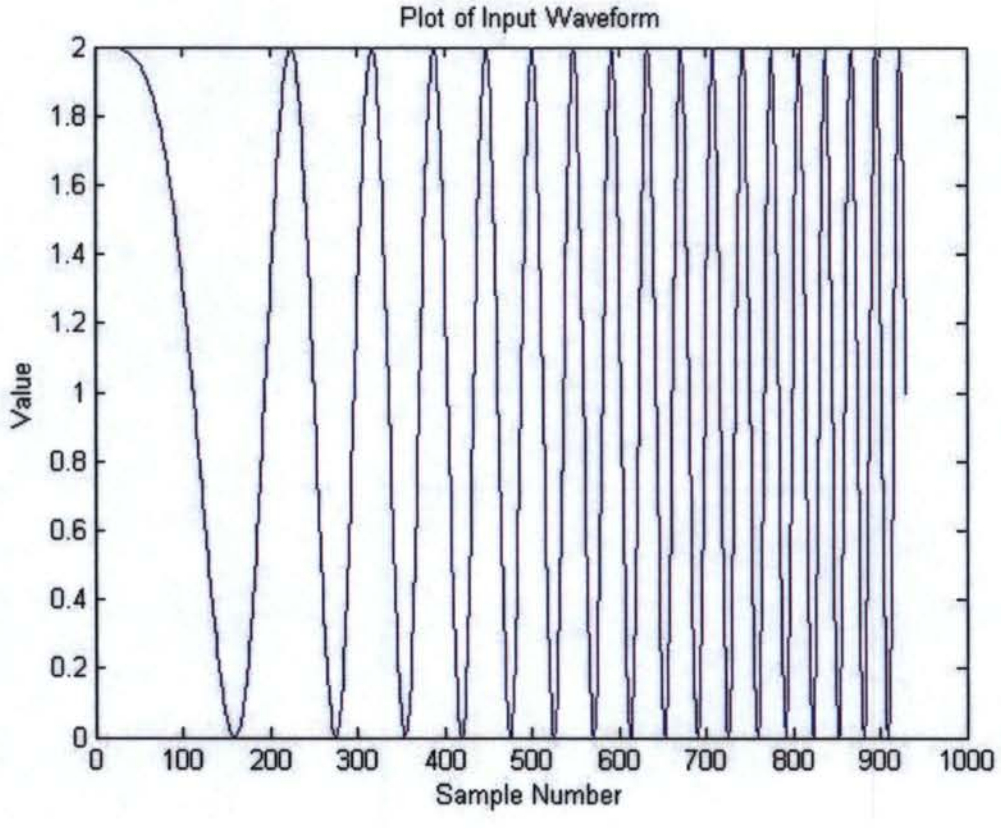

Figure 6-9: Input Chirp Comprising of 900 Samples

When the PRI is sufficiently long, the output for the first pulse contains just the waveform, and zeros which are added as padding to fill the full 502 byte packet, as seen in Figure 6-10.

