University of Windsor

Scholarship at UWindsor

Electronic Theses and Dissertations

Theses, Dissertations, and Major Papers

2009

TCP/IP Control Server for a Multi-Drop Test Bench Network

Christopher Rennick
University of Windsor

Follow this and additional works at: https://scholar.uwindsor.ca/etd

Recommended Citation

Rennick, Christopher, "TCP/IP Control Server for a Multi-Drop Test Bench Network" (2009). *Electronic Theses and Dissertations*. 8203.
https://scholar.uwindsor.ca/etd/8203

This online database contains the full-text of PhD dissertations and Masters' theses of University of Windsor students from 1954 forward. These documents are made available for personal study and research purposes only, in accordance with the Canadian Copyright Act and the Creative Commons license—CC BY-NC-ND (Attribution, Non-Commercial, No Derivative Works). Under this license, works must always be attributed to the copyright holder (original author), cannot be used for any commercial purposes, and may not be altered. Any other use would require the permission of the copyright holder. Students may inquire about withdrawing their dissertation and/or thesis from this database. For additional inquiries, please contact the repository administrator via email (scholarship@uwindsor.ca) or by telephone at 519-253-3000ext. 3208.

TCP/IP Control Server for a Multi-Drop Test Bench Network

By
Christopher Rennick

A Thesis
Submitted to the Faculty of Graduate Studies
Through the Department of Electrical and Computer Engineering
In Partial Fulfillment of the Requirements for
The Degree of Master of Applied Science at the
University of Windsor

Windsor, Ontario, Canada
2009

Library and Archives Canada

Published Heritage Branch

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque et Archives Canada

Direction du Patrimoine de l'édition

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence
ISBN: 978-0-494-70580-3
Our file Notre référence
ISBN: 978-0-494-70580-3

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

© 2009 Christopher Rennick
All Rights Reserved. No part of this document may be reproduced, stored, or otherwise retained in a retrieval system or transmitted in any form, on any medium by any means without prior written permission of the author.

Co-Authorship Declaration

I hereby declare that this thesis incorporates material that is result of joint research, as follows:

This thesis also incorporates the outcome of a joint research undertaken in collaboration with Stephen Fox under the supervision of Dr. Roberto Muscedere. The details of the collaboration is covered in Chapter 2 of the thesis. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that with the above qualification, this thesis, and the research to which it refers is the product of my own work.

Abstract

This thesis describes the design, construction and verification process in full for the test server portion of the second generation of an automated testing network. The system was built for, and with, AMD/ATI of Markham, Ontario and will be used to test large batches of their graphics processing units (GPU's). The final test system has the capability to simultaneously test and control several parameters on a large number of test nodes.

The TCP/IP Control Server for a Multi-Drop Test Bench Network was designed to test and control a network of 256 test nodes over an RS-485 network. The contents of this thesis will describe the test server hardware in full, while the test nodes are described in Stephen Fox's thesis. The test server consists of an Ethernet-enable MCU, an Altera Cyclone II FPGA and a custom RS-485 transceiver board used to communicate with the test nodes.

To my family and friends, thanks for being there and for providing distractions when I needed them.

To Sandra, thanks for ten years of your love and support.

Acknowledgements

I would like to sincerely thank Dr. Muscedere for his invaluable assistance and guidance on this project, and for providing me the opportunity to work on such an interesting project with such a reputable industrial partner.

To my committee members, Dr. Khalid and Dr. Kent, I appreciate your time and energy spent on me and this research.

I would like to thank my colleague on this project, Stephen Fox, for his individual contributions to this project and for any aid he provided me.

I would also like to thank Advanced Micro Devices (AMD)/ATI for their funding of this research as well as NSERC for their contributions to the project.

Finally, I would like to thank my parents for their seemingly endless guidance and assistance, without you, this would not have been possible.

Table of Contents

CO-AUTHORSHIP DECLARATION .... **IV**

ABSTRACT .... **V**

DEDICATION .... **VI**

ACKNOWLEDGEMENTS .... **VII**

LIST OF TABLES .... **X**

LIST OF FIGURES .... **XI**

LIST OF ABBREVIATIONS .... **XIII**

CHAPTER 1. INTRODUCTION **1**

1.1 Project Overview 2
1.2 Thesis Organization 3

CHAPTER 2. TEST SERVER OVERVIEW **5**

2.1 History of ATE Systems 5
2.2 Currently Available ATE Systems 6
2.3 Test System, Generation One 7
2.4 Proposed Solution 9
2.4.1 Test Node Overview *12*
2.4.2 Description of RS-485 .... *14*

CHAPTER 3. HARDWARE AND SOFTWARE SELECTION **17**

3.1 Hardware Selection 17
3.1.1 Microcontroller Selection *17*
3.1.2 Microprocessor Development Kit Selection *19*
3.1.3 FPGA Selection *19*
3.1.4 FPGA Development Kit Selection *21*
3.1.5 RS-485 Transceiver Selection *21*
3.2 Software Selection 22
3.2.1 MCU Compiler and Programmer Selection *22*
3.2.2 FPGA Synthesizer and Programmer Selection *23*
3.2.3 Electrical Schematic Editor Selection *23*

CHAPTER 4. VHDL DESIGN AND VERIFICATION OF FPGA-BASED MULTI-PORT UART **24**

4.1 VHDL Design – Server 25
4.2 VHDL Design – MCU_to_485 Module 28
4.2.1 VHDL Design – MCU_buf Module *30*
4.2.2 VHDL Design – FIFO Module *30*
4.2.3 VHDL Design – RS485_output Module *33*
4.3 VHDL Design – RS485_to_MCU Module 33
4.3.1 VHDL Design – MCU_output Module *36*
4.3.2 VHDL Design – RS485_fifo Module *36*
4.3.3 VHDL Design – Arbiter Module *37*
4.3.4 VHDL Design – RS485_buf Module *38*

4.4 VHDL Design - Server_Test Module 39

CHAPTER 5. DEVELOPMENT OF C CODE FOR TEST SERVER EMBEDDED MICROCONTROLLER 41

5.1 Initialization Procedures 42
5.2 Main Processing Loop 43
5.2.1 Stack Operations *44*
5.2.2 Accept and Store Packet *44*
5.2.3 Communication with FPGA *45*

CHAPTER 6. PC CLIENT DESIGN 46

6.1 TCP Versus UDP 46
6.2 PC Client Operation 48
6.2.1 Packet Sort Algorithm *49*
6.3 Changes to Adapt Client.c Code to Unix/Linux 51

CHAPTER 7. CONCLUSION 53

7.1 Test Server Limitations 55
7.2 Thesis Contributions 55
7.3 Test Server Verification and Final Results 57
7.4 Future Work 57

REFERENCES 60

APPENDIX A. SCHEMATICS AND BILL OF MATERIALS 62

A.1 Schematics 62
A.2 Bill of Materials 69
A.2.1 Bill of Materials – Prototyping Stage *69*
A.2.2 Bill of Materials – Final Test Server *69*

APPENDIX B. TEST SERVER VHDL CODE 75

B.1 server.vhd 75
B.2 mcu_to_485.vhd 76
B.2.1 mcu_buf_mem.vhd *76*
B.2.2 rs485_output.vhd *77*
B.2.3 fifo.vhd *79*
B.2.4 mcu_buf.vhd *81*
B.3 rs485_to_mcu.vhd 83
B.3.1 rs485buf_mem.vhd *84*
B.3.2 mcu_output.vhd *85*
B.3.3 rs485_fifo.vhd *87*
B.3.4 arbiter.vhd *88*
B.3.5 rs485_buf.vhd *89*
B.4 Server_test.vhd 92

APPENDIX C. TEST SERVER C CODE 95

APPENDIX D. PC CLIENT C CODE 109

VITA AUCTORIS 125

List of Tables

Table 2.1. Test Server Component Summary 11
Table 2.2. Summary of Packet Structure 12
Table 3.1. Microcontroller Summary 18
Table 3.2. Summary of Chosen MCU [11] 18
Table 3.3. Summary of Available MCU Development Kits [12] [13] 19
Table 3.4. Available FPGA Summary [14] 20
Table 3.5. Summary of Chosen FPGA 20
Table 3.6. Summary of Available FPGA Development Kits [15] 21
Table 3.7. Summary of Selected RS485 Transceiver [16] 22
Table 6.1. Acknowledge Values and Their Meaning 48
Table 6.2. Header Differences, Unix vs. Windows [19] 51
Table 6.3. Socket Initialization Differences, Unix vs. Windows [20] 51
Table 6.4. Socket Application Shutdown Differences, Unix vs. Windows [20] 52
Table 6.5. Socket Error Reporting Differences, Unix vs. Windows [20] 52
Table 6.6. Conditional Group C Code Example - Header Calls 52
Table A.1. Bill of Materials - Prototyping Stage 69

List of Figures

Figure 1.1. System Overview 2
Figure 2.1. Visualization of Star Topology 8
Figure 2.2. Proposed System Overview 9
Figure 2.3. Test Server Layout Overview 11
Figure 2.4. Test Node Overview 13
Figure 2.5. Length of Cable vs. Data Rate of an RS-485 Network [6] 15
Figure 2.6. Daisy Chain Topology 16
Figure 4.1. VHDL Design Flow 25
Figure 4.2. Block Diagram of Server Module 26
Figure 4.3. Waveform of Data Transfer From MCU to FPGA 27
Figure 4.4. Waveform of Data Transfer From FPGA to MCU 27
Figure 4.5. Block Diagram of MCU_to_485 Module 29
Figure 4.6. Waveform of Reading From Dual-Port RAM Module 32
Figure 4.7. Waveform of Writing To Dual-Port RAM Module 32
Figure 4.8. Block Diagram of RS485_to_MCU Module 35
Figure 4.9. Flowchart of Arbiter Operation 38
Figure 5.1. Flowchart of MCU Main Processing Loop Operation 43
Figure 6.1. Flowchart of UDP Operation [17] 47
Figure 6.2. Flowchart of TCP Operation [17] 47
Figure 6.3. Flowchart of Packet Sort Algorithm 50
Figure A.1. Test Server MCU Schematic 62
Figure A.2. Test Server MCU Schematic – Ethernet, ICSP, LCD, LEDs, Oscillator, and Power 63
Figure A.3. Test Server FPGA Schematic 64
Figure A.4. Test Server FPGA Schematic - LEDs, Oscillator, Power, and Reset 65
Figure A.5. RS485 Transceiver Network (First 8) Schematic 66

Figure A.6. RS485 Transceiver Network (Second 8) Schematic.. 67

Figure A.7. RS485 Transceiver Network Schematic – Power.. 68

List of Abbreviations

A/D	Analog to Digital
ASIC	Application-Specific Integrated Circuit
ATE	Automated Test Environment/Equipment
D/A	Digital to Analog
E-Pot	Electric Potentiometer
FPGA	Field Programmable Gate Array
FIFO	First In First Out
GP I/O	General Purpose Input/Output
GPU	Graphics Processing Unit or Graphics Card
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IC	Integrated Circuit
ICSP	In-Circuit Serial Programming
MAC	Media Access Control
MCU	Microcontroller
MPU	Microprocessor
MSI	Medium Scale Integration
PC	Personal Computer
PCB	Printed Circuit Board
PCI	Peripheral Component Interconnect
RAM	Random Access Memory
ROM	Read-Only Memory
SNMP	Simple Network Management Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
UART	Universal Asynchronous Receiver/Transmitter
UDP	User Datagram Protocol

VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
VLSI	Very Large Scale Integration

Chapter 1. Introduction

In modern industrial research and development, the time to market, or the length of time that it takes to fully design and test a product and offer it for sale to the general public, needs to be as short as possible, while still maintaining all previous quality control standards. There are many steps in this process, including but not limited to, determining customer wants/desires in a new product, initial design phases outlining the overall functionality of the product, development of any new technologies to enable this product to come to market (if necessary), design of the product in full, complete testing of all features of the new product, final marketing and finally, the sale of the product. Any of these stages can be shortened to quicken the overall time to market, though this thesis is mainly concerned with only one of these stages of development, the testing.

Testing of new products is a very complicated and time consuming process, and just because a certain test on one machine in particular performed flawlessly, does not mean that the same test on a different machine will perform the same way. In other words, when components from outside parties are used in a system, they need to be verified in several different systems to ensure that they are functioning correctly in the system as a whole. This then requires that large batches of a new machine or product need to be tested to make sure that manufacturing and design tolerances do not factor into overall product robustness. If one were to test large batches of machines with a complicated testing procedure one at a time, the testing process would be prohibitively long, thus it is desired to have the ability to test large batches of a new product simultaneously with as little human input as possible.

This thesis and the joint research performed by Stephen Fox, is concerned with the development of an automated test environment (ATE) with the capability to test several key performance criteria on up to 256 separate computers simultaneously.

1.1 Project Overview

This project has two main deliverables: the test server and the test nodes. This thesis will describe the design and construction of the test server, whereas the thesis of my colleague, Stephen Fox, will describe the design and construction of the test nodes. The test server has the ability to control, and communicate with, a maximum of 16 buses, each with a maximum of 16 nodes for a total maximum node count of 256. The nodes in each bus are in a daisy chain configuration, which was chosen for its simplicity and for its affinity to the RS485 electrical standard which is our chosen communication medium. The basic physical layout of the system is shown in Figure 1.1.

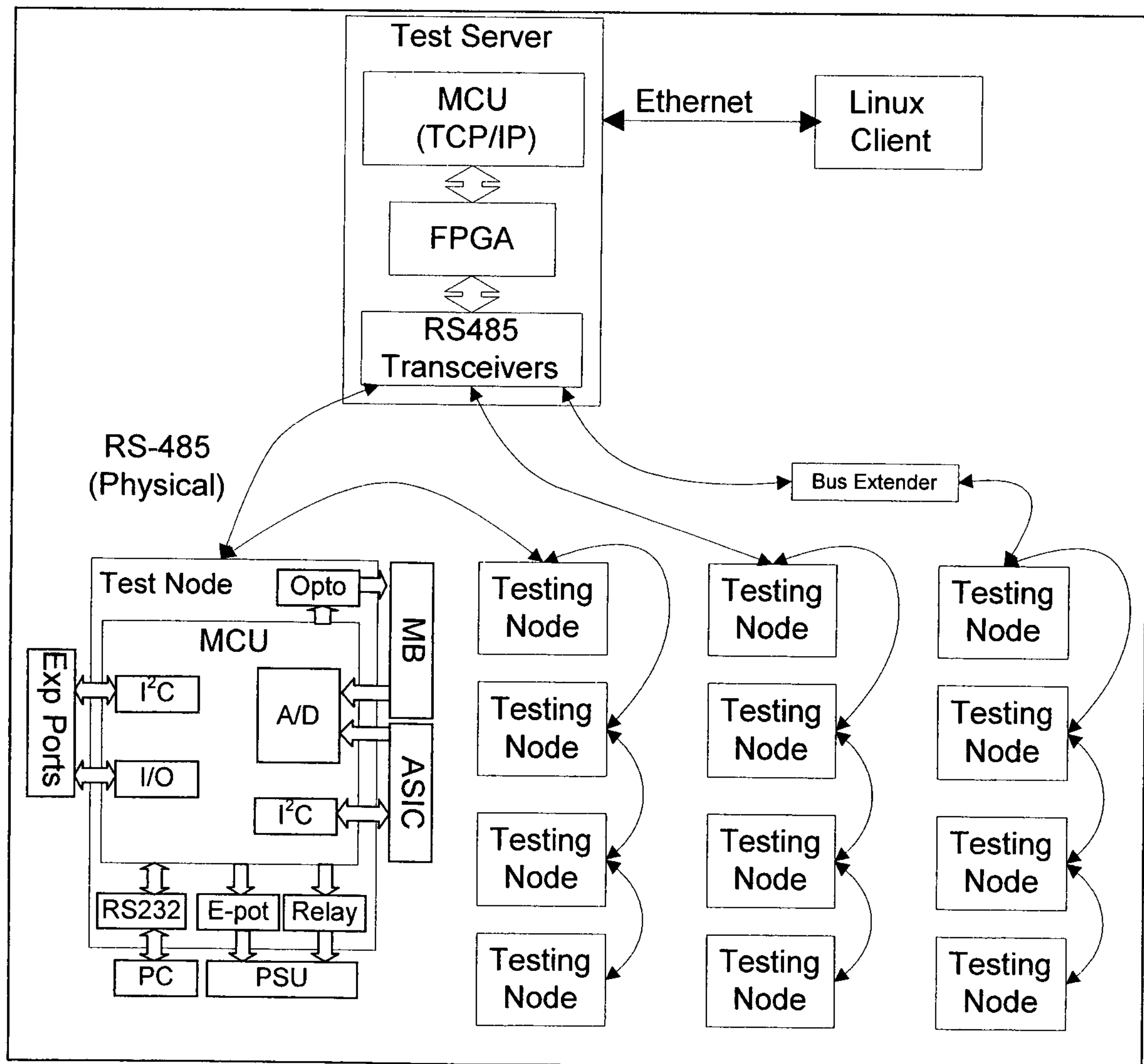

Figure 1.1. System Overview

The system functions as follows: the client reads in a file of 30 byte long packets, which are in hexadecimal format, orders the packets in a manner such that any possibility for contention on a bus is eliminated, and transmits them, one at a time, over the TCP/IP socket to the test server microcontroller. The test server then calculates a CRC-16 checksum number and adds it to the end of the packet, making a packet length of 32 bytes. The checksum is needed to ensure that noise or other external factors did not alter the data during transmission to the node. The test server FPGA will then send the packet out to the correct bus based on the address byte in the packet. The test node that the packet is addressed to will then act on the command passed to it in the "command" field of the packet and will send back any data that is requested. The test server, on receipt of the data from the test node, will re-calculate the CRC and make sure there were no transmission errors; it will then forward the reply over the TCP/IP socket back to the PC client software which is maintaining control and safe operation of the entire system.

1.2 Thesis Organization

This thesis describes the design and construction of the test server for a custom automated test environment (ATE). Chapter 2 will provide an overview of the test server and its capabilities as well as review the structure of the test nodes as designed by my colleague Stephen Fox. Chapter 3 will describe the various pieces of hardware that make up the test server and the associated software tools that were used in the project's development. This chapter will describe the pieces of hardware currently being used in the prototype of the test server and will give recommendations for the hardware that should be used in the production of the server. Chapter 4 will describe the development and design of the VHDL code for the MCU_to_485 and RS485_to_MCU FPGA modules. Chapter 5 will discuss the PIC microcontroller that was chosen for this

project in detail and will also discuss the C code that was written for its operation. Chapter 6 will discuss the design of the PC client software, written in the C programming language, which is controlling the system as a whole. Finally, Chapter 7 will conclude the thesis and will discuss suggestions for future work.

The content of the appendices is as follows: Appendix A consists of the schematics of the test server hardware and the Bill of Materials, Appendix B contains the VHDL code for the test server FPGA, Appendix C contains the C code for the test server microcontroller and Appendix D contains the C code for the PC client.

Chapter 2. Test Server Overview

2.1 History of ATE Systems

ATE systems have had a relatively short life and have developed hand in hand with commercial computers. The first automatic test devices were created in the 1950's by the United States' Department of Defense (DOD), primarily to test missile systems and other military electronics. It should be noted that this development coincides with the increasing availability of digital logic and commercial computers. The U.S. DOD realized that a more repeatable and consistent method of testing new military electronics was needed, and thus a need for automated testing equipment was born. The first ATE systems were created primarily of existing manual instruments which had custom designed digital logic added to allow them to be programmable. This was certainly not an ideal situation, and the end goal was to move to more universal test equipment [1].