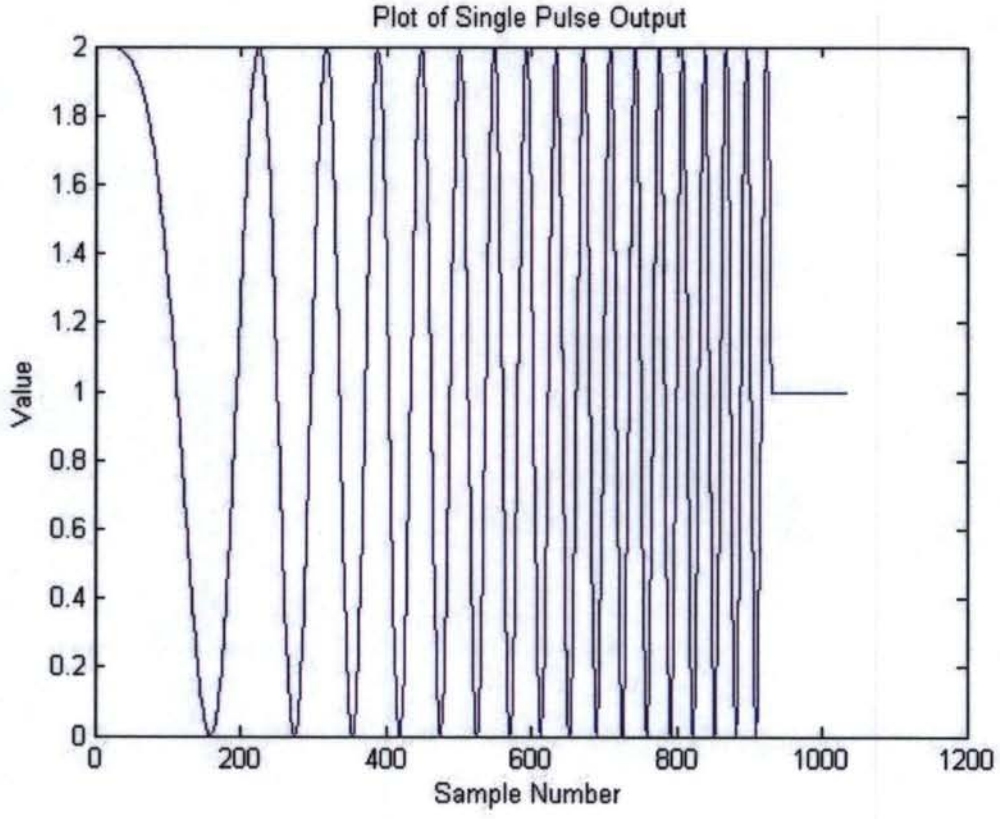

Figure 6-10: Output of Single Pulse with no Errors.

When the test is repeated with a PRI under 1.3ms (which gives a total data rate of above 25MB/s) the start of the second pulse is seen where the padding zeros should be; as shown in Figure 6-11. This behavior indicates that pulses are not being transmitted at a sufficient rate to clear the output buffer between each pulse. This form of operation causes ambiguity in the output, and will eventually cause the output buffer to become full, and data will be lost.

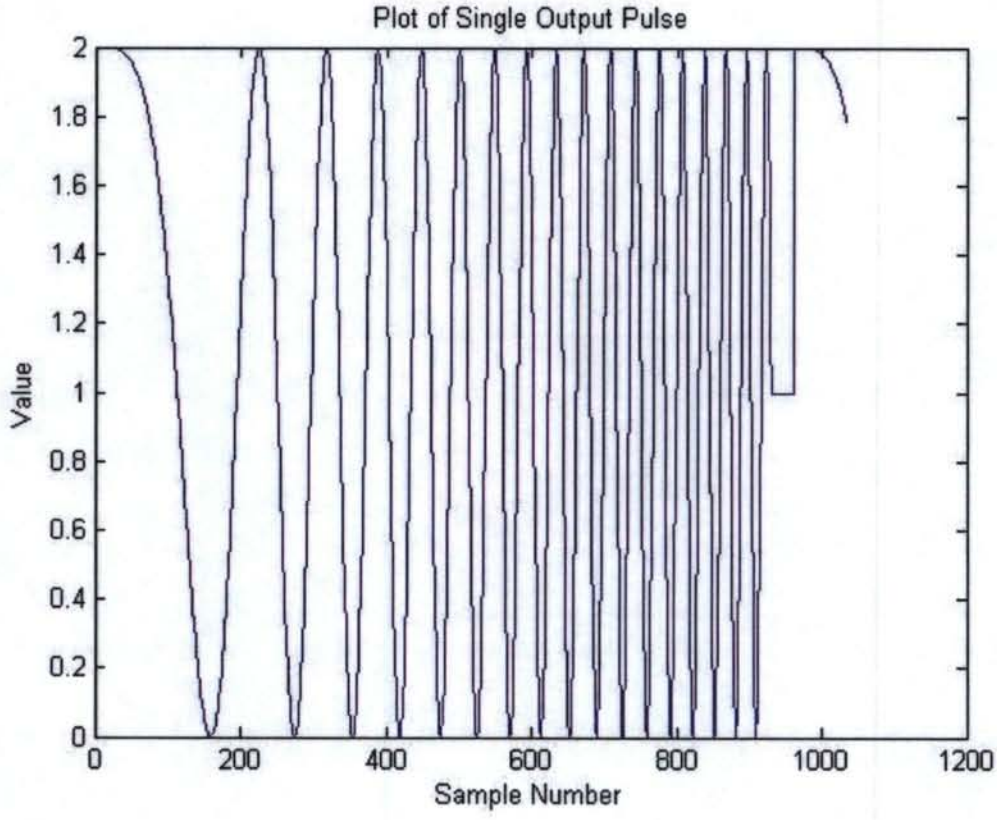

Figure 6-11: Output of Single Pulse with Second Pulse Overlapping.

6.7.3 **Other System Tests**

Full System Verification Test (F3, F8)

The purpose of this test was to confirm the systems operation as a complete system. The system was tested with operating parameters at the limits of the specifications, and the output data was examined.

Figure 6-8 shows the final packet of data containing the first output pulse for two different PRI values, 0x1025aa(10.58218ms) and 0x1025af(10.58218ms). In both the left and right image the pulse number is highlighted in purple, and the final bytes of the first pulse are highlighted in green. On the left, highlighted in red, the first samples of the second pulse can be seen at the end of the frame; while on the right these are not present and only padding zeros are present. This occurs when the second pulse is received before the first pulse has finished transmitting and can cause confusion in the output. It also means that data is being received faster than it can be output over the gigabit Ethernet.

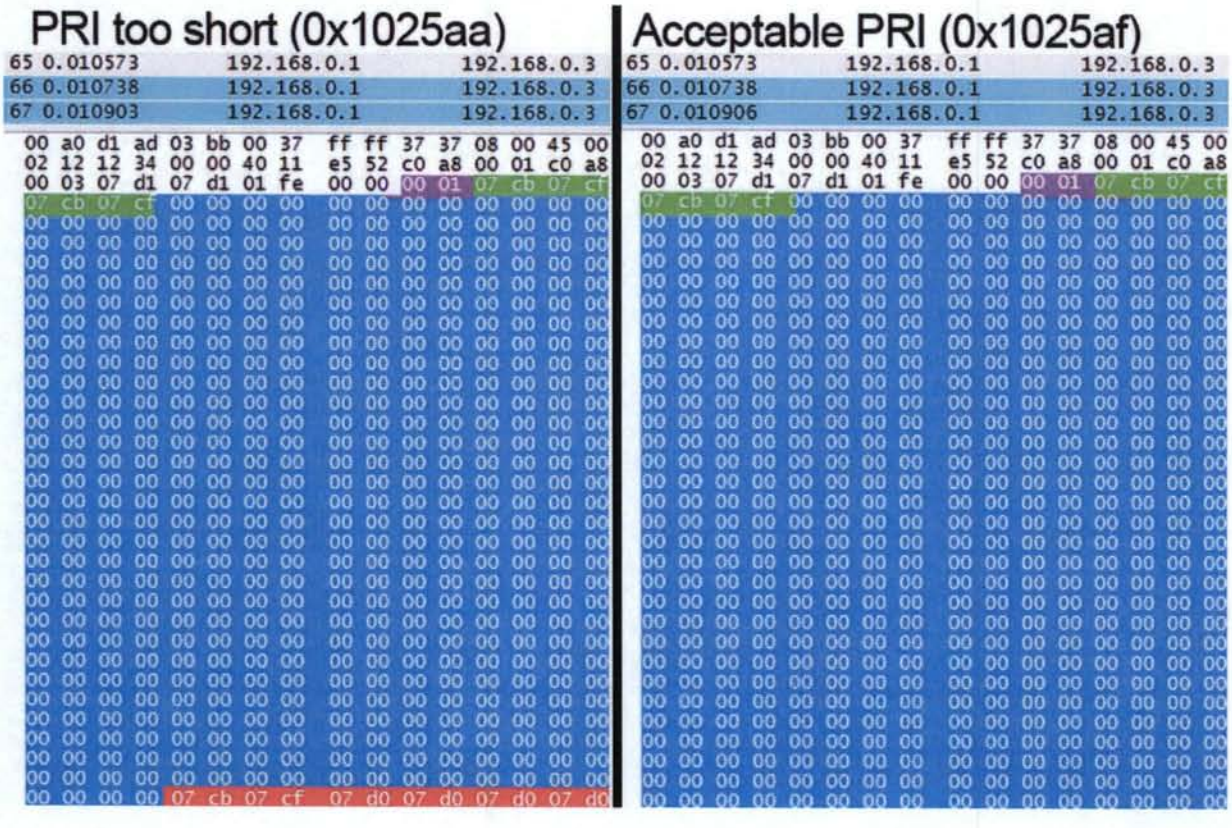

Figure 6-8: Comparison between output from acceptable and short PRI values.

This test confirms suspicions that the gigabit Ethernet output does not output data fast enough to support the full range of pulse repetition frequencies at the maximum pulse length.

The maximum PRF that can be used for any give pulse size is based on the total output data rate.

The shortest PRI that can be used with an input pulse of 32128 bytes is approximately 10.58ms; simple multiplication gives a maximum data output rate of 25Mb/s.

In order to illustrate this point, the test is repeated using wave sizes that are easily displayed graphically. Using an input wave 930 samples long, as seen in Figure 6-9, the test is repeated