A system was desired that had a full set of programmable instruments which could be configured and reconfigured at will, as quickly and easily as possible. The first of these systems was created between 1968 and 1972 and it was called the Versatile Avionics Shop Test (VAST) and it consisted of a test station, a computer and a data transfer unit. It was created to test the avionics systems on F-14A and S-3A aircraft and was deployed on aircraft carriers starting in 1974. The VAST systems eliminated some special support equipment and reduced the number of technicians required to test and maintain the avionic systems, and thus were, at least partially, successful at what it set out to do. A test program set was created to test each individual system, but unfortunately, it could only test one avionic system at a time [2].

As computers continued to improve, so too did the automated test environments, by the mid 1970's, they were capable of testing medium scale integration

circuits (MSI) that had hundreds of transistors per chip. During the 1970's another important development happened: buses were introduced which greatly enhanced interfacing between components. This led to the development by Hewlett-Packard of the Hewlett-Packard Interface Bus (HPIB), also referred to as the General Purpose Interface Bus (GPIB). This bus was standardized by the IEEE in 1975 into IEEE 488.1 and it is still widely used to this day, though the standard has been upgraded by the IEEE in the past 30 years. In the early 1980's, the automated test environments started to include more modern embedded systems, including embedded microprocessors, this allowed the systems to become "smarter" and more customizable [1].

The late 1980's and into the 1990's saw the further development of ATE systems, which included standardizing the communication bus to allow multiple vendor's components to communicate easily with each other, and the introduction in the 1990's of "plug and play", which functioned very similar to USB in that components could be plugged in and the system handled installing any drivers to allow that component to work in the system as a whole.

2.2 Currently Available ATE Systems

There are currently several large vendors of off the shelf ATE systems. These include National Instruments, Northrop Grumman, and Teradyne [3] [4]. Prices on individual modules from these vendors range in price from $1,000 to more than $18,000 depending on the module and many of these ATE systems are the size of a refrigerator. These systems can test virtually all aspects of a custom VLSI design as the modules include A/D converters, D/A converters, signal generators, variable power supplies, signal analyzers, multimeters and radio frequency and high frequency devices. However, many of these systems are designed to test one chip at a time. Clearly the cost, size and the inability to test many systems at once is a large drawback of these devices.

There is also a large number of competing standards as to how these units should communicate with each other. There are currently several standard instrument types including using PC standard input/output (I/O) such as USB and Ethernet as well as RS-232 serial connections and also add-in cards like ones which used the GPIB interface. There are also some custom instrument types including VME eXtensions for Instruments (VXI) components which was development in the 1980's, primarily for use by the U.S. military, PCI eXtensions for Instruments (PXI), which was created by National Instruments in 1997 and is based off the Compact PCI bus, and finally LAN eXtensions for Instrumentation (LXI) which uses high-speed LAN as the backplane of the ATE [5].

2.3 Test System, Generation One

Approximately three years ago, AMD/ATI realized that the currently available, off-the-shelf ATE systems were not ideal for testing the functionality and performance of their graphics cards when deployed in a motherboard, so they set out to create their own system. This first generation of the project consisted of similar hardware to generation 2, however, to save on the development time, some off-the-shelf components were used that were not an ideal solution for the manner in which they were going to be used.

Generation one consisted of a National Instruments digital I/O board that was connected to a host PC that controlled the entire system. The digital I/O board ran direct connections to each of the test nodes and sent them the control signals directly using a serial connection. This created several problems, but was deemed at the time to be the ideal solution as it could be deployed quickly. The first issue with the system is the increased cost of the National Instruments board as compared to a custom built control board. The cost of a custom board with an FPGA and a microcontroller, similar to what is used in the solution that will be described in this paper, is considerably less than the cost of the National Instruments board. The second major issue is that the

overall system had a very large quantity of cabling that was necessary to connect everything; this was mainly due to the topology that was used in deploying the network. When creating a system with one central server which has an output that connects to each individual node, you generally end up with a star network topology as shown in Figure 2.1. The problem with a star is that as nodes get farther away from the server, the cable length to connect them also increases in size. The third problem was due to the communication medium they used to talk to the nodes; they used a common serial connection that used ground as a signal reference, which led to galvanic isolation problems. The problem with this, and it was a large problem that is not easy to overcome, is that "ground" is a relative measurement. There can be a difference in the voltage levels of ground between the different nodes as they are spaced around the room in which the system is deployed and this difference in ground level can cause one node to think a digital '1' is a '0' or a '0' is a '1'.

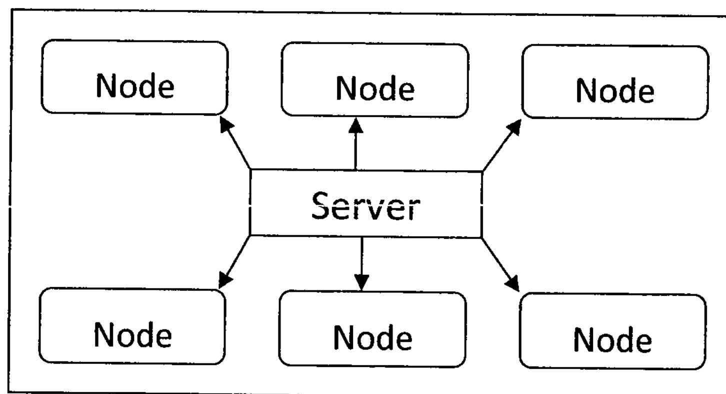

Figure 2.1. Visualization of Star Topology

With these difficulties in mind, generation two of the test network was deemed the next step in development. The proposed solution for the second generation of this project must then meet several criteria. First of all, it needs the ability to test and control multiple parameters of multiple nodes simultaneously. Secondly, it needs to communicate with these nodes in such a manner that cabling length and complexity is reduced from the first generation and the effects of external noise and differences in ground levels between nodes needs to be eliminated from the system. The system must also be easily upgradeable, as it is impossible to determine what parameters will need

to be tested or controlled in the future. Lastly, it needs to be deployed as cheaply and easily as possible.

2.4 Proposed Solution

With the criteria and problems defined, as stated in the previous section, the second generation of the system could be designed. We desired the system as a whole to have Ethernet communication to a Linux client as well as the capability to test up to 256 test nodes simultaneously. The decided upon layout of the system can be seen in Figure 2-2. The test network will consist of a test server that can communicate over Ethernet to the client on one side and on the other it needs to communicate to 16 buses of 16 nodes each, laid out in a daisy chain topology which was chosen for its simplicity. Thus the system provides an overall count of 256 nodes.

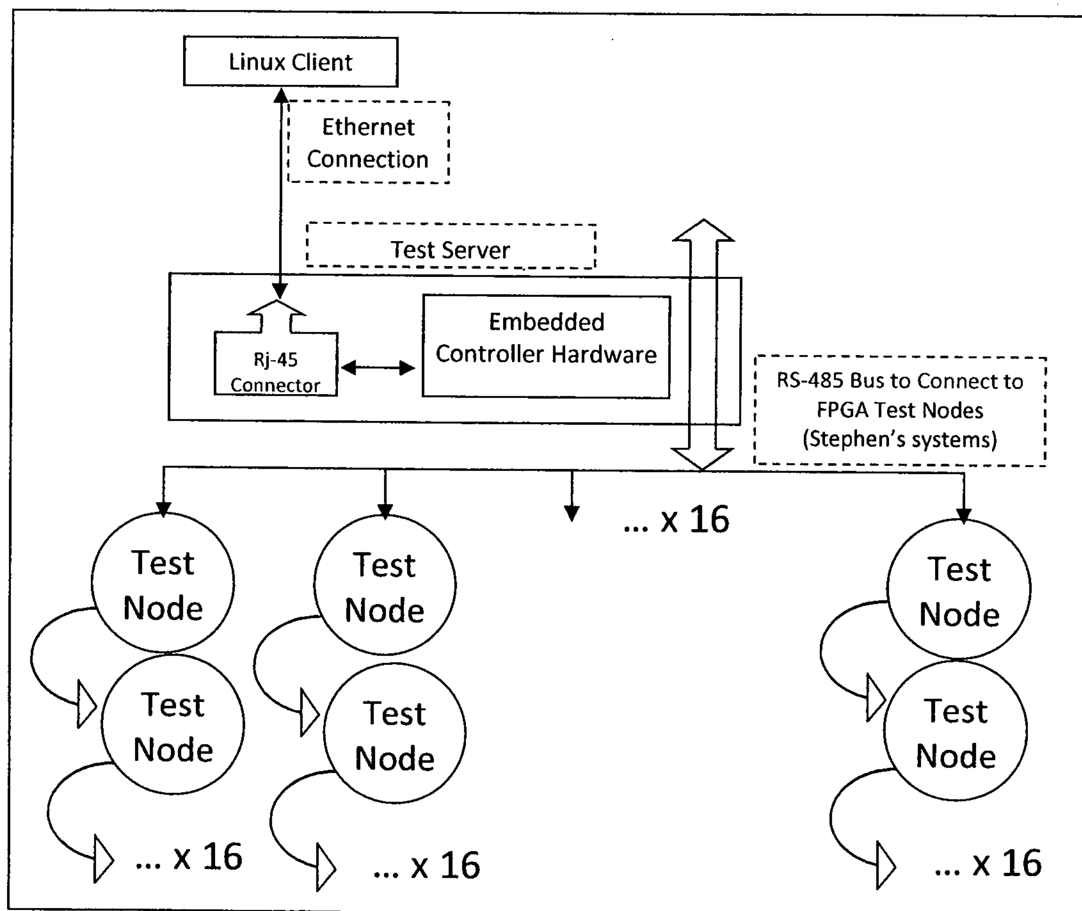

Figure 2.2. Proposed System Overview

From the above figure, it is apparent that there are three main components to the system as a whole: the Linux client, the test server and the test nodes. This thesis will describe the development of the test server and the Linux client in detail, the design of the test nodes can be seen in Stephen Fox's thesis.

From this abstract design of the system, the functionality of the test server could be determined:

1. It needs to be Ethernet capable
2. It needs to have enough I/O to communicate effectively with 16 buses of test nodes
3. It needs the computing power to handle processing a large throughput of data from the Linux client to the test nodes.

From this early stage it was realized that we would not be able to find a microcontroller that had the I/O capabilities and processing power needed to communicate with 16 buses of test nodes as well as having the ability to communicate over an Ethernet connection, an FPGA of some kind would be needed in the test server to handle the large amount of I/O to the test nodes. Alternatively, an FPGA is not an ideal solution on its own as the nature of the test server lends itself better to being deployed on a microcontroller, and the cost of a microcontroller is significantly less than that of a FPGA, thus both a microcontroller and an FPGA are to be used in the test server. The microcontroller would be used to communicate over the Ethernet connection with the Linux client and the FPGA would be used as a multi-port UART to communicate with the test node network. The layout of the test server can be seen in Figure 2-3.

The final criteria that the system needed to meet was that the effects of the different levels of ground between the nodes and the external electronic "noise" present in the test server be eliminated. This was accomplished with the choice of RS-485 as the communication medium for the test node network and with standard Ethernet cabling terminated with RJ45 connectors, or, more officially, CAT-5e cabling, as the physical transmission medium. RS-485 is a half-duplex, differential electrical

standard that is ideal for a multi-drop communication link (a multi-drop communication link is one in which a single bus has multiple devices connected to it). Though RS-485 is a half-duplex communication medium, it can be made to be full-duplex by using one transceiver for transmitting and another for receiving; this is how RS-485 is implemented in this system. The choice of RS-485 and CAT-5e cabling will be described in more detail in Chapter 3.

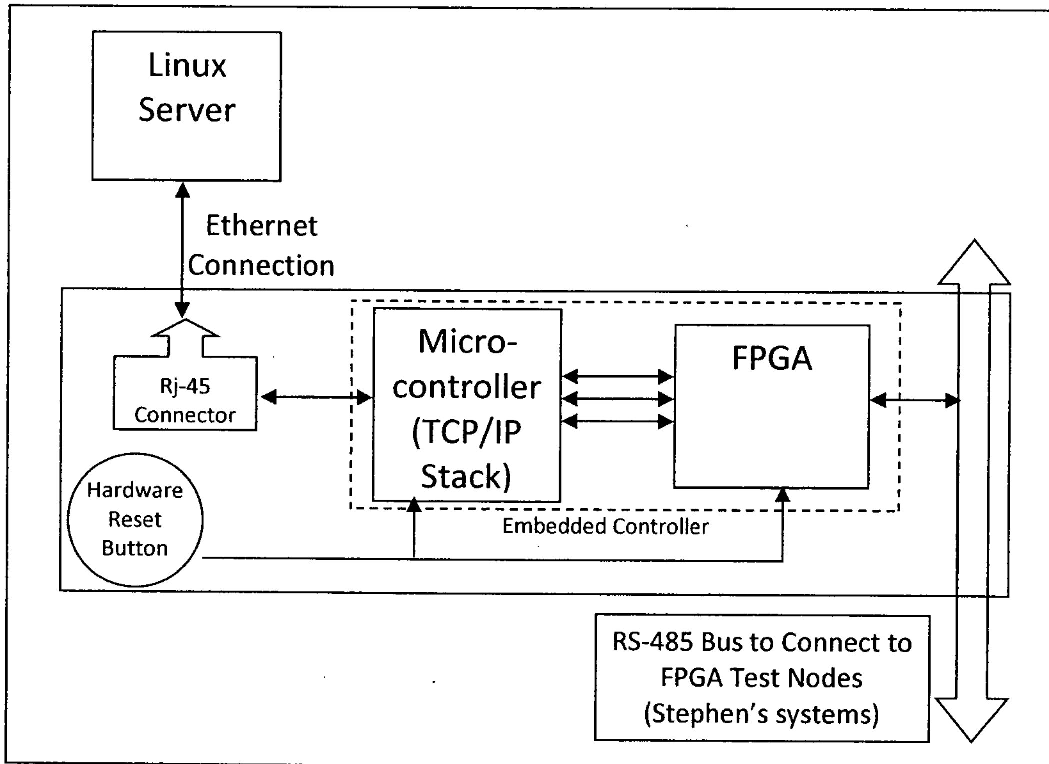

Figure 2.3. Test Server Layout Overview

After finalizing the above general layout of the test server hardware, a component list can be created that will outline all hardware needed to construct the test server. This component list is summarized in Table 2-1.

Table 2.1. Test Server Component Summary

Component	Manufacturer	Quantity Needed
Microcontroller	Microchip	1
FPGA	Altera	1
RS485 Transceiver	Linear Technology	32
RJ45 Female Connector	Tyco Electronics	17

One final decision that had to be made was the format of the packet. It was deemed the best solution to use a packet length that is a factor of two, and since many of the packets that are going to be transmitted are quite short, a smaller packet size is ideal. The chosen packet length is 32 bytes, including the CRC-16 check bytes at the end of the packet. The structure of a packet is shown in Table 2.2.

Table 2.2. Summary of Packet Structure

Byte(s):	1	2	3	4	5-30	31-32
Field:	Address	Command	Acknowledge	Length	Data	CRC-16

2.4.1 Test Node Overview

Each test node consists of a PIC microcontroller with in-circuit serial programming ability (ICSP), a power relay board, an external temperature sensor, an analog to digital converter (A/D converter), RS-232, RS-485 and I^2C communication, and an electric potentiometer (E-Pot). An overview of the test nodes can be seen in Figure 2.4. The in-circuit serial programmer is used for updating the program in the MCU, the power relay board is used for cutting the power to the PC under test and is only used in emergency situations (similar to unplugging the machine from the wall outlet).

The test nodes are capable of controlling and monitoring several different aspects of the PC under test. The nodes can turn the PC under test on/off, cut the power to the PC using a relay board, set and monitor the temperature of the PC under test, and set and monitor the voltage on the 12V and 3.3V voltage rails. The nodes can also communicate with both the PC under test and the GPU in the PC, and finally the node can update its own program using the bootloader mode that has been programmed into the microcontroller. Also, the nodes were designed with expandability in mind; they can be expanded to add extra functionality to each of the individual test nodes.

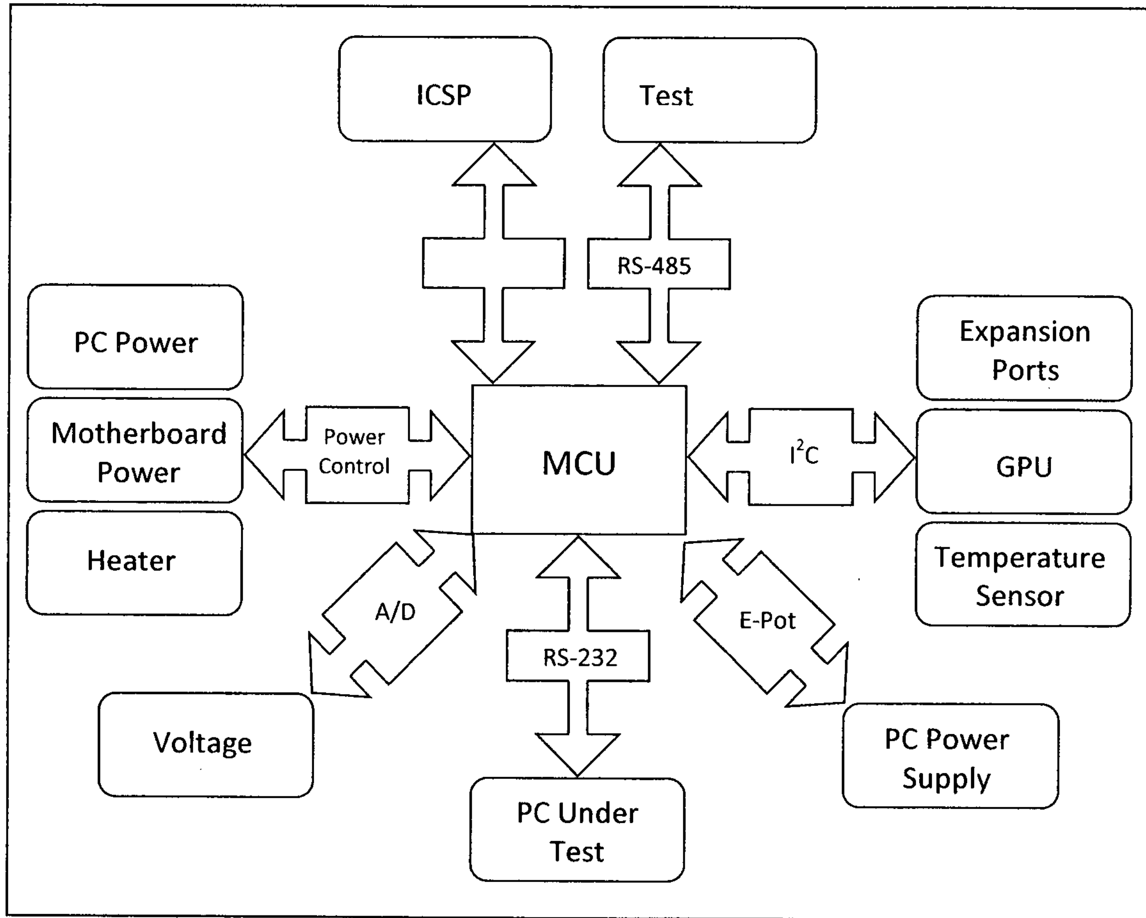

Figure 2.4. Test Node Overview

The external temperature sensor is used to monitor the temperature of the PC under test and the A/D converter is used to monitor the system voltages. The external temperature sensor has a built-in A/D converter while the one used for monitoring the system voltages is embedded in the MCU. The external temperature sensor is part of the feedback on the temperature control of the system. The final system, when executed at the offices of AMD/ATI in Markham will have a heater element at each node that will allow the user to increase the ambient operating temperature of the test node. The external heating element is controlled by an external relay to the MCU. The RS-232 communication is used to communicate with the PC under test, the RS-485 is used to communicate with the test server, and the I^2C bus is used to communicate with the

GPU, communicate with the temperature sensor, and also for future expansion of the test node. Finally, the E-Pot is used to alter the voltage from the PC power supply.