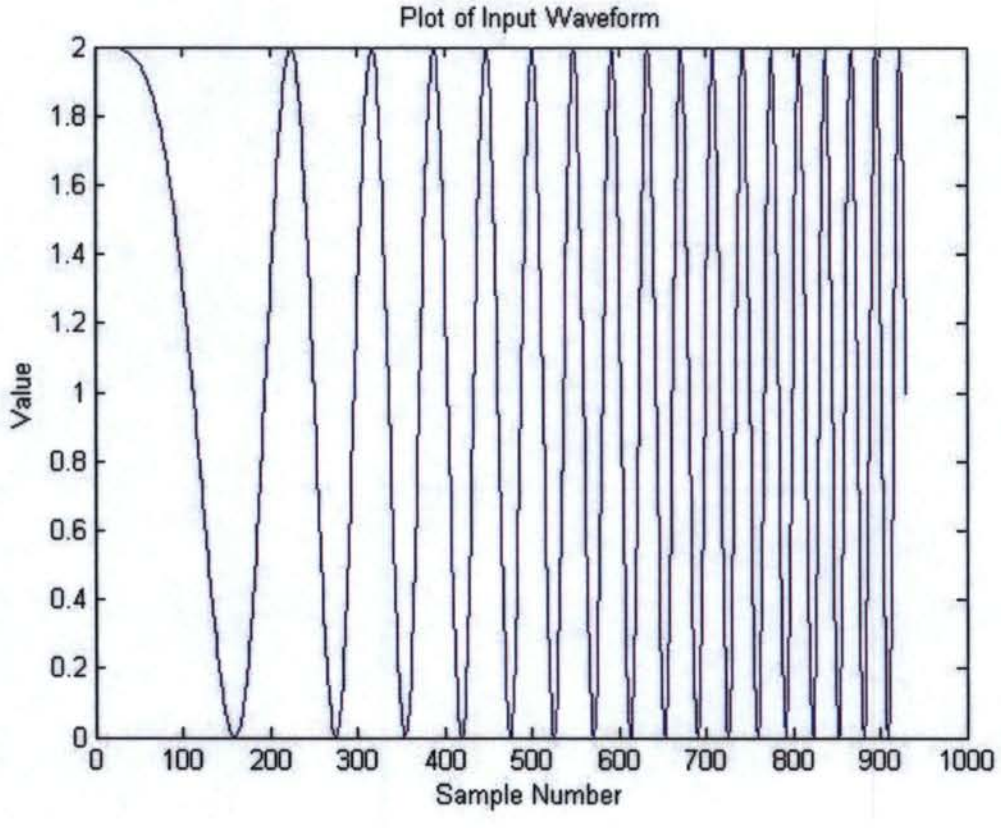

Figure 6-9: Input Chirp Comprising of 900 Samples

When the PRI is sufficiently long, the output for the first pulse contains just the waveform, and zeros which are added as padding to fill the full 502 byte packet, as seen in Figure 6-10.

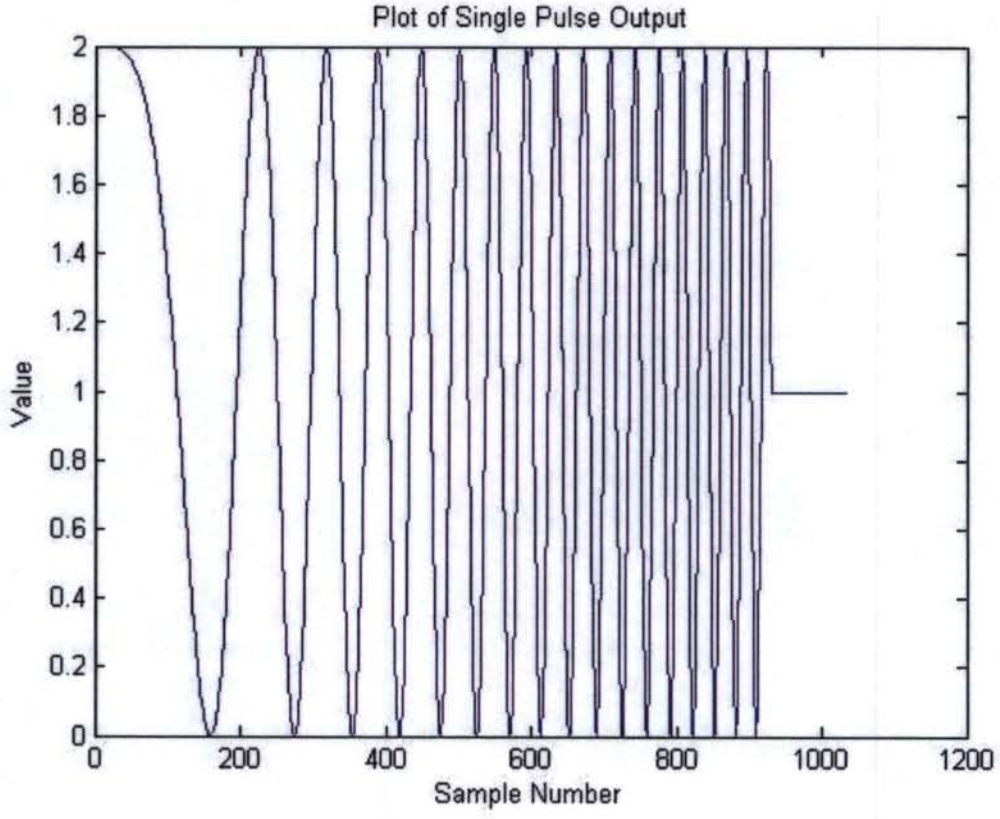

Figure 6-10: Output of Single Pulse with no Errors.

When the test is repeated with a PRI under 1.3ms (which gives a total data rate of above 25MB/s) the start of the second pulse is seen where the padding zeros should be; as shown in Figure 6-11. This behavior indicates that pulses are not being transmitted at a sufficient rate to clear the output buffer between each pulse. This form of operation causes ambiguity in the output, and will eventually cause the output buffer to become full, and data will be lost.

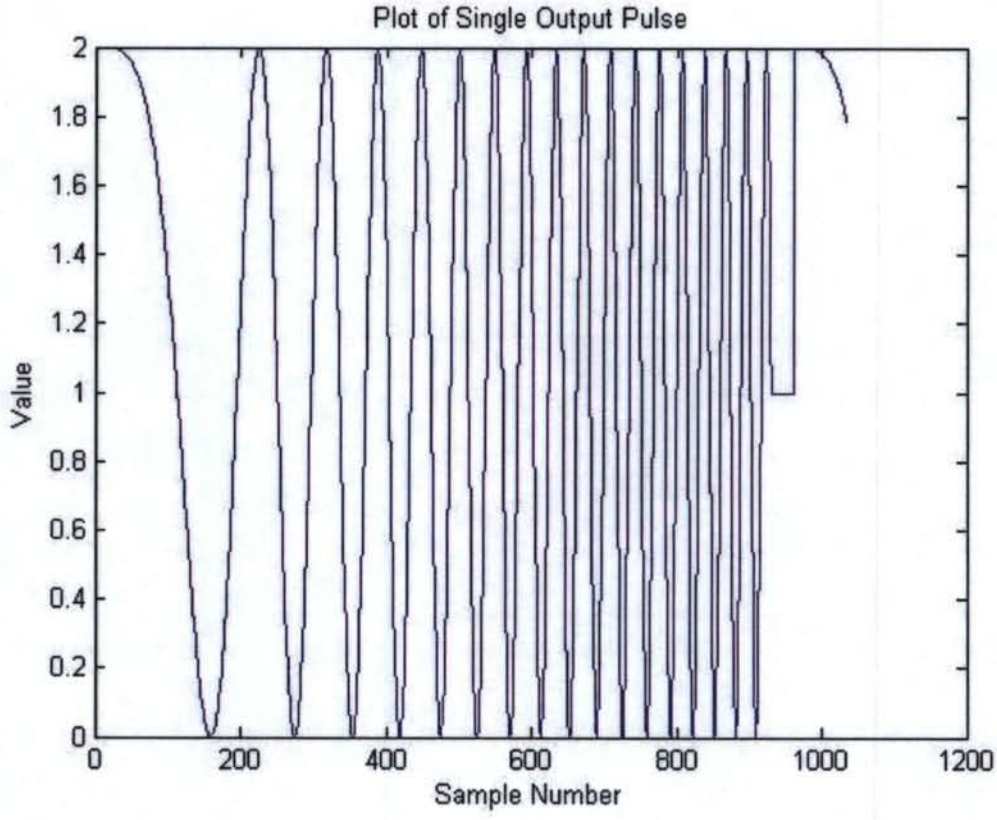

Figure 6-11: Output of Single Pulse with Second Pulse Overlapping.

7 Conclusions and Future Work

This chapter covers the conclusions that are drawn regarding the success of the project. The testing results of the final system are discussed and compare to the system specifications to determine whether or not the requirements have been met.
Each of the original objectives given is discussed in relation to the final output system of the project.

In the final section of this chapter recommendations are made for future work.

7.1 System Functionality

7.1.1 [F1]Program FPGA

The FPGA programming function successfully programmed the FPGA in 53 seconds. While this is considerably longer than the theoretical minimum programming time of 0.7s, it is well under the required time of 5 minutes.
The discrepancy in the actual time taken to program the FGPA compared to the theoretical time the SPI interface should take to transfer the required amount of data suggests that there is a large overhead in transferring data to the SPI. It is likely that this overhead would dramatically decrease if some form of DMA (Direct Memory Access) was implemented.

7.1.2 [F2] FPGA-ARM Data Bus

The FPGA-ARM Data bus was required to transfer data from the ARM processor to the FPGA at a rate exceeding 304Kbps in order to transfer configuration parameters in under one second. The measured data transfer rate was approximately 50Mbps, which greatly exceeds the required speeds and was more than sufficient for the purposes of this project. The

performance requirement for the data bus for this was set for a human using the system; if the system were automated, the increased bus speed may prove useful.

The relatively high transfer speed on the data bus is unnecessary for this project, but because the bus was designed to be reusable it was designed to be as fast as possible.

7.1.3 **[F3] Radar Block Parameters**

The Radar Block Parameter function was required to allow users to modify the parameters of the radar block. The testing on this block showed that registers the user attempted to update were successfully storing the correct values. These register values were used in the radar block and allowed the user full control of the parameters, including starting and stopping the operation of the radar block.

7.1.4 **[F4] Pulse Repetition**

The Pulse Repetition function was required to create a PRI in the range of 0-1s, with a resolution of 1µs to an accuracy of 1%.

The PRI register was tested and shown to handle values between 0s and 4295s. This range was well over the required spec due to the fact that the data bus writes data in chunks of 16 bits; a 16 bit register was insufficient, so a 32 bit register was used.