2.4.2 Description of RS-485

As mentioned in the previous sections, the test node network is implemented using the RS-485 electrical specification. Though RS-485 does not contain a communication protocol as part of the specification, and thus a custom protocol had to be developed for the system, RS-485 as a physical transmission standard solved many of the problems that affected the first version of this test network. RS-485 was standardized by the Electronic Industries Alliance (EIA) in association with the Telecommunications Industry Alliance (TIA) and thus the official name of the standard is TIA/EIA-485-A. The last revision of the standard was in 1998 [6].

The RS-485 electrical standard is described as a half-duplex, differential signal, multi-nodal, serial communication medium. Half-duplex refers to the fact that communication can only happen in one direction at any point in time, for example, point A can send a message to point B or B can send a message to A, but A cannot send a message to B while B is sending a message to A. A typical example of half-duplex communication is a common "walkie-talkie". Differential signaling means that both the signal itself, and its digital logic inverse, are sent at the same time over two separate wires from the sender to the receiver, these wires are often referred to as a twisted pair, as in reality, the pair of wires is usually physically "twisted" together. In differential signaling, the signal is determined at the receiver by taking the difference between the two signals that are sent by the receiver. This property means that any noise that is introduced to one wire during transmission is introduced to both, and when the difference between the wires is taken at the receiver, the noise is removed from the signal. Finally, a serial signal is one in which each bit of the message is sent one at a time from the sender to the receiver.

RS-485 as a standard is designed to support up to 32 nodes on one bus with a maximum length and data transmission rate that are inversely proportionate to one another. In other words, as the data transmission rate increases, the maximum length of the cable between the sender and receiver decreases, and inversely, as the length of the cable increases, the maximum data transmission rate decreases. Theoretically, the standard can support up to 10Mbit/s for a cable up to 10m or 100kbit/s for a cable length of 1200m. If the user wishes to increase the length of the cable at a given transmission rate, or the number of nodes on the bus, an rs-485 repeater is needed to increase the strength of the signal. This tradeoff is shown in Figure 2.5.

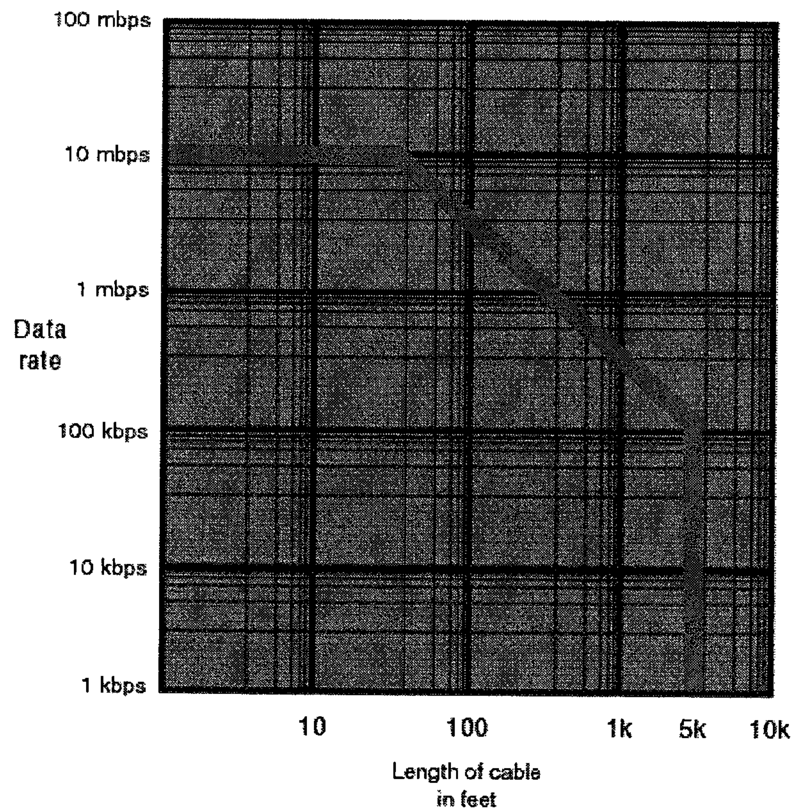

Figure 2.5. Length of Cable vs. Data Rate of an RS-485 Network [6]

In the implementation of RS-485 in this project, several things need to be mentioned. The first is that although RS-485 is defined in its standard as a half-duplex communication, this project has executed it as full-duplex by running a second twisted pair of wires for communication and thus an extra transceiver is needed at each end of the communication. The second is that even though RS-485 is a differential

communication standard, there can exist a difference in the voltage level of ground between a transmitter and a receiver, thus a wire is needed to connect the ground signals between the transmitter and receiver to alleviate this issue. In this implementation, we are using Cat-5e cable as the physical transmission medium, and thus have access to 4 twisted pairs, or 8 wires total. Wires 1 and 2 are used for transmission from the test server to the test nodes, where wire 1 is the positive signal and wire 2 is the negative signal. Wires 3 and 6 are used for transmitting from the test nodes to the test server, thus implementing a full-duplex communication, where wire 3 is for the positive signal and wire 6 is the negative signal. Finally, wire 8 is used for the ground signal which connects all ground pins of all nodes on the RS-485 bus together. The last thing that needs to be mentioned in regards to our implementation of an RS-485 network is the topology of the network. Though many topologies exist including star networks, networks with a backbone, ring networks, etc..., we chose to use the daisy-chain topology as shown in Figure 2.6. The main reason to use a daisy-chain configuration is to reduce the reflection of the signal that is present in the physical wire that is transmitting the signal. This reflection is further reduced by adding a termination resistor at either end of the communication network.

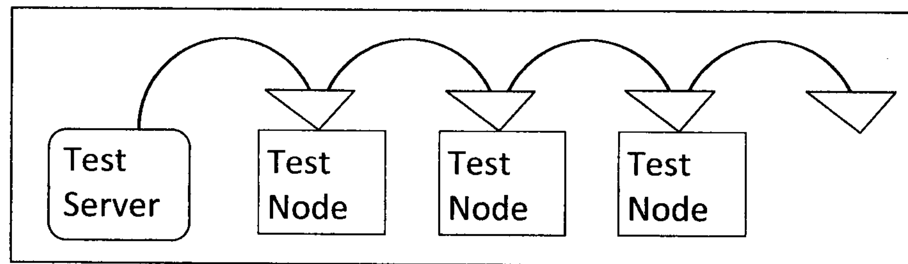

Figure 2.6. Daisy Chain Topology

Chapter 3. Hardware and Software Selection

3.1 Hardware Selection

3.1.1 Microcontroller Selection

After the realization that a microcontroller on its own would not be sufficient to implement both Ethernet functionality and output to the 16 buses of test nodes, a more detailed specification list was arrived at for the microcontroller hardware. The system needs to be able to communicate both with the client computer over an Ethernet connection and the FPGA over some custom direct communication protocol and it needs the processing power and internal data memory to handle the large throughput of data while still being able to do checks and calculations for the checksum digits and any other calculations that may be needed to maintain full functionality of the system.

The need for Ethernet connectivity with the microcontroller created two options: use a microcontroller with built-in Ethernet capability, or run an external Ethernet controller. An external Ethernet controller would pull all TCP/IP stack functionality out of the main microcontroller and thus would save the internal data memory of the MCU for user programming. Unfortunately, running an external Ethernet controller adds complexity to the final board layout and it creates a bottleneck in the communication between the Ethernet controller and the microcontroller which is in control of the system as a whole. Thus it was decided that a microcontroller with built-in Ethernet functionality would be the more desirable solution for the test server. This decision instantly reduced the number of possible microcontroller vendors down to a very small group. These vendors include Microchip, the manufacturers of PIC microcontrollers [7], Freescale Semiconductor [8], Digi International, manufacturer of ARM based

microcontrollers [9] and Atmel [10]. A summary of these products can be seen in Table 3.1.

Table 3.1. Microcontroller Summary

Company	Microchip	Freescale	Digi	Atmel
Microprocessor	PIC18FXXjXX	MCF532X	NSXXXX	AVR32
Max Processing Speed	41.667 MHz	240 MHz	200 MHz	66 MHz
Architecture Word Length	8 bits	32 bits	32 bits	32 bits
Program Memory	128 KB	128 KB	8 KB	512 KB
RAM (Bytes)	3,808	32K	4K	64K
Max Pin Count	80	256	388	144
Average Cost	$7.76	$40	Unknown	$15.17
TCP/IP Stack Software Support	Yes	Yes	Yes	Yes
Development Kit Available	Yes	Yes	Yes	Yes

All listed microcontroller architectures meet the base requirements for the microcontroller in the test server, so other factors were used to choose the ideal microcontroller for the project. Cost is a large concern for this project; costs need to be kept as low as possible or one of the advantages of building a system from scratch is lost to an off-the-shelf system, thus the Freescale chip was deemed too costly. The Digi microcontroller proved difficult to find a vendor that sold the microcontroller separately from a development board and it thus was ruled out as the final deliverable of the project will consist of a custom PCB with the microcontroller mounted on the same board as the RS485 transceivers and the chosen FPGA. The Atmel microcontroller and the PIC microcontroller were the two remaining candidates, but with the PIC microcontrollers reduced cost, the availability of already licensed software compilers at the University and previous experience with the PIC microprocessor; the Microchip PIC18F97J60 microcontroller was chosen as the ideal option. Table 3.2 shows a summary of the PIC18F97J60 MCU.

Table 3.2. Summary of Chosen MCU [11]

PIC18F97J60	
Flash Program Memory (bytes)	128K
SRAM Data Memory (bytes)	3,808
Ethernet Buffer (bytes)	8192
I/O	70

3.1.2 Microprocessor Development Kit Selection

With the choice of the PIC microcontroller as the chosen device for the test server, a development kit needed to be found for the prototyping stage of development. There are two main vendors of PIC based development kits, Microchip and CCS. A comparison between the two development kits is shown in Table 3.3.

Table 3.3. Summary of Available MCU Development Kits [12] [13]

Company	CCS	Microchip
MCU Used	PIC18F67J60	PIC18F97J60
Program Memory	128K	128K
I/O Pins available	20	70
Access to all MCU Pins?	No	Yes
Available Buttons	1	4
Available LED's	3	8
Includes Programmer?	Yes	No
Cost	$149US=$162 CDN	$194.06

Communication with the FPGA requires exactly twenty pins; 8 pins for data to the FPGA, 8 pins for data from the FPGA and 4 pins for request and acknowledge lines, therefore, even though the CCS board has 20 I/O pins available, choosing this board and MCU would eliminate the possibility of any further expansion of the test server MCU. The Microchip development kit, therefore, is the board that was chosen for the prototyping of the test server hardware.

3.1.3 FPGA Selection

Much of the decision on which FPGA to use is based on past experience and on keeping costs as low as possible. Since I have had quite a bit of past experience with Altera's FPGAs and their development environment, Quartus II, it would speed up development time to use an Altera FPGA for the test server. The next major consideration was in finding a low cost FPGA that meet our requirements of running at 25MHz with enough memory bits and logic elements to contain the entire test server

hardware. Since the number of logic elements required is not something that can be determined prior to synthesizing the design as a whole, the decision had to be made on clock speed, cost and available internal memory bits. A summary of available Altera devices is shown in Table 3.4.

Table 3.4. Available FPGA Summary [14]

FPGA family	Stratix III	Stratix II	Cyclone III	Cyclone II
Available Speed Grades *	2,3,4	3,4,5	6,7,8	6,7,8
Available Memory bits	2.1M-16.2G	419K-9.4M	424K-8.2M	120K-1.1M
Available Logic Elements	47.5K-337.5K	15.6K-180K	5K-200K	4.6K-68.4K
Available User I/O	296-1120	366-1170	182-413	158-622
FPGA Cost	$540-13500	$223-13800	$31.40-740	$40-500
Min. Cost that meets Specs	$540	$223	$52	$49
Dev. Kit Cost	$3,000	$1100-3500	$234-4100	$176-1700
* Speed grade refers to delay in ns through a macrocell in device, lower equates to faster				

Upon reviewing the available options for FPGA's from Altera, the Stratix family of devices are too costly, thus the Cyclone family of FPGA's will be the target device. From synthesizing the complete VHDL code that is targeted to the FPGA, the system will need 11,520 logic elements, 68 I/O pins, it must be speed grade 7 or faster and it needs 90,112 of memory bits. The cheapest device in the cyclone II and III families that matches these specifications, and is successfully "fitted" by Altera's Quartus II software, is the EP2C15AF256C7N Cyclone II FPGA. A summary of this device is shown in Table 3-5.

Table 3.5. Summary of Chosen FPGA

EP2C15AF256C7N	
Total Logic Elements	14,448
Total Memory Bits	239,616
Embedded Multipliers	52
Speed Grade	7
Functional Temperature Range	0-85°C

3.1.4 FPGA Development Kit Selection

After coming to the conclusion that an Altera Cyclone II device would be the best choice for the FPGA in the test server, a suitable development kit needed to be found. When the search began for a development kit for this system, the total resource count needed for the VHDL code on the FPGA was unknown. Thus a development kit had to be found that provided more than enough memory bits and logic elements, and access to 52 I/O pins to communicate with the MCU and the RS485 transceiver board. The two main vendors for Altera FPGA based development kits are Altera and Terasic. A summary of their available Cyclone II development kits is shown in Table 3.6.

Table 3.6. Summary of Available FPGA Development Kits [15]

	Altera	Terasic	
Development Kit	Cyclone II Starter Kit	DE2	DE2-70
FPGA Used	EP2C20F484C7	EP2C35	EP2C70F896C6
Logic Elements Available	18,752	33,216	68,416
Memory Bits Available	239,616	483,840	1,152,000
I/O available	315	475	422
Price	$187.45	$582	$704

With these kits available, the DE2-70 kit was chosen as it was deemed better to err on the side of caution as far as size goes, or in other words, as only one of these development kits need to be purchased for the entire life of the project, it is better to get the larger board at the beginning to ensure that we will not have to purchase another one due to an underestimate of logic elements needed.

3.1.5 RS-485 Transceiver Selection

For the selection of RS-485 transceiver, there were just a couple criteria that the selected IC needed to meet. The transceiver had to be low power, as 32 of them would be needed in the test server alone, and it needed to be available in both a surface mount and DIP package; surface mount for final PCB manufacture and DIP package for

prototyping. The selected device is the LTC485 IC manufactured by Linear Technologies. Table 3.7 has a summary of the major electrical specifications of the device.

Table 3.7. Summary of Selected RS485 Transceiver [16]

LTC485	
Max Supply Voltage	12V
Recommended Supply Voltage	5V
Driver Output/Receiver Input Voltage	$\pm 14V$
Driver Input/Receiver Output Voltage	$-0.5V$ to $V_{CC} + 0.5V$
Operating Temperature Range	0-70°C

3.2 Software Selection

3.2.1 MCU Compiler and Programmer Selection

For any microcontroller, the first choice that must be made is whether to program in C or in the microcontroller assembly language. For this project, the decision was made for me as the TCP/IP stack that is provided by Microchip for their Ethernet-enabled MCU's was written in C, thus a C compiler is needed to create the rest of the code for the MCU. There are a couple vendors that make a C compiler for the Microchip PIC microcontrollers, though the University currently only has licenses for two of them, the CCS compiler and Microchip's MPLab software.

MPLab v8.33 and Microchip's C18 were chosen as the development software and C compiler and CCS's ICD-U40 programmer and CCS load software were chosen to load the program on the MCU. MPLab v8.33 was the newest version of the software upon starting development of the MCU C code. MPLab and the C18 compiler were chosen over the CCS compiler environment as the TCP/IP Stack software was written with Microchip's MPLab software in mind, thus the beginning development of the board would be easier on MPLab than it would in the CCS C compiler as much of the work is done for me by the TCP/IP Stack software. The ICD-U40 programmer and CCS load software were chosen as my partner on this project, Stephen Fox, had already

purchased the programmer for his half of this project, thus to save from buying two programmers, the same programmer was used for the test server as for the test nodes.

3.2.2 FPGA Synthesizer and Programmer Selection

For the VHDL synthesizer and programmer, Altera's free Quartus II Web Edition Software Version 7.2 was used. Quartus II was chosen as the synthesizer over more expensive products from vendors such as Cadence or Synopsys as the only software capable of loading the Cyclone II FPGA is the Quartus II fitter and programmer. I have also had a substantial amount of prior experience with using the Quartus II software for development, simulation and verification and thus it was any easy choice to use Quartus II for all necessary VHDL synthesizing, simulating and programming of the FPGA.

3.2.3 Electrical Schematic Editor Selection

CadSoft Eagle v5.3 was used for all electrical schematics. Eagle has the capability to convert your electrical schematics into a layout for PCB manufacture. It also has a built-in components list that includes most popular components which greatly speeds up development time. Eagle is available for Linux, Windows or Mac based computers and the University already had a license for Eagle.

Chapter 4. VHDL Design and Verification of FPGA-Based Multi-Port UART

An early decision that had to be made was the division of processing and checking tasks that are handled by the FPGA, the MCU, and the client, respectively. To keep the costs down on the FPGA chip itself, as it is the most expensive single component in the test server, it was deemed best to keep the design of the FPGA hardware as simple as possible and leave the processing and checking tasks to the more capable, and less costly, MCU and client. As the slowest portion in the pipeline of the test server is the communication to the nodes over the RS485 connection, which is operating at a speed of 100K Baud as compared to the system clock of 25MHz, the FPGA needs to be able to buffer packets that are waiting to be sent to the nodes as the MCU is not capable of buffering data with its overall lack of data memory. Thus the functionality of the FPGA is essentially to be a multi-port UART with buffering capability.

The design of the hardware for the FPGA was done in a top-down manner in VHDL. Thus the design process began with the largest component, the server itself, and determined what it needed for inputs and outputs and what its general functionality needed to be. The task was then subdivided into two halves: the first will handle data coming in from the MCU and being sent out to the RS485 network. This module is called MCU_to_485.vhd, it is described in detail in Section 4.2, and its code can be seen in Appendix B. The second half of the server will handle data coming in from the RS-485 network and being sent out to the MCU, and this module is called rs485_to_MCU.vhd, it is described in detail in Section 4.3, and its code can also be seen in Appendix B. These two halves are then subdivided into smaller modules as was deemed necessary. The design flow of each module in the system can be seen in Figure 4.1.