The PRI was shown to have a resolution of 10ns and was accurate to within 200ps. Both of these specifications were well under the specified requirements due to the speed and accuracy of the system clock.

The PRI function was tested using an oscilloscope to confirm its functionality; the scope showed that the system was outputting pulses at the specified frequency of operation.

7.1.5 [F5] Pulse Transmission

When tested the pulse transmission block was shown to accurately output the desired waveform. The speed of data transmission was calculated to be 400MB/s; this value does not reach the transfer rate of 25Gb/s which is required to handle operation and maximum PRF and pulse length. The fact that pulse transmission does not reach this data rate is not a major issue for this system, as this transfer rate is well above other hardware bottlenecks caused by DACs/ADCs and the gigabit Ethernet output. The speed of 400MB/s is sufficient to supply samples to a 16bit ADC running at 200 mega samples per second.

The transfer rate can be easily increased by either increasing the clock speed of the transmit function, or increasing the number of bits sent per clock cycle to suite the DAC being used.

7.1.6 [F6] Pulse Reception

Pulse Reception was tested similarly to the Pulse transmission and was calculated to have the same data transfer rate of 400MB/s. As with Pulse Transmission this value does not reach the transfer rate of 25Gb/s which is required to handle operation and maximum PRF and pulse length. For the same reasoning as the pulse transmission the slower clock speed of 400MB/s did not cause any negative effect on the operation of the system.

As with the pulse transmission function the transfer rate can be easily increased by either increasing the clock speed, or increasing the number of bits sent per clock cycle to suite the ADC being used.

7.1.7 [F7] Receive Delay

Performance specifications required the receive delay to match the rage of the PRI (0-1s), with a resolution of 5ns to an accuracy of 1%.

The delay register was designed to be the same size as the PRI register but with double the clock speed, and thus the range of the delay was shown to be between $0 - 2148$s.

The 200MHz clock of the delay function allowed a resolution of 5ns and was accurate to within 200ps. Both of these specifications were well under the specified requirements due to the speed and accuracy of the system clock.

The delay function was tested using an oscilloscope to confirm its functionality; the scope showed that the system was delaying the reception of pulses by the correct time period.

7.1.8 [F8] Gigabit Ethernet

The Gigabit Ethernet block successfully allowed data to be output from the system reliably with no errors occurring in transmission as was shown in Figure 6-5: Screen Capture of Wireshark, showing a packet received..

The speed of the Gigabit Ethernet was shown to be 24.8MB/s, which is well below the 25Gb/s that would be required to run the system at maximum PRF and pulse length. While the 25Gb/s requirement is well above the theoretical limit of the gigabit Ethernet, the goal of 400Mb/s transmission was a reasonable goal, but was also not met.
The MAC core used did not allow data to be transferred at a sufficient rate, which seemed to cause a bottleneck in data output.

The gigabit Ethernet's slow transmission speed severely hampered the functionality of the system, as the range of PRI and pulse length values that can be used is limited by the output transfer speed.

7.1.9 [F9] Network Storage

In Figure 6-7: Comparison of Input and Output Waves., the output data file stored by the network storage program was shown to match the input file. This showed that the storage program was capable of storing incoming data from the radar block sufficiently quickly.

7.1.10 Radar System Conclusions

When tested as a full system the Radar block was shown to function correctly only when the total data rate through the system was low enough to be handled by the gigabit Ethernet. The gigabit Ethernet proved to be a serious bottleneck in the system, limiting the PRI to a minimum of 10.58 ms when running with the maximum pulse length, and forcing the product of the PRF and pulse length to be under 25Mb/s.

7.2 Future work

The system would benefit greatly from the addition of a completed BORPH operating system, which was not available at the start of this project. A port of the Borph operating system would include the programming and data busses, and allow them to be more closely integrated with the kernel. The use of DMA for these communication interfaces would increase their performance

There are aspects of the radar block's functionality which still need to be worked on. The radar still requires ADC and DAC cards in order to output the data in a form transmittable by antennas and convert received data to a form readable by the FGPA. In order to add these cards to the system an FMC block is required to interface with the ADCs/DACs. This FMC block

would replace the temporary FMC block used in this project. The block was designed to be simple to add Digital Signal Processing (DSP) blocks to, additional DSP blocks for use within the radar block would be beneficial.

The speed of the gigabit Ethernet requires improvement, and the system would benefit greatly from increased output speed. Replacing the Mac core used with one capable of better output rates may help improve the performance of the gigabit Ethernet.

7.3 Final conclusions and Recommendations

The system performed well as a pulsed radar system, and provides a good stepping stone for future users to be able to develop applications on Rhino. The speed of the Gigabit Ethernet is the only real bottleneck in the system that needs to be overcome; once this is improved the system will be able to function properly for a much larger range on parameters. Once the system is properly integrated with Borph it will be even easier to learn and use. All of the blocks created can be reused in other applications, and the tools are fairly easy to use.