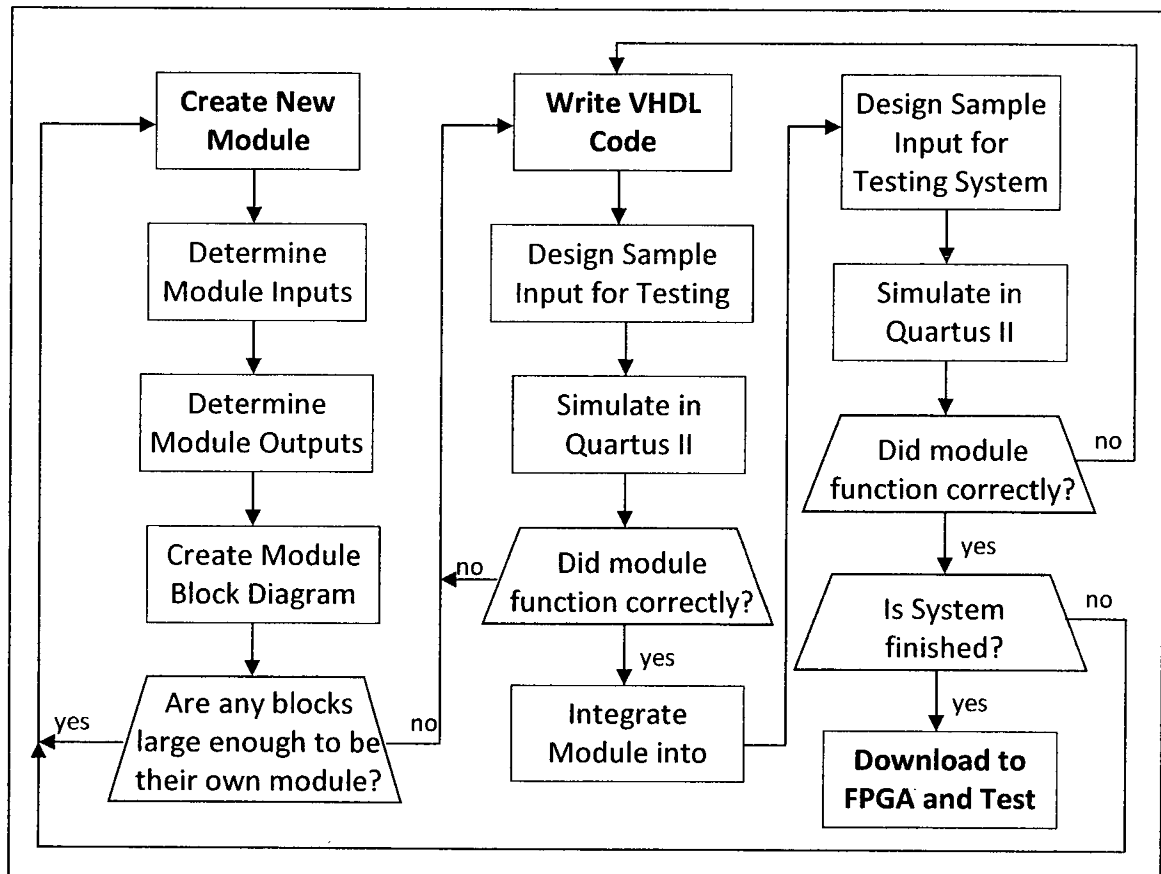

Figure 4.1. VHDL Design Flow

4.1 VHDL Design - Server

The first portion of the server.vhd code that had to be determined was the system inputs and system outputs, and consequently, the communication protocol with the MCU also had to be designed. Since FPGA's are hardware, and they operate in parallel, as opposed to the MCU which is a serial device (it executes one command at a time from the start of its program to the finish), some kind of arbitration was needed between the two halves of the FPGA and the MCU, or they may both try to speak to the MCU at the same time. There are two methods of doing this, have the two halves of the

FPGA talk to each other so that they can never communicate to the MCU at the same time, or have all arbitration handled by the MCU. The first method was attempted originally, but after simulating the hardware, the better solution was to have the MCU handle all arbitration. Thus four handshake lines are needed, one request and one acknowledge line for each half of the server hardware, plus the 8 bit wide data bus from the MCU to the FPGA and the 8 bit wide data bus from the FPGA to the MCU giving a total of 20 communication lines between the MCU and the FPGA. The outputs of the server module were simple, one 16 bit wide bus for sending data out over the RS-485

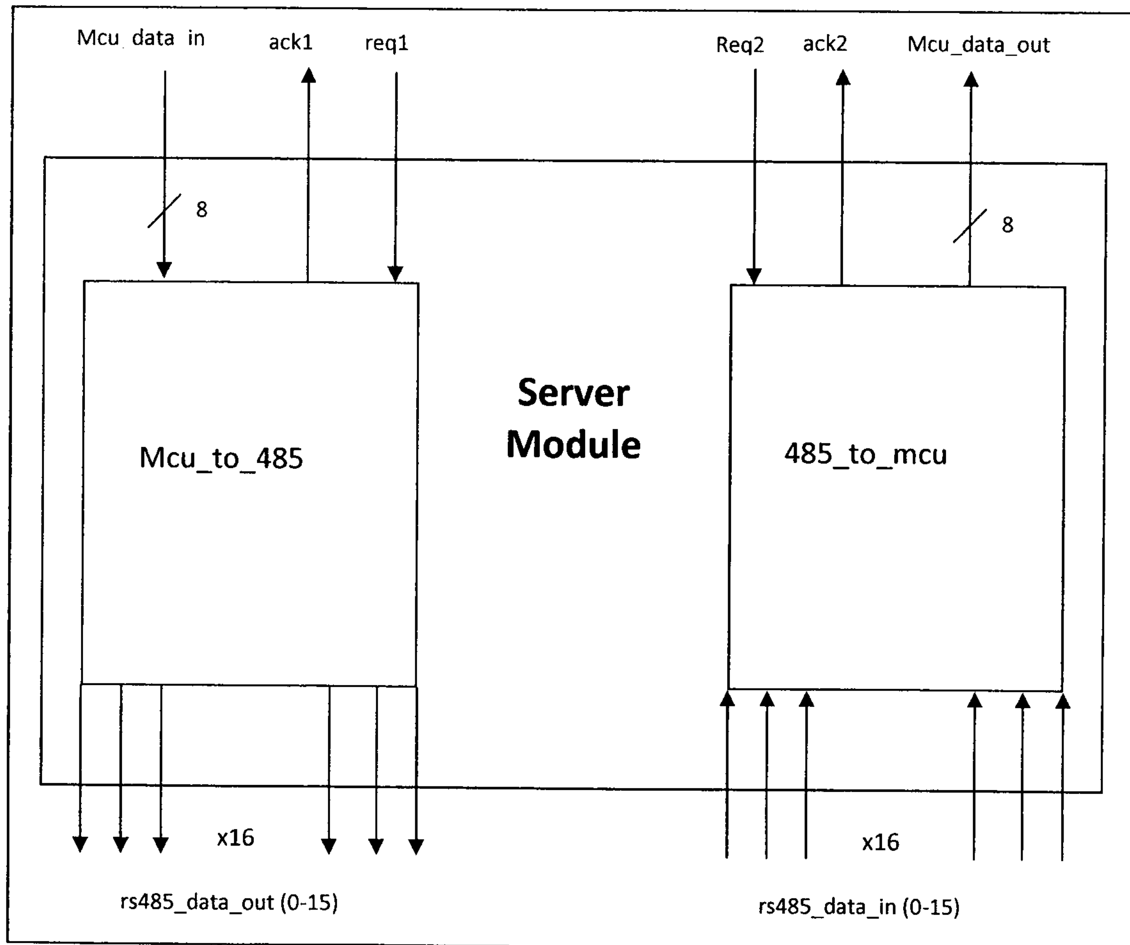

Figure 4.2. Block Diagram of Server Module

network and one 16 bit wide bus for receiving data from the RS-485 network. Thus the server module appears as in Figure 4.2.

For communication from the MCU to the FPGA, the MCU readies the data to be sent and asserts the Req line, and when the FPGA asserts the ACK line, the data has been

successfully received. The MCU then de-asserts REQ and removes the data from the bus and waits for the FPGA to de-assert the ACK line signifying that it is ready to receive again. If the MCU has more data to send, this process is repeated. A waveform diagram of this handshake process is shown in Figure 4.3.

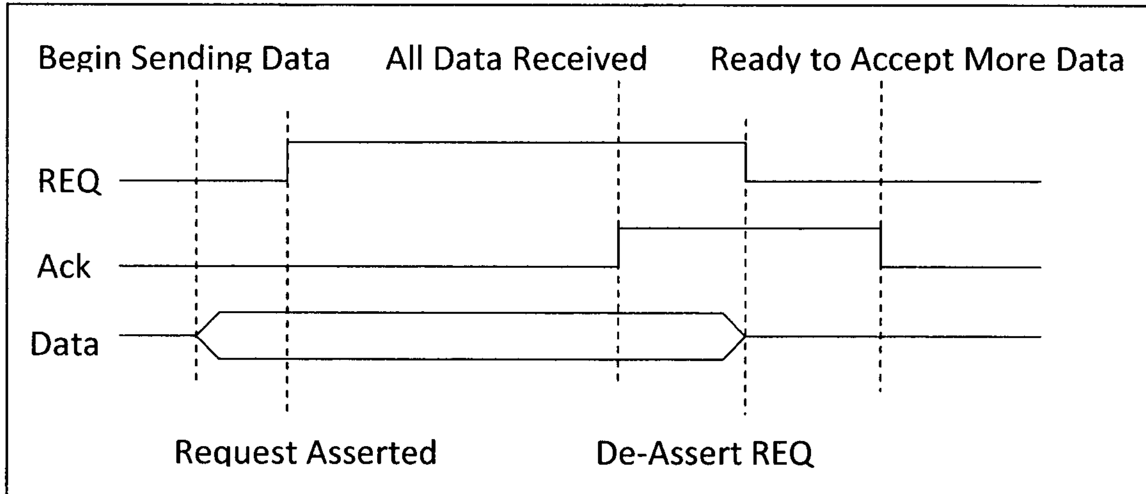

Figure 4.3. Waveform of Data Transfer From MCU to FPGA

For communication from the FPGA to the MCU, the FPGA readies the data to be sent and asserts the ACK line. When the MCU has received the data, it asserts the REQ line and the FPGA de-asserts the ACK line and removes the data from the bus. The MCU will then de-assert the REQ line and the process is repeated if there is more data to send. A waveform diagram of this process is shown in Figure 4.4.

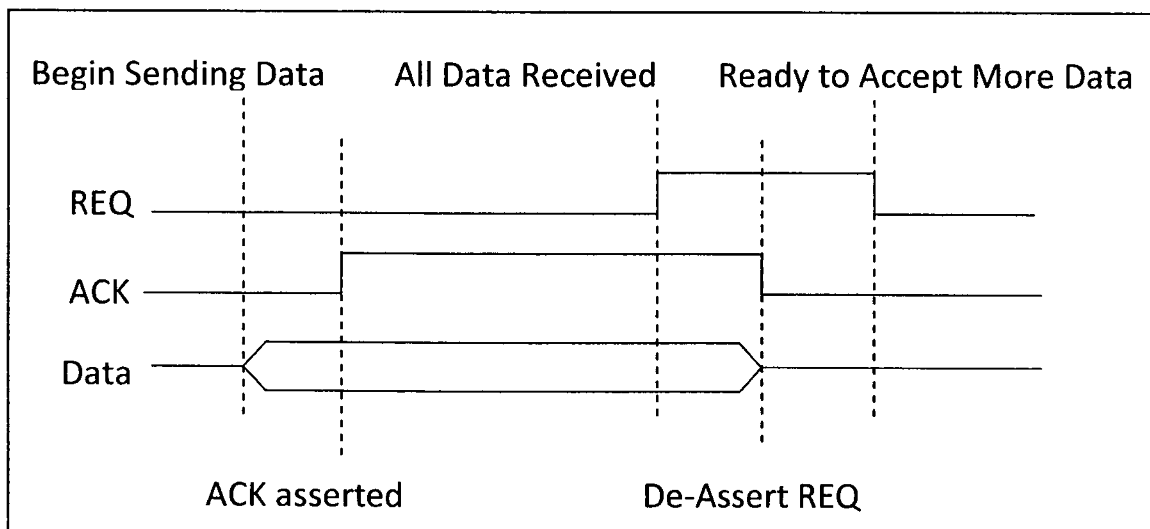

Figure 4.4. Waveform of Data Transfer From FPGA to MCU

4.2 VHDL Design – MCU_to_485 Module

The MCU_to_485 module, as stated previously, accepts the 8 bit wide data from the MCU, using a handshake protocol with an acknowledge and a request line, buffers it in a FIFO block and outputs it on the proper RS-485 bus, based on the high order 4 bits of the address byte of the packet. Thus it needs one module to accept the input from the MCU, one module to output the packet serially over the RS-485 connection, and one module to buffer the data. A block diagram of the MCU_to_485 module can be seen in Figure 4.5.

There is only a small amount of communication needed between the modules within the MCU_to_485 module. Between the MCU_buf module, which accepts the data from the MCU, and the FIFO module, which buffers all incoming data until it can be sent out; there is an 8 bit wide data bus, the FIFO_full signal and an enable line. When the MCU_buf module wants to send data to the FIFO, the EN line is asserted and the data is sent. When the difference between the point in memory that is being written to, and the point in memory that is being read from, is greater than 2048, which is one quarter of the FIFO memory, the FIFO_full signal is asserted. When it is asserted, the MCU_buf will not accept any new data until it is de-asserted. It can only be de-asserted by advancing the read address pointer which happens when the rs485_output module reads data from the FIFO to be sent to the test nodes.

Between the MCU_buf and RS485_output module there is only one line common to both and that is the WR /busy signal. When the rs485_output is ready to accept new data from the FIFO, it de-asserts the WR line and data is sent from the FIFO over an 8 bit wide bus to the RS485_output module. The WR signal is continuously checked by the MCU_buf module and when it is logic 1, the MCU_buf is able to output to the FIFO, when it is logic 0, the MCU_buf cannot write to the FIFO, thus ensuring that the same position in memory is not written to and read from simultaneously. The final signal that connects the modules together is the data_avail signal. The FIFO asserts this signal when the difference between the read and write address pointers is 32 or greater. This

signifies to the rs485_output module that there is data waiting in the FIFO to be sent out to the RS-485 network.

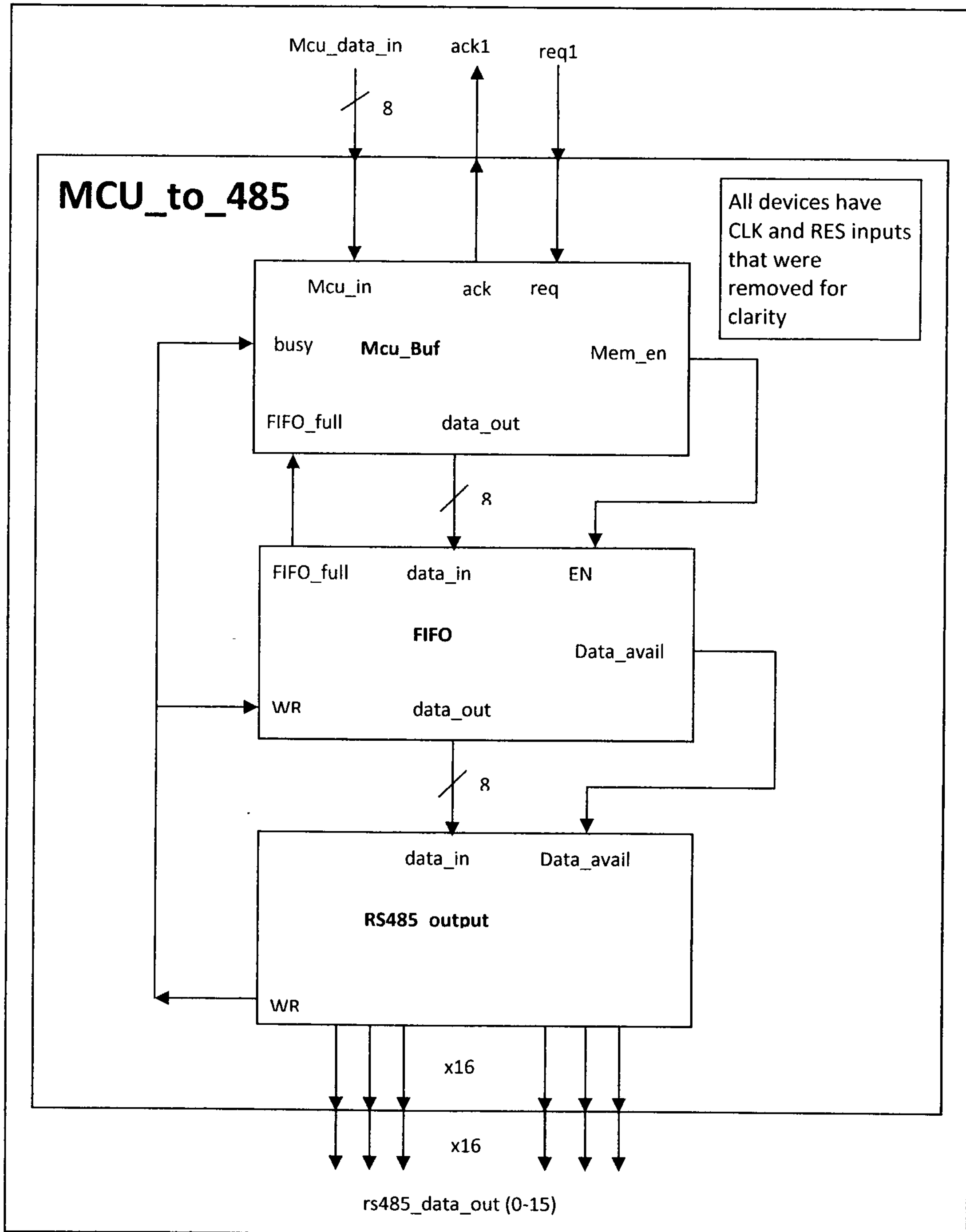

Figure 4.5. Block Diagram of MCU_to_485 Module

4.2.1 VHDL Design – MCU_buf Module

The MCU_buf module serves three main functions: communicate with the MCU through the handshaking protocol described previously, accept incoming packets one byte at a time, and write the packet, one byte at a time, to the FIFO module. It accomplishes this by utilizing 4 internal 32 byte long buffers that store one packet each. There are two internal pointers that indicate which buffer to use for reading in a new packet and for writing out the last packet to the FIFO, called switch and writing respectively. Switch is incremented when a packet is read in full from the MCU and writing is incremented when a packet is written in full to the FIFO. The module can thus both read in a new packet and write one to the FIFO at the same time. This functionality is desirable as the FIFO module can become unavailable for writing if it is being written to by the RS485_output module and it is desirable that communication with the MCU not be disrupted in such a case as the MCU has very limited time to output the full packet to the FPGA.

4.2.2 VHDL Design – FIFO Module

The FIFO module consists of a dual-port RAM megafunction, with storage of 8192 bytes, created by the Quartus II software to my specifications and a custom addressing, "wrapper" module that handles turning the RAM block into a FIFO block. This "wrapper" module allows the FIFO to have very simple inputs as all addressing is handled internally. The only input needed to write to the FIFO is to assert the EN line, and the only input needed to read from the FIFO is to de-assert the WR line.

The term FIFO describes the operation of the module, as FIFO is short for First In, First Out and thus the FIFO operates as follows. When a new byte of data is written to the module, the write_addr pointer is incremented by one and when a byte of data is read from the module, the read_addr pointer is incremented by one. Since both

pointers are initialized to 0 by a hardware reset, the first byte of data written in is the first byte of data read out, thus First In, First Out, or FIFO.

The FIFO module also has two status flags, data_avail and FIFO_full. Data_avail is asserted when there is a difference of 32 or more between the write_addr pointer and the read_addr pointer. This signifies that there is a packet that has been written to the FIFO that hasn't been read from the FIFO module. The FIFO_full flag is asserted when the difference between the write_addr pointer and the read_addr pointer is more than 2048. Since the total storage capability of the module is 8192 bytes (or 256 packets), this signifies that the FIFO is one quarter full. When this line is asserted, the MCU_buf module will not accept any new data until some data is read from the FIFO. This line was necessary to ensure that the data is not being written to the FIFO faster than it is being read from the FIFO.

As previously stated, the FIFO also contains a dual-port RAM megafunction that was created by Quartus to my specifications. The benefit of a dual-port RAM block as opposed to a single-port RAM block is that it can be written to and read from at the same time. This RAM module has two input ports and two output ports labeled port A and port B, and thus also has two address buses, two data input and output buses, and two enable lines. The RAM block was set up to write to the module only using port A and to read from the module only using port B. Thus the port B data input bus and the write enable signal (wren_b) can both be set to 0, and also since data is never read from port A, the data output of port A is disregarded. Finally, when data needs to be written to, or read from, the RAM module, the "wrapper" module simply sets the port A address to write_addr and the port B address to read_addr, respectively. The waveform for reading from the dual-port RAM block can be seen in Figure 4.6 and the waveform for writing to the dual-port RAM block can be seen in Figure 4.7.

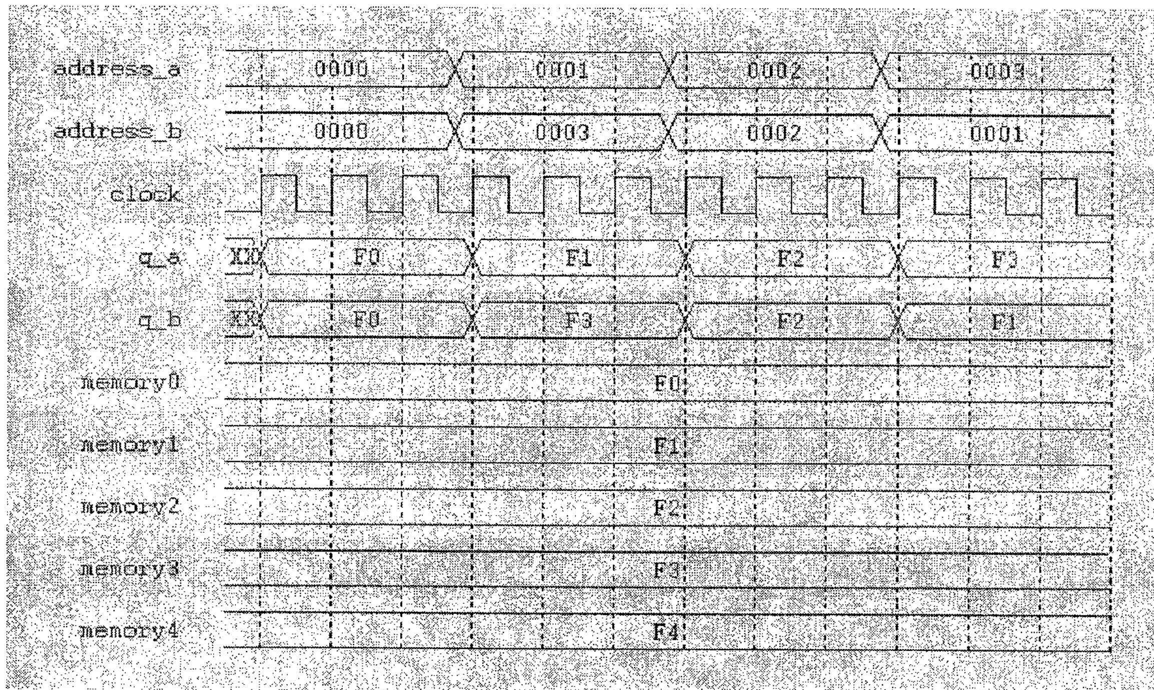

Figure 4.6. Waveform of Reading From Dual-Port RAM Module

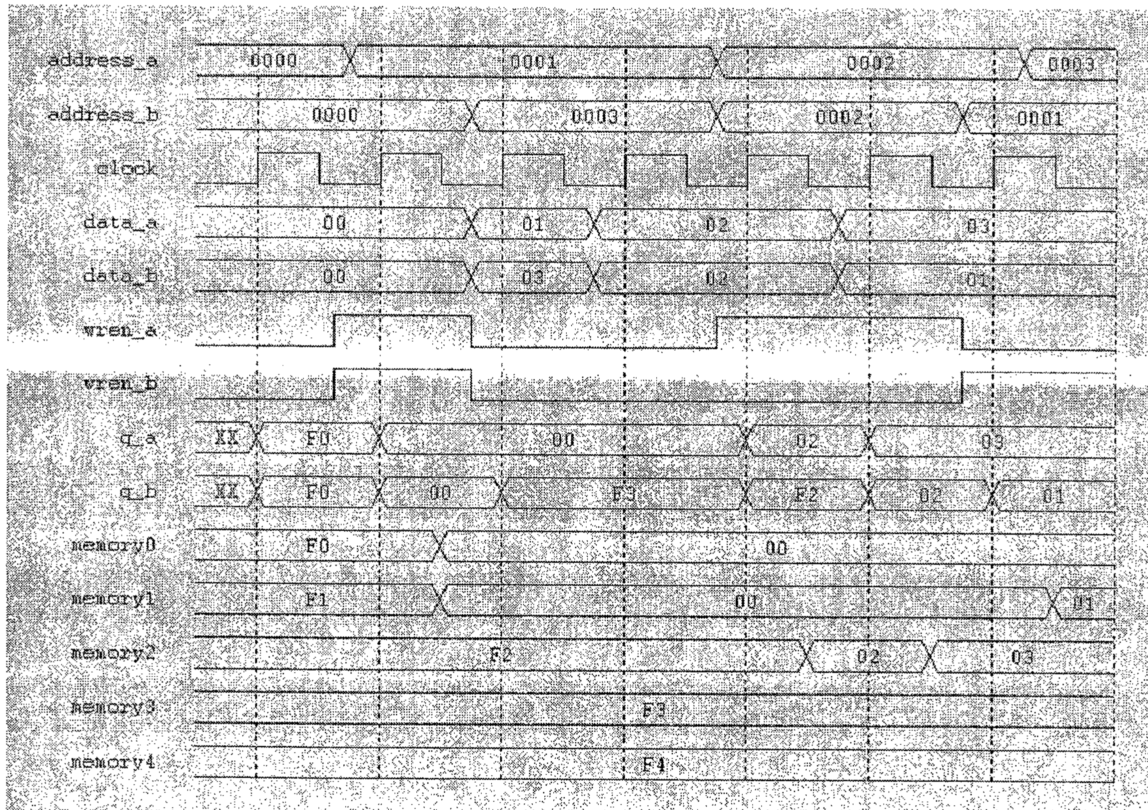

Figure 4.7. Waveform of Writing To Dual-Port RAM Module

4.2.3 VHDL Design – RS485_output Module

The RS485_output module serves two main functions, to read in a packet of data from the FIFO, one byte at a time and to write this data serially at a 100K baud transmission rate, to the RS-485 bus that matches the high order 4 bits of the address byte of the packet being sent out. Its operation is thus quite similar to the MCU_buf module; however, the RS485_output module performs its function with only one 32 byte internal buffer instead of four.

The RS485_output module first reads in the packet from the FIFO module by de-asserting the WR line and storing each byte as they arrive in the correct place in the output buffer (named outbuff in the VHDL code of the module), it then switches to the sending state of the module. In the sending state, since the system clock is running at 25MHz and the output Baud rate is 100K Baud, the clock needs to be divided by a factor of 250. After the clock is divided, the module examines the high order 4 bits of the first byte of the packet to determine the correct bus to output the data to, it then writes the packet data serially to that particular output port of the module at the correct transmission rate. It should be noted that even though the system is outputting at 100K Baud, the system does not have a data throughput rate of 100Kbps (bits per second) as there is a start bit of logic 0 and a stop bit of logic 1 and the start and finish of each byte, respectively. Thus the effective throughput of the system is actually 80Kbps.

4.3 VHDL Design – RS485_to_MCU Module

The RS485_to_MCU module functions very similarly to the MCU_to_485 module, but it does the reverse task. The RS485_to_MCU module accepts serial data over the RS485 test node network, saves it as a 32 byte wide packet, and buffers it temporarily in a FIFO Ram buffer until it can be transmitted back to the MCU using the same handshake protocol as the MCU_to_485 module, as described in Section 4.1. The

RS485_to_MCU module, however, is made up of a significantly larger number of smaller modules as compared to the MCU_to_485 module. Contained within the RS485_to_MCU module are a module for outputting to the MCU, named MCU_output.vhd, a FIFO module named RS485_fifo.vhd, 16 modules named RS485_buf.vhd which buffer the serial data from each of the 16 RS-485 buses before outputting that data to the FIFO, and finally a module named arbiter.vhd which handles the arbitration between the 16 RS485_buf modules and the RS485_fifo module. A block diagram of the RS485_to_MCU module can be seen in Figure 4.8.

As there are more modules making up the RS485_to_MCU module, there is also more communication between these modules as compared to the MCU_to_485 module. Between the RS485_fifo module and the MCU_output module, there is an 8-bit wide data bus for transmitting data from the FIFO to the MCU_output module, a data_avail signal to indicate to the MCU_output module that the FIFO has at least one unread packet stored within, and a WR signal that is held high by the MCU_output module until it is ready to read data, upon which the signal is de-asserted and the FIFO outputs the packet data one byte at a time to the MCU_output module.

The communication between the RS485_buf modules and the arbiter is slightly more complicated. Each RS485_buf module has a data_avail line output and a busy input connected to the arbiter. When the RS485_buf module has received a complete package from the RS-485 network of test nodes, it asserts the data_avail line. The arbiter will assert the busy line for that particular RS485_buf module when it is its turn. The arbiter is also watching the status of the WR input line to the FIFO and when it is de-asserted to logic 0, the arbiter will not allow any RS485-buf module to communicate with the FIFO. This is to ensure that it is impossible for the same address to be written to and read from at the same point in time. The arbiter is a fairly simple rotating priority arbiter, and its operation is described more fully in Section 4.3.3.

The last inter-communication between the modules within the RS485_to_MCU module is between the RS485_buf modules and the RS485_fifo module. Each of the RS485_buf modules has a tri-state logic mem_en line that is asserted to logic 1 when it

wishes to write to the FIFO and is in a high-impedance state otherwise. This allows all mem_en lines to form a tri-state bus that connects to the EN input on the RS485_fifo module. The RS485_buf modules also have an 8-bit wide, tri-state logic data_out bus

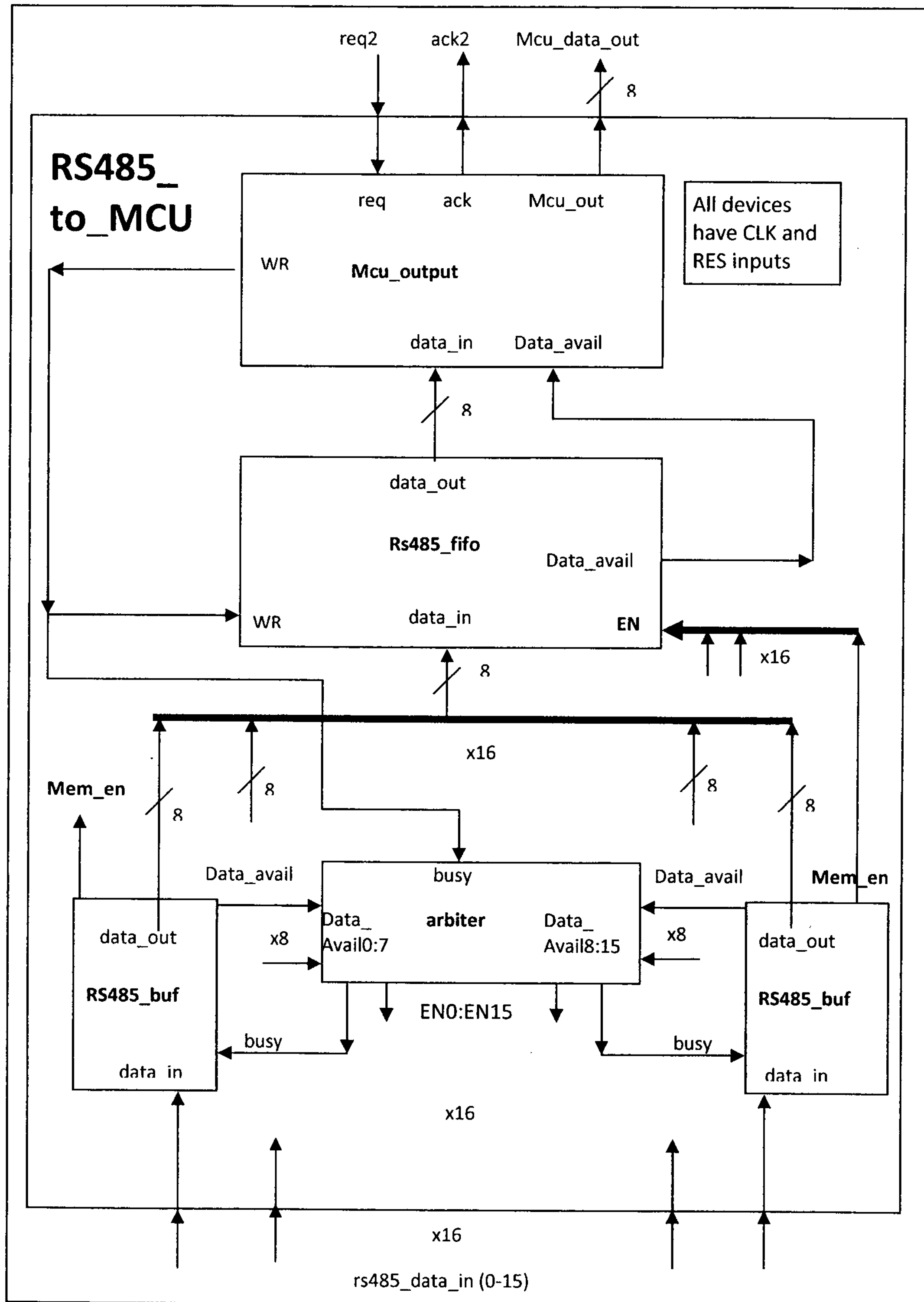

Figure 4.8. Block Diagram of RS485_to_MCU Module

that is used for sending data to the RS485_fifo module and is set to a high-impedance state when not in use. This allows the data_out buses from each RS485_buf module to form a tri-state bus going to the rs485_buf module.

4.3.1 VHDL Design – MCU_output Module

The MCU_output module has two main functions, to read data from the RS485_fifo module and to output this data over the 8 bit wide data bus to the MCU using the handshaking protocol previously described. When the MCU_output module is not in its sending, or output, stage, it is checking the data_avail flag from the RS485_fifo module, if the flag is high, data is waiting to be read and the module goes into its read state. For as long as the module is in its read state, it holds the WR line to a logic 0 and accepts the data from the RS485_fifo module one byte at a time. When the module has accepted an entire packet from the FIFO, it switches into its sending state. While in its sending state, the module asserts the ACK line and places the data onto the output bus (named data_out in the VHDL code). When the MCU asserts the REQ line to say that it received the data, the send_count signal is incremented and ACK is de-asserted. The module continues this operation until send_count reaches 32, at which point it stops asserting ACK and switches out of its sending state and begins checking for the data_avail flag again.

4.3.2 VHDL Design – RS485_fifo Module

This module functions exactly the same as the FIFO in the MCU_to_485 module, with a few exceptions. The first difference is that the size of the RAM in the RS485_fifo module is only 1024 bytes, or 32 packets. This RAM block can be smaller because data will be coming in much slower over the RS485 connection as compared to the

connection between the MCU and the MCU_buf module. The second difference is that this module does not have a FIFO_full signal. The reasoning for this is the same as for why the RAM block can be smaller; since the data is coming in much slower, the data can be read out of the FIFO fast enough that the possibility of overflowing the FIFO is very low.

4.3.3 VHDL Design – Arbiter Module

The arbiter module is a fairly simple rotating priority design that handles the arbitration of the rs485_buf modules in their communication with the rs485_fifo module. The module has 16 data_in inputs, one from each rs485_buf module, which is used to tell the arbiter that the particular buffer in question has data waiting to be sent to the FIFO module. There are also 16 EN outputs from the arbiter that connect to each of the rs485_buf modules to tell that particular module that it may now speak to the FIFO module. There is an internal 16 bit array, called en_array, which has only one bit set to logic 1 at a time; it is used as a pointer to keep track of which buffer is the one to speak to the FIFO. When a full packet has been sent to the FIFO or if the buffer module that the en_array is pointing at has no data to send, then the en_array is rotated left by one bit to allow the next buffer to speak. The operation of the arbiter in flowchart form is shown in Figure 4.9.

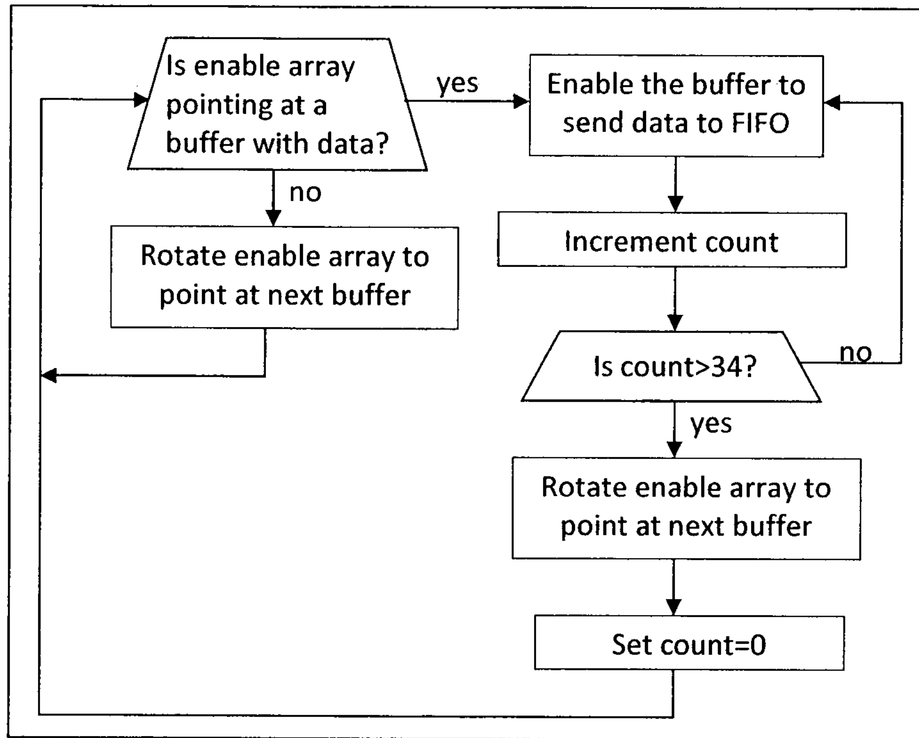

Figure 4.9. Flowchart of Arbiter Operation

4.3.4 VHDL Design – RS485_buf Module

The RS485_buf module is one of the more complicated modules in the system; it contains a small 128 byte (4x32 byte packet capacity) two-port RAM block megafunction created by Quartus II to my specifications, and a "wrapper" that controls both the RAM block and the module as a whole. The module functions similarly to the MCU_buf module, there are two states the system can be in, it is either waiting for, or accepting, a packet over the RS-485 connection or it is sending the packet, one byte at a time to the rs485_fifo module.

When the rs485_buf module is waiting for a packet to begin transmitting over the RS485 connection, it is waiting for the line to be driven low. This signifies the start bit of the packet transmission, and the modules clock is synchronized to the moment it arrives so that it is ready to accept each bit of the packet every 250 clock cycles after that. The module uses a running "average" to determine what each bit of the transmission is. It does this as the line may have some noise present and there may be a

certain amount of "bounce" to the signal. The module, obviously, must not confuse a temporary fluctuation of the signal from 0 to 1 or from 1 to 0, with the actual value; thus, for every bit of the transmission, the module samples the input signal 10 times and determines the value of the bit based on the average of these ten values. Once a byte of data is received, the packet byte count is incremented by 1 and the byte is stored in the internal RAM module. When the entire packet has been received, the system switches to its sending state and awaits the signal from the arbiter to begin sending to the rs485_fifo module.