8 References

[1] Rhino Wiki. [Online]. ://www.ohwr.org/projects/rhino-hardware-01/wiki

[2] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, 2001, pp. 642-649.

[3] B. W. Boehm, "A spiral model of software development and enhancement," *Computer*, vol. 21, pp. 61-72, 1988.

[4] K. Compton, S. Hauck, and Katherine Compton, An Introduction to Reconfigurable Computing, 2000.

[5] Nathan Clark, Hongtao Zhong, and Scott Mahlke, "Processor Acceleration Through Automated Instruction Set Customization," in *Proceedings of the 36th annual IEEE/ACM International Symposium on*, 2003, p. 129.

[6] Andre DeHon and John Wawrzynek, "Reconfigurable computing: what, why, and implications for design automation," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, 1999, pp. 610-615.

[7] BerkeleyDesignTechnologyInc., BDTI DSP Dictionary, 2011.

[8] Roger Bryner Karen, "Comparing and Contrasting FPGA and Microprocessor System Design and Development," *White Paper:213*, vol. v1.1, p. 213, 2004.

[9] R.L. and Dudley, J. and Sadler, D. Walke, "An FPGA based digital radar receiver for soft radar," in *Signals, Systems and Computers, 2000. Conference Record of the Thirty-Fourth Asilomar Conference on*, 2000, pp. 73-77.

[10] (Bee2Wiki2006) Bee2OperatingSystem Wiki. [Online]. http://bee2.eecs.berkeley.edu/wiki/Bee2OperatingSystem.html#head-acfe74f819c0eec6dad90b9bbbc17b5897bff37f

[11] Hoyden Kwok-Hay So and R.W. Brodersen, "Improving Usability of FPGA-Based Reconfigurable Computers Through Operating System Support," in *Conference on Field Programmable Logic and Applications, 2006. FPL '06. International*, 2006, pp. 1-6.

[12] H So, "BORPH: An Operating System for FPGA-Based Reconfigurable Computers,"

UNIVERSITY OF CALIFORNIA, BERKELEY, 2007.

[13] B Hamilton and H So, "BORPH: Operating system support on the NetFPGA platform," in *In Proceedings of the 2nd North American NetFPGA Developers Workshop 2010*, 2010.

[14] H. So and R. Brodersen, "Runtime Filesystem Support for Reconfigurable FPGA Hardware Processes in BORPH," in *International Symposium on Field-Programmable Custom Computing Machines, 2008. FCCM '08. 16th*, 2008, pp. 285-286.

[15] Hayden Kwok-Hay So and Robert Brodersen, "A unified hardware/software runtime environment for FPGA-based reconfigurable computers using BORPH," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 2, pp. 14:1--14:28, 2008.

[16] Simon Scott, "RHINO: RECONFIGURABLE HARDWARE INTERFACE FOR COMPUTATION AND RADIO," 2011.

[17] Texas_Instruments_Incorporated, AM3517 Technical Reference Manual, 2009.

[18] FPGA Mezzanine Card (FMC) standard, 2008, Approved in 2008, revised in 2010.

[19] Roach Wiki. [Online]. https://casper.berkeley.edu/wiki/ROACH#Specifications

[20] [Online]. http://www.digicom.org/special-products/roach-board.html

[21] USRP N200 Series. [Online]. http://www.ettus.com/downloads/ettus_ds_usrp_n200series_v3.pdf

[22] Merrill I. Skolnik, *Radar handbook.*: McGraw-Hill, 1970.

[23] Skolnik, *Introduction to radar systems.*: New York, McGraw Hill Book Co., 1980.

[24] Rudolf Usselmann, "OpenCores SoC Bus Review,".

[25] Stojcev Mile Mitic, "An overview of on-chip buses," *Facta universitatis - series: Electronics and Energetics*, vol. 19, pp. 405-428, 2006.

[26] [Online]. http://cdn.opencores.org/downloads/wbspec_b4.pdf

[27] [Online]. http://www.altera.com/literature/manual/mnl_avalon_spec.pdf

[28] [Online]. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php

[29] [Online]. https://www-01.ibm.com/chips/techlib/techlib.nsf/products/CoreConnect_Bus_Architecture

[30] OpenCores. (2011) Open Cores. [Online]. http://opencores.org/opencores,wishbone

[31] Peterson, *WISHBONE System-on-Chip (SoC)Interconnection Architecturefor Portable IP Cores.*: OpenCores, 2010.

[32] Altera_Corporation, "Avalon Interface Specifications,".

[33] Richard Wilson, "ARM gives details of AMBA bus for embedding in FPGAs," 2010.

[34] Tieren Pelgrims¸Driessen, "Embedded Systeemontwerp op basis van,".

[35] IBM, "IBM CoreConnect bus cores,".