When the arbiter signals to the buffer module that it may begin sending the data to the FIFO, it starts pulling the data out of the internal RAM module and sends it to the FIFO module. The timing is a little more complicated as compared to the MCU_buf module as the delay of the internal RAM block and the delay of the FIFO module need to be taken into account; both of these RAM modules have latched inputs, and thus the signals need one clock period before the data is actually saved.

4.4 VHDL Design - Server_Test Module

It quickly became apparent that to simulate or test the VHDL code for the server module, a test bench "wrapper" module would need to be created. This test bench had to simulate the server module being connected to the MCU. Without the test bench simulating the connection to the MCU, nothing would happen with the server module as there would be no REQ signals or data coming in to start the server module running.

The server_test module has two main states, the first is used to send packets to the server module and the second is used to accept data from the mcu_out bus of the server module. For physical testing purposes, the packets that were sent to the server module would instruct the test node to turn on the external power relay for the motherboard power and then to turn off the relay. These packets were used as the results of their successful operation are very obvious to the naked eye. Not only can the

motherboard clearly be seen to have power, but the relay "clicks" when it is toggled on or off.

The second state of the server_test module simulates the REQ signal response the MCU would provide if the server module were to send data to it. Nothing is done with the data that the server module outputs, but without toggling the REQ signal, the mcu_output module would simply freeze until the signal is asserted.

Chapter 5. Development of C Code for Test Server Embedded Microcontroller

As described in Chapter 3, this project requires the use of a microcontroller with built-in Ethernet capability. The microcontroller that was chosen for this project is made by Microchip and is part number PIC18F97J60. The PIC18F97J60 microcontroller has the following specifications [11]:

- 100 Pin microcontroller with 70 I/O pins
- 128Kbytes of Flash Program Memory
- 3808 bytes of SRAM Data Memory
- 2.778 to 41.667 MHz Clock Speed
- 5 Timers
- 4 External Interrupt Pins
- 2 Enhanced USART modules
- 10-Bit, 16 Channel A/D Converter Module
- IEEE 802.3 Compatible Ethernet Controller that Supports One 10Base-T Port
- 8-Kbyte Transmit/Receive Packet Buffer SRAM

The functionality of the MCU needs to be broken into small, manageable sections of code so that the MCU has enough time to handle all processing that is required of it. Thus the MCU, after initialization procedures have been completed, enters an infinite loop that contains all processing steps that it must complete in each cycle of the MCU code. This infinite loop contains the following functions which will be described in more detail in the following sections: StackTask and StackApplications, which were part of Microchip's TCPIP Stack software and are required to handle communication over the Ethernet connection, BerkeleyTCPServer which was developed by Microchip and then altered to provide the needed functionality of this project. This loop must also execute CRC16, which is used to calculate the CRC check value, FPGA_transmit, which handles

sending data to the FPGA using the previously described handshaking protocol, FPGA_receive, which receives data from the FPGA and checks the CRC value, and finally a routine to check the MCU board IP address and display it on the development kit LCD display.

5.1 Initialization Procedures

The initialization procedure for setting up the MCU is quite a long procedure that involves initializing 5 separate aspects of the MCU. The first initialization is contained within a function that was part of the Microchip TCPIP Demo App and is used to configure settings on the MCU board. The function is called InitializeBoard() and is used to initialize the LEDs on the MCU board, the Phase Lock Loop (PLL) for the main system clock, the USART module, and the Baud rate for transmissions. This function also enables interrupts and sets up the external memory chips. The second initialization is contained within the function Initialize() and is used to set up the data direction on ports B, C and D which are used for the ACK and REQ signals, outputting to the FPGA and accepting inputs from the FPGA, respectively. The next initialization is to set up the LCD display on the development kit board and is used to display the version of Microchip's stack software currently in use as well as the IP address of the MCU.

The remaining initialization procedures were part of the TCP/IP stack software provided by Microchip, and perform several key tasks to set up the TCP/IP stack. TickInit initializes the Tick.c functions which provide the TCP/IP stack with a large scale timer capable of keeping track of anything from a few microseconds to a few hours. InitAppConfig initializes stack and application variables which are necessary for stack operation. These include the IP address, DNS Server values and SNMP server initialization. The next process initializes the development kit reset buttons, and the final function, StackInit, initializes the core stack layers such as MAC, TCP, UDP and application modules such as HTTP and SNMP.

5.2 Main Processing Loop

The main processing loop of the MCU C code executes the core tasks given to the MCU. The main loop handles accepting and sending packets over the TCP/IP socket to the PC client. It also communicates with the FPGA, both sending packets to the FPGA and accepting packets from the FPGA. This communication is using the handshaking protocol described in Chapter 4. All CRC check values are calculated in the main loop, both before sending to the test nodes, and upon receipt of a package from the test nodes to ensure that no data corruption occurred during the transmission from the node to the server. The main loop must also handle all other TCP/IP stack related tasks as required to allow the Ethernet communication to function properly. A flowchart of the main processing loop operation is shown in Figure 5.1.

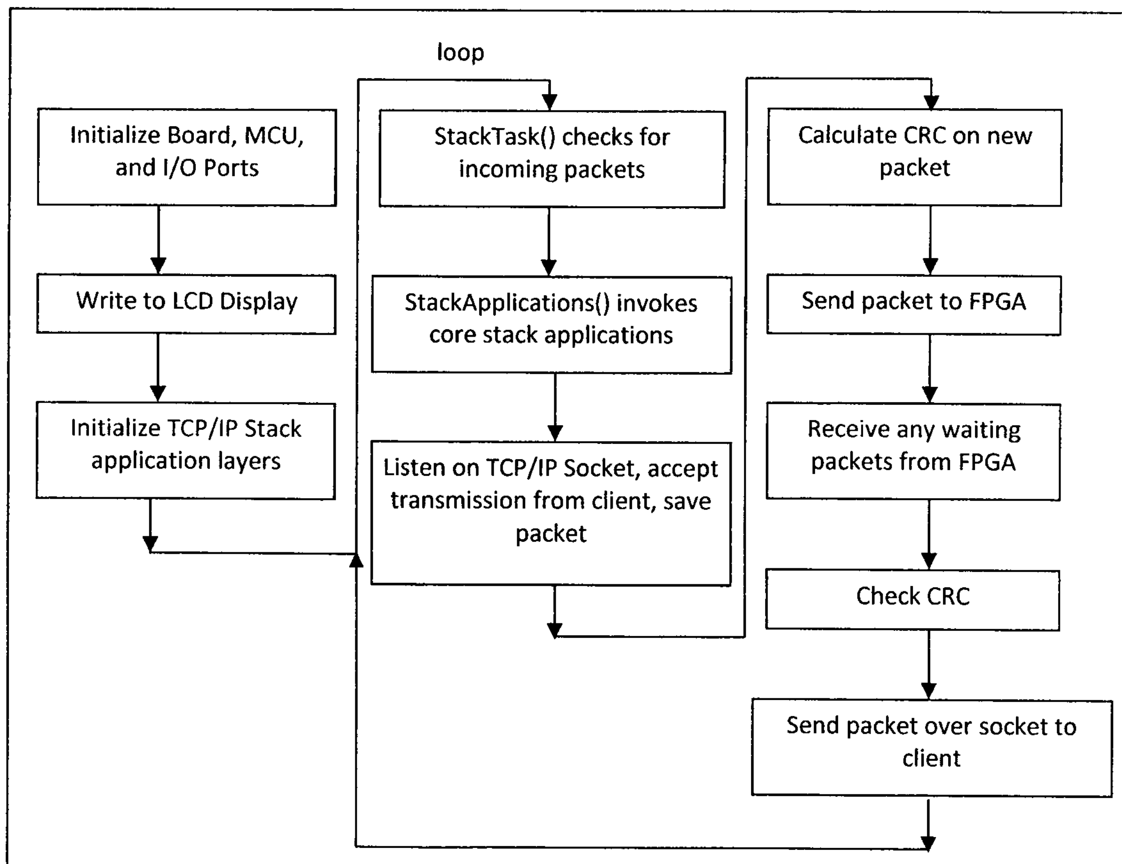

Figure 5.1. Flowchart of MCU Main Processing Loop Operation

5.2.1 Stack Operations

The first function calls of the main MCU processing loop are functions that came as a part of Microchip's free TCP/IP software: stacktask() and stackapplications(). The stacktask() function is contained within StackTsk.c file that came with the TCP/IP stack. When stacktask() is called, the function fetches a packet and throws away any old ones, and then transfers the packet to the appropriate handler to begin processing the various layers that are added to a data packet as it passes through the TCP/IP protocol to the MAC, and then onto the Ethernet connection. This function also checks the incoming address on the packet to ensure that it is meant for this particular IP address.

The stackapplications() function is also found in the StackTsk.c file and is used after initializing the stack with the StackInit() function. Stackapplications() loads all application modules that the user chose to include in the TCPIPConfig.h header file which was created by Microchip's TCP/IP Configuration Wizard program. These application modules include, but are not limited to, the HTTP server application, FTP server application, SNMP server application, Telnet server application, Reboot server application, and several others.

5.2.2 Accept and Store Packet

After initialization routines have been executed and all necessary stack operations have been taken care of, the next function that the MCU needs to perform is to accept the data packet from the client over the TCP/IP socket. Two functions are needed for retrieving the data packet, storing the data array and calculating the CRC16 value titled BerkeleyTCPServer()and crc16(). The BerkeleyTCPServer() function was part of Microchip's free TCP/IP stack software, though its functionality was to merely echo the data received back to the client. Thus several changes needed to be made so that the function did not merely echo the recently arrived data, but instead stored it in

memory, and also, so that the data sent back to the client is not only an echo of the received data, but is also any data that was received from the FPGA on the previous cycle that is waiting to be sent back to the client program. Since the data sent over the TCP/IP socket is raw binary data, no type conversions are needed before the data can be processed. Finally, the crc16 function does exactly what its name implies, it is sent the packet one byte at a time and it calculates the 2 byte long CRC-16 value of the packet and adds it to the end of the packet.

5.2.3 Communication with FPGA

Two functions are required for communication with the FPGA: FPGA_transmit() and FPGA_receive(). FPGA_transmit() handles the handshaking protocol needed to send data to the FPGA one byte at a time. Since FPGA_transmit requires the CRC value, before FPGA_transmit() can be called, the CRC needs to be calculated by the crc16() function. After a packet has been transmitted to the FPGA, FPGA_receive is called to see if there is any reply packet data waiting to be sent from the FPGA to the MCU, if there is, it applies the previously described handshake protocol and accepts the data one byte at a time. The function then internally calls crc16() to calculate the CRC value, compares it to the one that is attached to the packet, and if they don't match, it overwrites the ACK value of the packet with 170 in decimal (or 10101010 in binary). If the ACK line from the FPGA is not asserted upon immediate execution of the function, it will wait for a count of 256 before leaving to execute the MCU's other functionality.

Chapter 6. PC Client Design

This section details the design process and methodologies used for creating the PC client software that is running the entire system. The PC client is used to input the hex file of input packets, keep track of whether a node responds to a sent packet and re-send any packets that were not responded to. If there is a problem with the CRC, either when the node receives the packet, or when the test server does, the client code will re-transmit that packet. The PC client also sorts the outgoing packets to ensure that the same bus does not receive a packet in consecutive transmissions, thus reducing the likelihood of bus contention issues when the nodes reply.

The client code is written for, and on, a Windows-based PC, and thus if the end user wishes to use a different platform, some modifications to the client code are needed. These changes are outlined in Section 6.3. Also, the client is written for IPv4, and modifications will need to be made to the socket code if the user wishes to have the system be IPv6 compatible.

6.1 TCP Versus UDP

Though it was requested by AMD/ATI to use the TCP protocol to communicate to the test server, a comparison of TCP with its alternative, UDP is necessary for a full understanding of the TCP protocol and why it was chosen for this project.

The User Datagram Protocol (UDP) is a connectionless protocol, or in other words, it does not require a direct connection from client to server. With UDP, a single socket can send and receive packets to many computers without any handshaking taking place. The lack of handshaking means that the reliability of the transmission, the order of the data sent out and the data integrity are not guaranteed. Thus the data, or datagram, sent out using UDP can arrive out of order, appear duplicated or not arrive at

all [17]. The process for sending data over a socket using UDP is shown in Figure 6.1 and the process for sending data over a socket using TCP is shown in Figure 6.2.

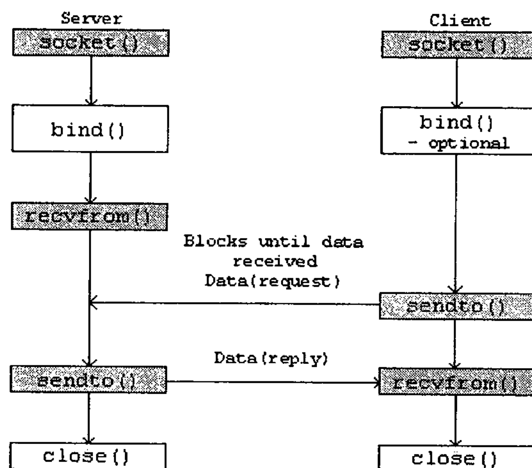

Figure 6.1. Flowchart of UDP Operation [17]

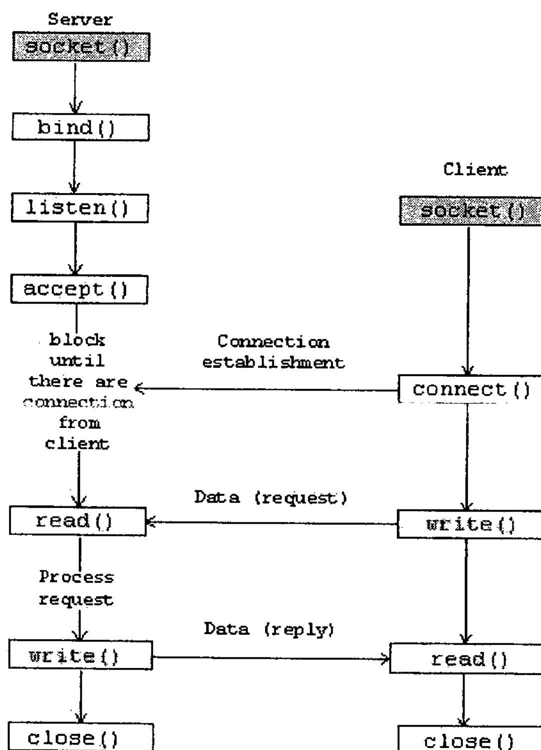

Figure 6.2. Flowchart of TCP Operation [17]

To effectively use UDP, the application must handle the task of ensuring good communication. This transfer of responsibility allows the header on a datagram to be

very small and thus it is more efficient [18]. An analogy to the UDP protocol would be the standard mail service. You can send out as many letters you want to as many places you want, but there is no guarantee they will ever arrive at their respective destinations, or the order in which they will arrive [19]. Therefore, due to the unreliability of the UDP protocol, the TCP protocol is the one chosen for this project.

6.2 PC Client Operation

The client program, which is in control of this entire test system, must perform several tasks. First of all, it needs to send the data packets over the TCP/IP socket to the microcontroller on the test server hardware. The client also needs to accept replies from the microcontroller over the socket, save the replies in an output file, and examine their contents. Specifically, the client program needs to examine the acknowledge byte of the packet to determine if there were any communication issues between the test server and the test node, this includes non-matching CRC value at the test node, non-matching CRC value at the test server and also non-existing command given to the test node. The pertinent ACK values are outlined in Table 6.1.

Table 6.1. Acknowledge Values and Their Meaning

Acknowledge Value		
Decimal	Binary	Meaning
255	1111 1111	CRC Matched, command was processed
0	0000 0000	CRC did not match at node
15	0000 1111	Command was unknown
240	1111 0000	Super-User condition invoked
3	0000 0011	Reset was triggered by node MCU timer
170	1010 1010	CRC did not match at server

The client program must also keep track of the time in which packets were transmitted, to know if a reply should have been received at a certain point in time. The client must keep track of whether a reply has been received for a particular packet that

was transmitted, and if the node cannot be reached by 4 transmission retries, the node is officially recorded as not being connected to the network.

The final task of the client is to sort the outgoing packets in the queue to ensure that the possibility of a bus contention on one of the test node buses is reduced as much as possible. This is handled entirely by the SortPackets() function in the client.c program and its operation is described more fully in Section 6.2.1.

The final thing that should be mentioned is that the client program was developed on a windows based PC, and thus some changes must be made to the code to get it to operate on a Unix/Linux based machine. These changes are outlined in Section 6.3.

6.2.1 Packet Sort Algorithm

The packet sort algorithm is very important to the functionality of the client code as it is the only measure in place that limits the possibility of bus contention problems with the test nodes. Though the nodes cannot initiate communication with the test server, per the communication protocol in use in this project, the delay in the response from the node is somewhat of an unknown. Thus if two consecutive packets were sent out to the same bus, let alone the same node, the two nodes could have their response communication overlap for a certain amount of time, causing a bus contention and possibly data corruption. The packet sort algorithm is thus in place to reduce this possibility by sorting the outgoing packets such that there is at least a three packet transmission delay between transmissions to the same bus. A flowchart describing the functionality of the packet sort algorithm is shown in Figure 6.3.

The packet sort function operates by keeping track of the last 4 buses that were communicated to, as well as, with the help of the function nodelastcommto(), the record of which nodes in the system have recently been communicated to. Any outgoing packet has its address field checked against the last 3 buses that were communicated to, and if it matches one of those three, it is swapped in the outgoing

packet queue with a packet that is destined for a different bus. The 4th last bus communicated to is kept for the nodelastcommto() function. If all nodes have been recently communicated to, the 0th node on the 4th last bus communicated to is sent a "check" packet, and the record of which nodes have been recently communicated to is re-initialized.

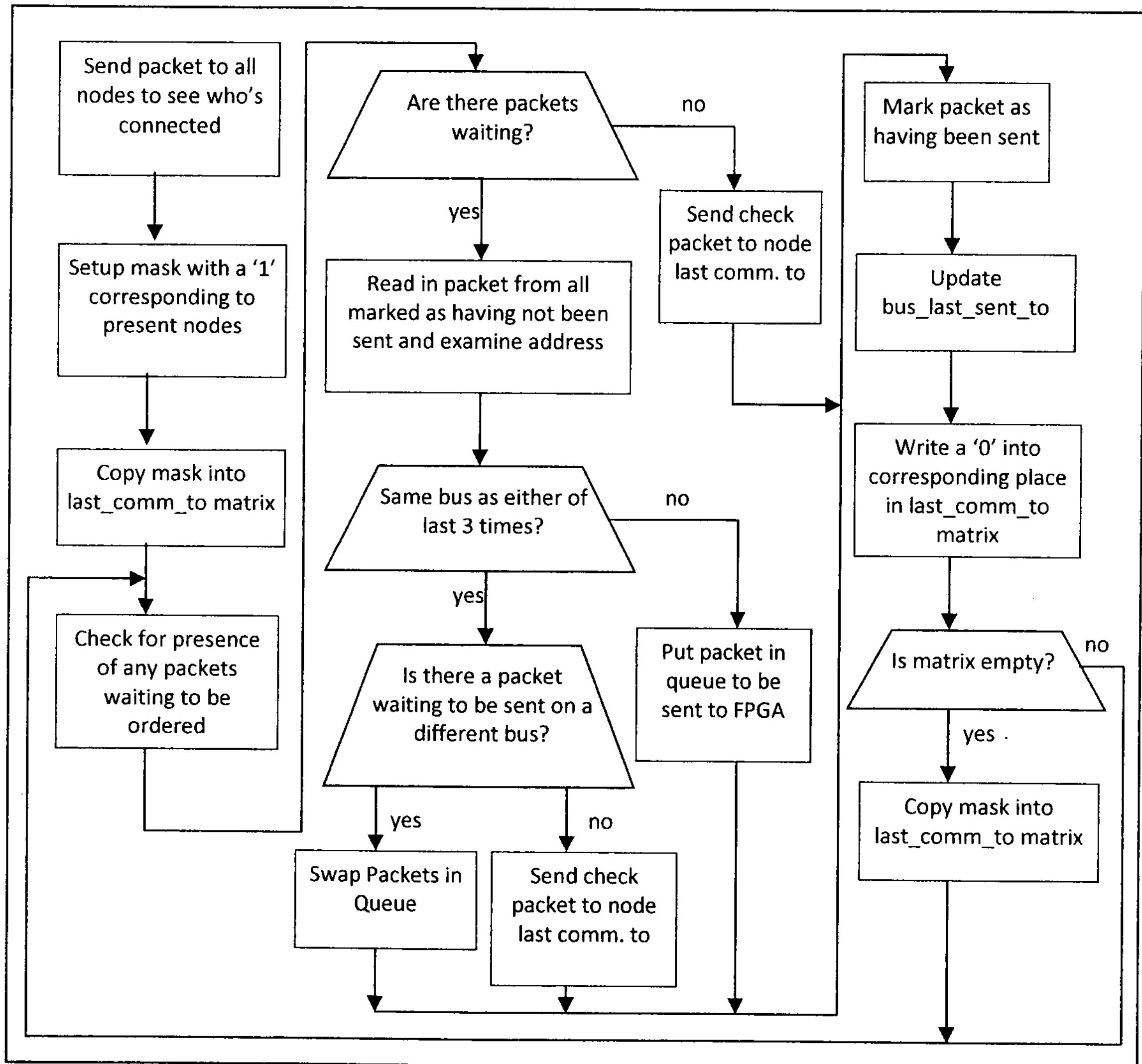

Figure 6.3. Flowchart of Packet Sort Algorithm

6.3 Changes to Adapt Client.c Code to Unix/Linux

The client code was written in C on, and for, a Windows-based PC environment. There are several changes that must be made to the code to get it to function properly in a Linux-based environment. The changes that must be made are only to the socket operations of the client, and are fairly simple to make. The differences between the two environments will be shown by performing a side by side comparison of the UNIX/Linux code to the code for a Windows system.

The first difference is in the header files of the code and is shown in Table 6.1. The include file for sockets in Windows is winsock.h, whereas for Unix, 4 different header files need to be included.

Table 6.2. Header Differences, Unix vs. Windows [19]

Unix	Windows
#include <sys/socket.h> #include <sys/types.h> #include <arpa/inet.h> #include <netinet/in.h>	#include <winsock.h>

The next difference is due to the fact that the winsock needs to be initialized using WSAStartup(). The associated code is shown in Table 6.2.

Table 6.3. Socket Initialization Differences, Unix vs. Windows [20]

Unix	Windows

int main(int argc, char *argv[])
{
Variable Initializations
 	
servIP=argv[1];
dataServPort=atoi(argv[2]);
	int main(int argc, char *argv[])
	{
	Variable Initializations
	WSADATA wsaData;
	
	servIP=argv[1];
	dataServPort=atoi(argv[2]);
	
	if (WSAStartup(MAKEWORD(2,0), &wsaData) != 0)
	fprintf(stderr, "WSAStartup() failed");

The last difference is in regards to the shutdown of the socket application. The side by side comparison of the code is shown in Table 6.3.

Table 6.4. Socket Application Shutdown Differences, Unix vs. Windows [20]

Unix	Windows
close(sock);	closesocket(sock); WSACleanup();
exit(0);	exit(0);

A more minor detail, but worth mentioning, is the difference in socket function error reporting. The side by side comparison is shown in Table 6.4.

Table 6.5. Socket Error Reporting Differences, Unix vs. Windows [20]

Unix	Windows

void DieWithError(char *errorMessage)
{
 	
perror(errorMessage);
exit(1);
}
	void DieWithError(char *errorMessage)
	{
	fprintf(stderror, "%s: %d\n", erorrMessage,
	WSAGetLastError());
	exit(1);
	}

All other socket functions perform in exactly the same way whether you are programming on a Windows or Unix machine. An alternative method exists, however, where conditional groups are used to make the code run on either system. This is implemented in the client.c code for this project. An example of this, executed with the socket header calls, is shown in Table 6.6.

Table 6.6. Conditional Group C Code Example - Header Calls

```
//If Windows system
#ifdef WIN32
        #include <winsock.h>
//if Linux/Unix system
#else
        #include <sys/socket.h>
        #include <sys/types.h>
        #include <arpa/inet.h>
        #include <netinet/in.h>
#endif
```

Chapter 7. Conclusion

This thesis described the design and construction of the test server for a custom automated test environment that is capable of testing up to 256 motherboard/GPU test nodes. The test server hardware consists of an Ethernet-enabled MCU, an Altera Cyclone II FPGA and an RS-485 transceiver network. The system as a whole will consist of one test server which has the capability to have 16 buses of 16 test nodes each, connected to it. A custom C client program was also created to provide the data packets that will be sent to the node network.

The basic functionality of the project is as follows:

1. Client software either creates, or reads from a file, a list of data packets, sorts them to reduce bus contention, and sends them over a TCP/IP socket to the MCU
2. The MCU converts the data packet from a character string with numbers encoded in hexadecimal to an array of integers and calculates a CRC-16 check value for the packet
3. The MCU adds the CRC-16 value to the end of the packet, making a total of 32 bytes and transmits the packet one byte at a time to the FPGA
4. The FPGA accepts the packet using a simple handshake communication protocol and buffers it until it is ready to transmit to the RS-485 node network
5. The FPGA will then transmit the packet over the RS-485 network to the appropriate node as given by the address byte of the data packet
6. The FPGA waits for a response from the node, accepts it when it arrives, and buffers the response packet until it can transmit it to the MCU
7. The MCU accepts the packet from the FPGA, checks the CRC value for any errors and transmits it back over the TCP/IP socket to the client program
8. The client accepts the response packet, if the ACK field checks out ok (i.e. The CRC was verified correct at both ends of the RS-485 transmission), it finds the

original packet and marks that a response has been received. If the ACK does not check out, the original packet is re-sent over the socket

9. The client then checks all of the original packets to see if there are some that never received a response. If there are, it will re-transmit those packets over the socket to the MCU

The first step in developing the project was to come up with a list of specifications for the system as a whole. These were provided, in large, by AMD/ATI, as the system will be utilized by them to functionally test their GPU's. From the specifications of the system, the components for the project could be chosen to meet those specifications. For the prototyping of the system, development kits were utilized instead of custom PCB layouts of the test server and test node boards, mainly due to the cost, both in terms of time and money, of getting custom PCBs manufactured. Once the components were chosen, the longest step in the process could take place. This consisted of designing and writing VHDL code for the FPGA test server module, designing and writing C code for the MCU, designing and creating the RS-485 transceiver board, designing and writing C code for the client program, and testing and verifying each of these components individually, and together as a system.

This version is the second iteration of the project, and was designed to correct the shortcomings of the original project, as well as expand its capabilities. There were several problems with the first version of the project, and the choice of components, transmission media and protocols has successfully dealt with these issues. The choice of a differential signaling communication standard (RS-485) removed the inherent galvanic isolation problems with the first generation of the project. The design of a custom test server using low-cost components has reduced the high cost and complexity of the off-the-shelf digital I/O board, which was in use for the first generation, as well, the system flexibility has been greatly improved. Finally, the choice of a daisy-chain network topology reduced the amount and length of cabling that is needed to implement the full network.

7.1 Test Server Limitations

One final, important consideration is in regards to the inherent limitation of the number of test nodes connected to the test server. Though the packet structure only allows for one byte of data for the address of the test node, and thus only 256 addresses are available, if the final user of the system wished to add more nodes to the test server, the packet structure is not the only portion of the project that would have to be altered. The nature of the communication between the FPGA and the MCU would likely also need to be re-examined if there were more nodes on the network. This is due to the fact that the acknowledge line from the FPGA that lets the MCU know that there is data to be sent back to the client is not on an interrupt pin of the MCU. The MCU merely polls the ACK2 pin once every iteration of the main processing loop, waits for a count of 256, and if the ACK2 pin is not asserted, it leaves the function and does other tasks. If the number of nodes was increased beyond 256, however, this would likely have to be altered in some way, most likely by placing the ACK2 pin on an I/O port of the MCU with interrupt capability. As the number of nodes increases, the number of packets that need to be sent back to the client will also increase, and at some point the polling functionality of the MCU will not be fast enough to service the number of requests to send data. This would likely cause the loss of data that is intended to be sent back to the client. Thus, if the end user wishes to increase the number of nodes under test, the easiest method of accomplishing this is to simply add a second test server.

7.2 Thesis Contributions

The final deliverable of this project is a novel solution to a complicated problem. Though the idea of an automated test environment is not a new one, they have been around in one iteration or another for almost 50 years, the implementation of a low-cost, fully programmable and upgradeable system to test large batches of ASICs is not frequently seen, at least in the public domain. Though the implementation of the test

server did not use any methodologies that were new, the combination of using a microcontroller with built in Ethernet capability to communicate through a custom UART module implemented on an FPGA to a network based on the RS-485 electrical specification is entirely new.

Based on the relatively small availability of different Ethernet-enabled microcontrollers, the market penetration of Ethernet microcontrollers in industry is still fairly small. Thus, the use of one in this project, as opposed to en external Ethernet controller, or some other communication medium all together such as USB, or even wireless communication like Bluetooth or something similar, is also something that has not been done much in the past.

By getting this system on-line, and by allowing the nodes to be remotely upgradeable through the bootloader program, this project has also almost completely removed the need of having a person in the same room as the running test network. Once the system has been "plugged in", the system will only need hands on attention if one of the nodes detects a critical failure that leads to out of control temperature increases or voltage swings.

The final goal of this system, however, where the test network is completely configurable, giving it the ability to functionally test any piece of electronic equipment, is the most novel idea presented in this thesis. Much of the design of such a system is performed here, though future work is definitely required. For instance, the hardware of the test server, and by that I mean the MCU, FPGA and transceiver network, does not care what type of data passes through it. The only pertinent fact that the test server is concerned with in relation to the data packets, is their length in bytes. The only change that is needed to the test server to get it to control any kind of test node is to alter the "babysitting" portions of the client code. These portions of code are the ones that verify that responses were received from the nodes and that the responses, and acknowledge fields in the responses, are formatted properly.

7.3 Test Server Verification and Final Results

As of this writing, all C and VHDL code has been completed and several stages of final testing have been performed. The PC client software has been tested extensively communicating with the MCU and the communication over the TCP/IP socket to the MCU is functioning properly and all "babysitting" logic of the client software is functioning properly. In addition to this, the RS485 transceiver prototyping board has also been extensively tested and functions flawlessly. The VHDL code running on the FPGA has been simulated and functions properly, and when using the test server test bench (server_test.vhd) implemented on the FPGA, data is being transmitted over the RS485 network at the proper Baud rate to the test node, though some bugs need to be worked out of the system. Finally, the MCU C code is functioning properly, though the development kit has a shortage of available input/output pins that are not tied to other functions implemented on the board. This causes problems with the communication to the FPGA and also needs more work if it is to function at full capacity.

7.4 Future Work

Future work on this project could move in three different directions. The first would be to make a more user-friendly front-end client program with a graphical depiction of the status of the test nodes, the second would be to design the system to be totally auto-configurable, allowing it to test any electronic system, instead of the motherboard/GPU test node that is implemented in this version. Finally, the last area of future work would be to implement functionality to the test server such that the test server MCU could update the test server FPGA hardware.

As of this writing, the user interface of the client program is extremely simple. The end-user simply creates a text file of data packets, written in hexadecimal, that they would like executed by the test nodes and then feeds this file into the client program.

The client program is then automated to handle outputting these packets to the test network. Unfortunately, with this setup, there is no easy way to suddenly add a new packet to be transmitted to the test node network.

The client program would also benefit from a graphical depiction of the status of the test network. This could be executed by having the client render a graphical depiction of the network on the screen, with appropriate displays of temperature, voltage and peripherals available for each test node, as opposed to outputting the received packets from the test network to a text file that then must be interpreted.

The second area of future work with this project, turning the system into one that is configurable by the end user, is a much more ambitious undertaking, and should be the ultimate end goal of a project such as this. The expansion of this system to one that is completely configurable to allow it to test any electronic system was described briefly in the previous section on thesis contributions, but more discussion on the topic is needed here. As stated in the above section, the hardware of the test server does not examine the contents of the packets that are passed through it, and thus any data can be transmitted over the network to the test nodes. The only pertinent information that the server hardware examines, and in fact, depends on, is that the length of the packets is 32 bytes long. The ability to include any information in the packet, in any format, means that any test node can be plugged into the network and the information passed to the server by the client will, with absolute certainty, be passed along to the test node.

The only changes to this system that would be needed, to accompany a change to the test node, are to the client software. The client software is continuously examining the second byte of the received packets to see if the acknowledge byte is an acceptable value, and if it is not, it re-transmits the packet. Also, in this configuration of the system, some commands are known to take more time to execute, for instance, the process of turning off the motherboard by using the opto-coupler can take several seconds to execute, and thus the client software needs to give it enough time to execute before it tries to communicate with that node again. Thus, if a test node was introduced to the test server that had packets with the acknowledge byte as the third

byte, or if some functions of the test node required large amounts of time to execute, the client software would have to be updated to include these new specifications.

One way to implement such a configurable system is to have a file that is loaded upon initial client start-up with all pertinent node specifications included within. This file would instruct the client program as to the packet structure and to such special cases as the ones mentioned above where a node needs a long period of time to perform a function.

The final area of future work could potentially save much time in the future incarnations of the project, as well as help aid this system to be more auto-configurable. This area is to provide the test server MCU with the ability to update the hardware on the test server FPGA. A new command would have to be created in the packet structure command set that would tell the MCU that the incoming packets are intended for re-writing the hardware currently installed on the FPGA chip. An external RAM would likely also be necessary to store all the reconfiguration data until it is all received at the MCU as the file could potentially be quite large. The MCU would thus also need the ability to interface with the programming pins of the FPGA to place it in programming mode and to allow the new data to be passed into the FPGA.

With these areas of future work set aside, the system in its current configuration has full functionality and performs to all specifications outlined to us by the end-user, AMD/ATI of Markham. The system also includes enough expandability and flexibility to provide them with the ability to test just about anything that can communicate with digital logic circuitry.

References

1. **IEEE.** Automated Testing. *Aeropsace and Electronic Systems Magazine, IEEE.* October 2000, Vol. 15, 10, pp. 125-130.
2. **United States Department of the Navy.** *U.S. General Accounting Office, Staff Study, Versatile Avionics Shop Test (VAST) System AN/USM-247.* Washington, D.C. : United States General Accounting Office, February, 1973.
3. **National Instruments.** Modular Instruments. [Online] National Instruments. [Cited: September 22, 2009.] http://www.ni.com/modularinstruments/.
4. **Teradyne.** Semiconductor Test. [Online] Teradyne. [Cited: September 22, 2009.] http://www.teradyne.com/flex/FLEX.html.
5. **Drenkow, G.** Future Test System Architectures. *Instrumentation & Measurement Magazine, IEEE.* August, 2005, Vol. 8, 3.
6. **Perrin, B.** The Art and Science of RS-485. *Circuit Cellar.* [Online] July 1999. [Cited: September 25, 2009.] http://www.circuitcellar.com/library/ccofeature/perrin0799/index.asp.
7. **Microchip.** Ethernet Solutions with Integrated MAC and PHY. *Microchip.* [Online] [Cited: September 23, 2009.] http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=2504.
8. **Freescale Semiconductor.** MCF532X: V3 ColdFire Microprocessor with LCD driver, Ethernet, USB and CAN. *Freescale Semiconductor.* [Online] [Cited: September 23, 2009.] http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MCF532X&fsrch =1.
9. **Digi International.** NET+ARM Microprocessors. *Digi International.* [Online] [Cited: September 23, 2009.] http://www.digi.com/products/embeddedsolutions/microprocessors.jsp.
10. **Atmel.** AVR32 32-bit MCU. *Atmel.* [Online] [Cited: September 23, 2009.] http://www.atmel.com/products/avr32/uc3/uc3_2.asp?family_id=682.
11. **Microchip.** *PIC18F97J60 Family Data Sheet.* U.S.A. : Microchip Technology Inc., 2008.
12. **CCS inc.** 3.3V Ethernet Controller Development Kit. *Custom Computer Services, Inc.* [Online] [Cited: September 23, 2009.] http://www.ccsinfo.com/product_info.php?products_id=proethkit.
13. **Microchip.** PICDEM.net 2 Development Board. *Microchip.* [Online] [Cited: September 24, 2009.] http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1406&dDocName=en028217.
14. **Altera.** Literature: Cyclone II Devices. *Altera.* [Online] [Cited: September 23, 2009.] http://www.altera.com/literature/lit-cyc2.jsp.
15. **Terasic.** FPGA Systems. *Terasic Technologies.* [Online] [Cited: September 24, 2009.] http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=.

16. **Linear Technology Corporation.** *LTC485 Low Power RS485 Interface Transceiver.* Milpitas, CA : Linear Technology Corp., 1994.
17. Linus Socket Part 2: Fundamentals. *The Tenouk's C, C++, STL, Win32, Winsock, MFC and Linux Socket Tutorials.* [Online] [Cited: September 29, 2009.] http://www.tenouk.com/Module39a.html.
18. **Kristoff, John.** The Trouble with UDP Scanning. [Online] March 11, 2002. [Cited: October 1, 2009.] http://condor.depaul.edu/~jkristof/papers/udpscanning.pdf.
19. **Donahoo, Michael J., Calvert, Kenneth L.** *TCP/IP Sockets in C.* Burlington, MA : Elsevier Inc, 2009. ISBN: 978-0-12-374540-8.
20. **O'Steen, Paul.** Transitioning from UNIX to Windows Socket Programming. *TCP/IP Sockets in C.* [Online] [Cited: September 30, 2009.] http://cs.ecs.baylor.edu/~donahoo/practical/CSockets/WindowsSockets.pdf.
21. **Rafiq, Abdul.** Microchip TCP/IP Stack With BSD Socket API for PIC32MX. *Microchip Corporation Web Site.* [Online] November 2, 2007. [Cited: September 1, 2009.] http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en532885. AN1108.
22. **Terasic Technologies.** *DE2-70 User Manual.* s.l. : Terasic Technologies, 2008.
23. **Axelson, Jan.** Designing RS-485 Circuits. *Circuit Cellar.* 1999, 107.
24. **Digi International.** Using RJ45 Adapters to Connect to DB9 Connector. *Digi International.* [Online] [Cited: July 30, 2009.] http://www.digi.com/support/kbase/kbaseresultdetl.jsp?kb=3.
25. **Vahid, Frank, Givargis, Tony.** *Embedded System Design.* Hoboken, NJ : John Wiley & Sons, Inc, 2002. ISBN: 0-471-38678-2.
26. **Simmons, M.** Ethernet Theory of Operation. *Microchip.* [Online] [Cited: July 15, 2009.] http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en533903. AN1120.
27. **Fox, Stephen.** *System Nodes for a Multi-Drop Test Bench Network.* Windsor, ON : University of Windsor, 2009.
28. *Cyclic Codes for Error Detection.* **Peterson, W.W., Brown, D.T.** Gainesville, Fla. : s.n., Jan., 1961. Proceedings of the IRE. Vol. 49, pp. 228-235.
29. **Sivakumar, Nishant.** Beginning Winsock Programming - Simple TCP Server. *The Code Project.* [Online] February 22, 2002. [Cited: July 10, 2009.] http://www.codeproject.com/KB/IP/winsockintro01.aspx.