[36] N. Alachiotis, S.A. Berger, and A. Stamatakis, "Efficient PC-FPGA Communication over Gigabit Ethernet," *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference*, pp. 1727-1734, 2010.

[37] Soft Panorama. [Online]. http://www.softpanorama.org/Net/tcp_protocol_layers.shtml

[38] University of Southern, RFC: 793, TRANSMISSION CONTROL PROTOCOL, 1981.

[39] Xilinx, Spartan-6 FPGA Configuration User Guide.

[40] EEHerald. [Online]. http://www.eeherald.com/images/rs232-3.jpg

[41] Crystek_Crystals, "CCPD-033 LVPECL Datasheet,".

[42] Christoph Steiger, Herbert Walder, and Marco Platzner, "Operating Systems for Reconfigurable Embedded Platforms: Online Scheduling of Real-Time Tasks," *IEEE Transactions on Computers*, vol. 53, pp. 1393-1407, 2004.

[43] A. Rudra, ""FPGA-based Applications for Software Radio"," *RF Design*, vol. Signal Processing, pp. 24-35, 2004.

[44] John W. Lockwood et al., "NetFPGA--An Open Platform for Gigabit-Rate Network Switching and Routing," *Microelectronics Systems Education, IEEE International Conference on/Multimedia Software Engineering, International Symposium on*, vol. 0, pp. 160-161, 2007.

[45] E.A. Lee and D.G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, pp. 1235-1245, 1987.

[46] S. Hauck and A. DeHon, *Reconfigurable computing: the theory and practice of FPGA-based computation.*: Morgan Kaufmann, 2008.

[47] Ben Cope, Peter Y.K. Cheung, Wayne Luk, and Lee Howes, "Performance Comparison of Graphics Processors to Reconfigurable Logic: A Case Study," *IEEE Transactions on Computers*, vol. 59, pp. 433-448, 2010.

[48] Katherine Compton and Scott Hauck, "Reconfigurable computing: a survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171-210, 2002.

[49] Shuai Che, Jie Li, J.W. Sheaffer, K. Skadron, and J. Lach, "Accelerating Compute-Intensive Applications with GPUs and FPGAs," , 2008, pp. 101-107.

[50] J. Axelson, *Embedded ethernet and internet complete: designing and programming small devices for networking.*: Lakeview Research, 2003.

[51] Krste Asanovic et al., "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, pp. 56-67, 2009.

[52] K. Asanovic et al., "The parallel computing laboratory at UC Berkeley: A research agenda based on the berkeley view," University of California at Berkeley, Electrical Engineering and Computer Science Department, 2008.

[53] Jonathan Corbet, Alessandro and Greg Kroah-Hartman, *Linux Device Drivers, 3rd Edition.*: O'Reilly, 2005.

[54] Bee2OperatingSystem Wiki, Bee2Wiki2006.

[55] M. Hatamian et al., "Design considerations for gigabit Ethernet 1000Base-T twisted pair transceivers," pp. 335-342, 1998.

[56] F.L. Herrmann, G. Perin, J.P.J. de Freitas, R. Bertagnolli, and J.B. dos Santos, "A Gigabit UDP/IP network stack in FPGA," pp. 836-839, 2009.

[57] Xu Xing, Chen Zezong, Jiang Jing, and Ke Hengyu, "Porting from Wishbone Bus to Avalon Bus in SoC Design," pp. 1-862 -1-865, 2007.

[58] W.H. Ho and T.M. Pinkston, "A design methodology for efficient application-specific on-chip interconnects," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 17, pp. 174-190, 2006.

[59] Simon Winberg, Alan Langman, and Simon Scott, "The RHINO platform: charging towards innovation and skills development in software defined radio," pp. 334-337, 2011.

[60] [Online]. http://www.ettus.com/images/usrp_n210.JPG

[61] M. Inggs, G. Inggs, A. Langman, and S. Scott, "Growing horns: Applying the Rhino software defined radio system to radar," pp. 951-955, 2011.

[62] Analouge Devices. AD9777, Interpolating Dual TxDAC+®. [Online]. http://www.analog.com/static/imported-files/data_sheets/AD9777.pdf

[63] Linear Technology. LTC2284, Low power 3v ADC. [Online]. http://cds.linear.com/docs/Datasheet/2284fa.pdf

[64] Clifford E. Cummings and Sunburst Design Inc, "Nonblocking Assignments in Verilog Synthesis, Coding Styles That Kill," in *SNUG (Synopsys Users Group) 2000 User Papers, section-MC1 (1 st paper)*, 2000.

[65] H.K.-H. So and R. Brodersen, "File system access from reconfigurable FPGA hardware processes in BORPH," in *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on}*, 2008, pp. 567-570.

[66] G Wigley and D Kearney, "Research Issues in Operating Systems for Reconfigurable Computing," in *In Proceedings of the International Conference on Engineering of Reconfigurable System and Algorithms(ERSA)*, 2002, pp. 10-16.