Appendix A. Schematics and Bill of Materials

A.1 Schematics

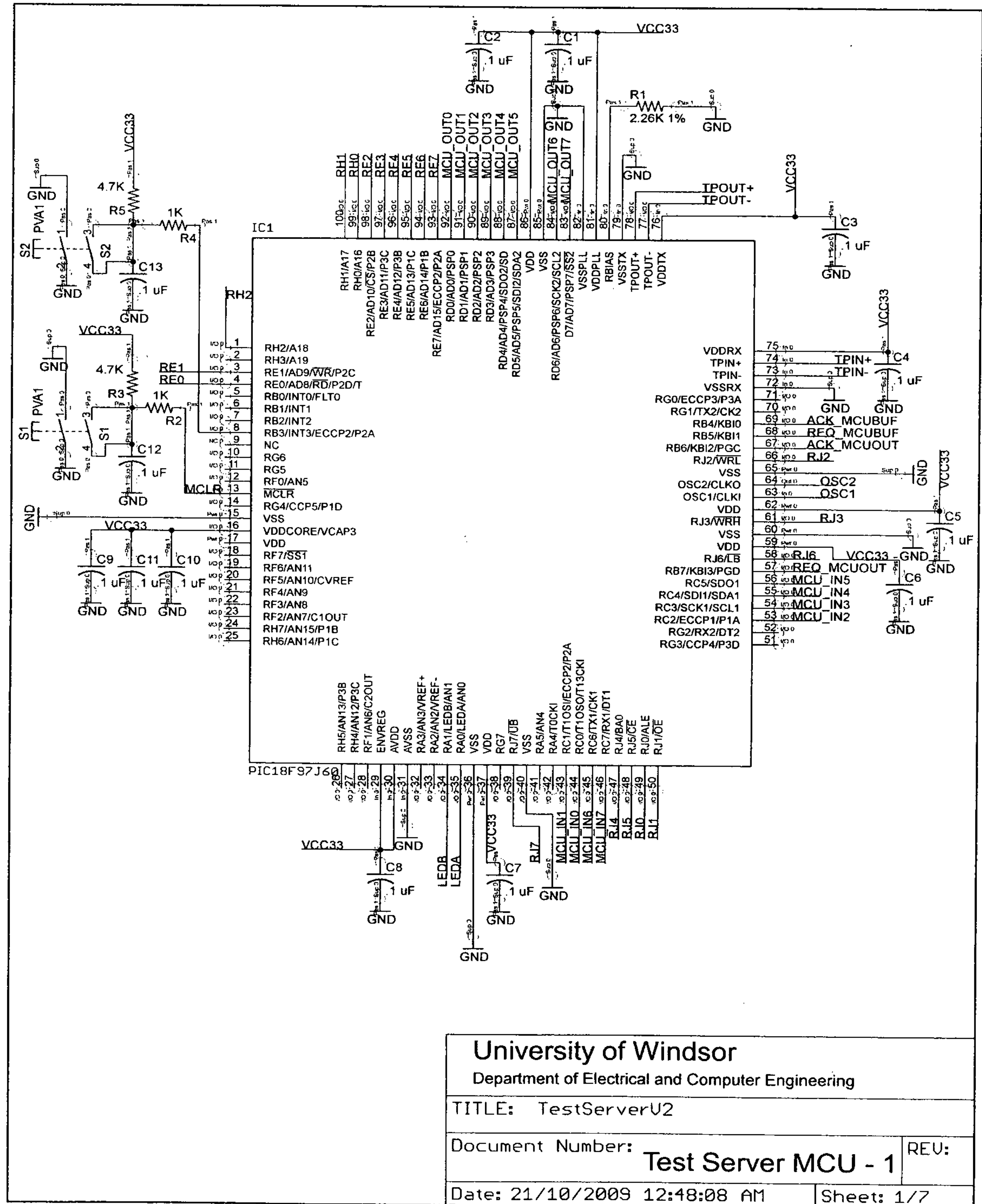

Figure A.1. Test Server MCU Schematic

University of Windsor

Department of Electrical and Computer Engineering

TITLE: TestServerV2

Document Number: Test Server MCU - 2

REV:

Date: 21/10/2009 12:48:08 AM

Sheet: 2/7

Figure A.2. Test Server MCU Schematic – Ethernet, ICSP, LCD, LEDs, Oscillator, and Power

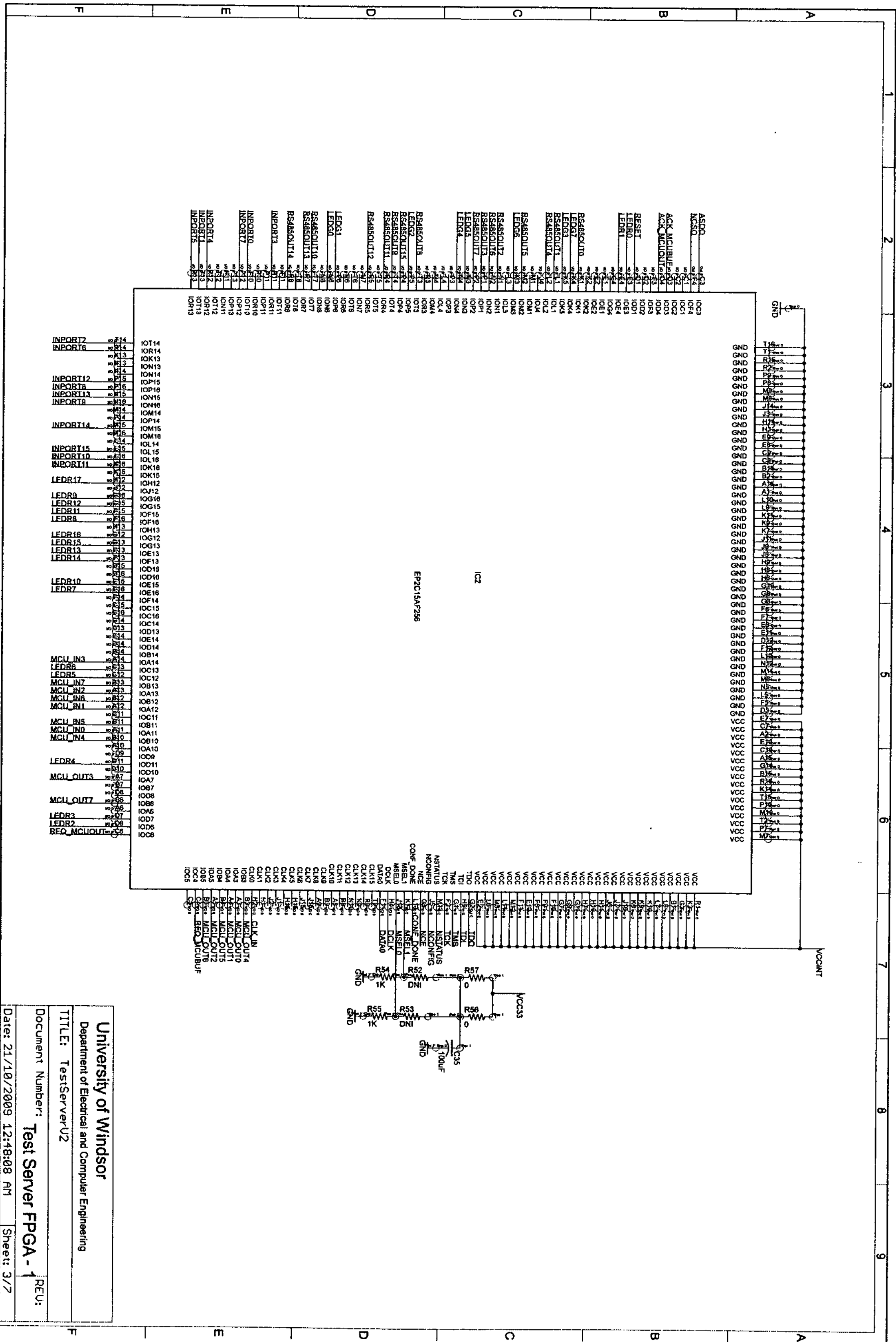

Figure A.3. Test Server FPGA Schematic

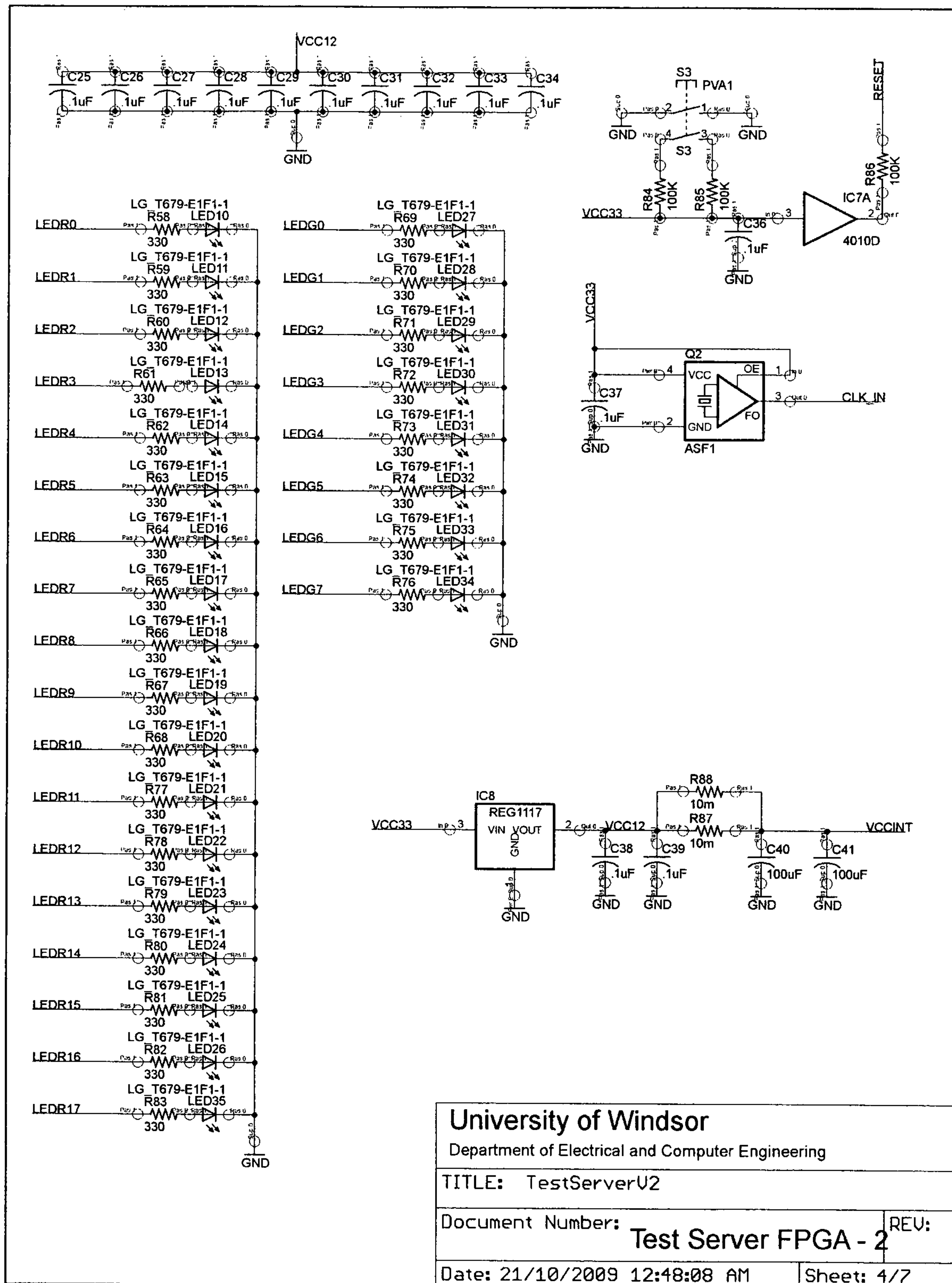

Figure A.4. Test Server FPGA Schematic - LEDs, Oscillator, Power, and Reset

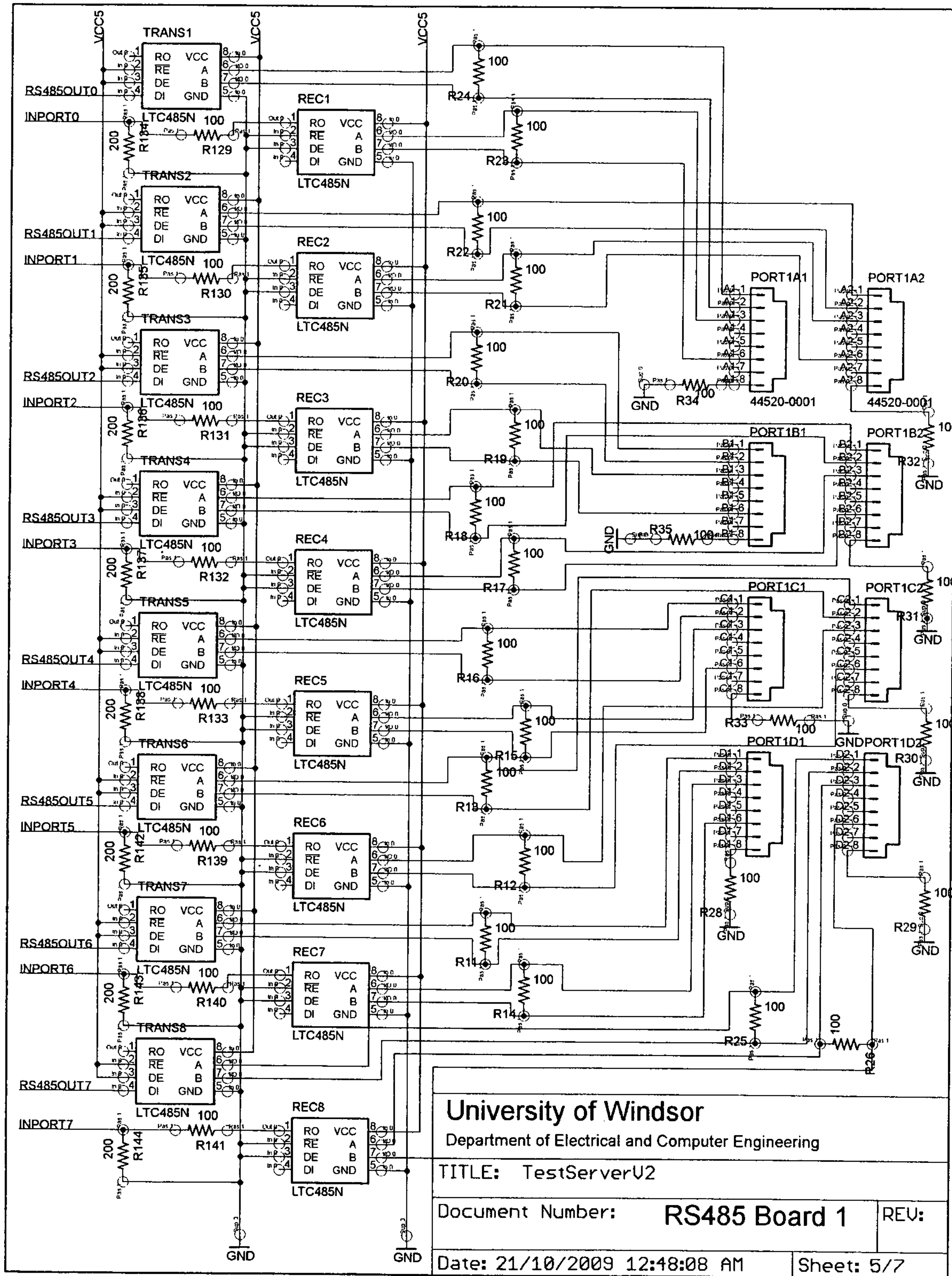

Figure A.5. RS485 Transceiver Network (First 8) Schematic

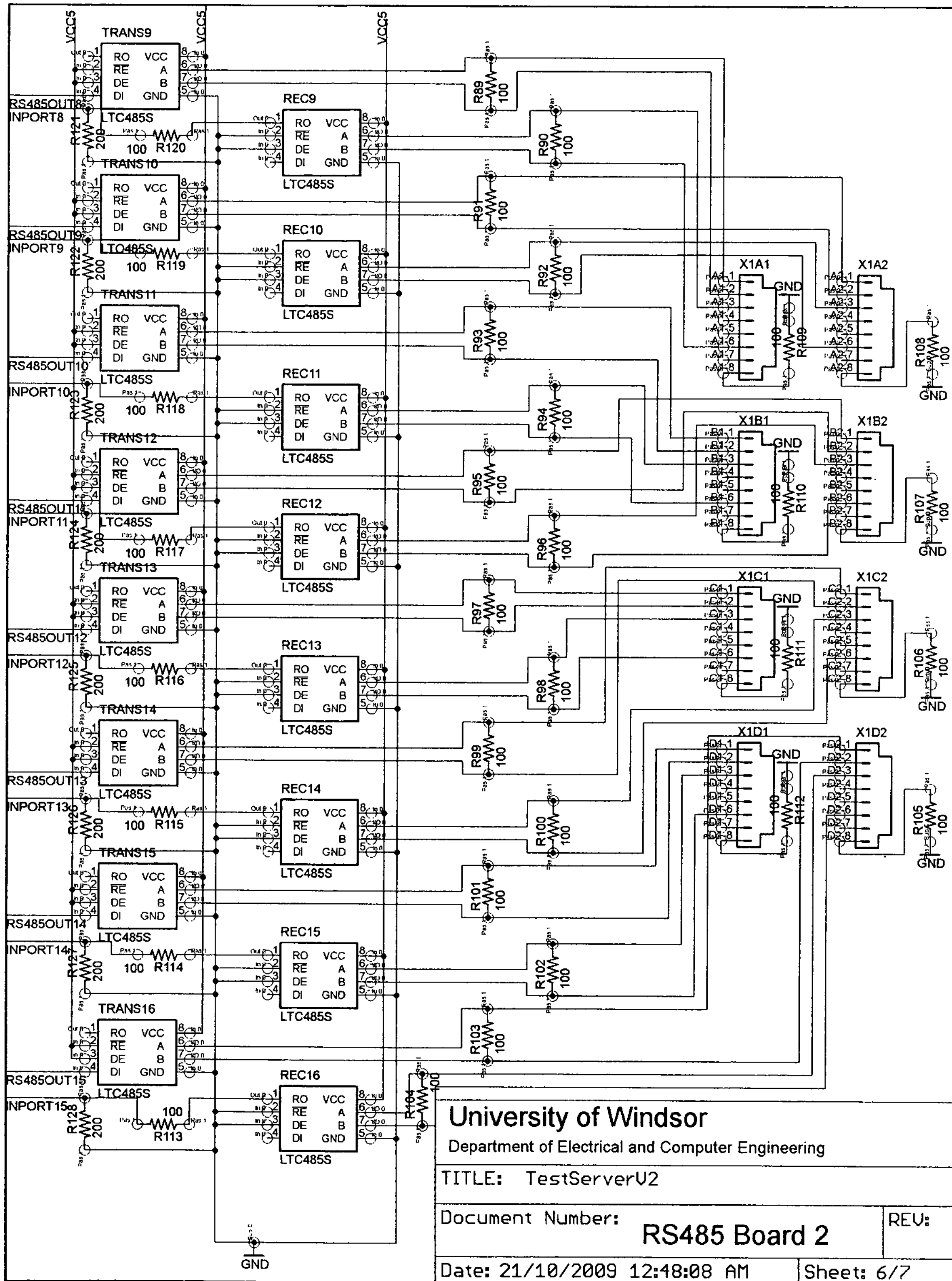

Figure A.6. RS485 Transceiver Network (Second 8) Schematic

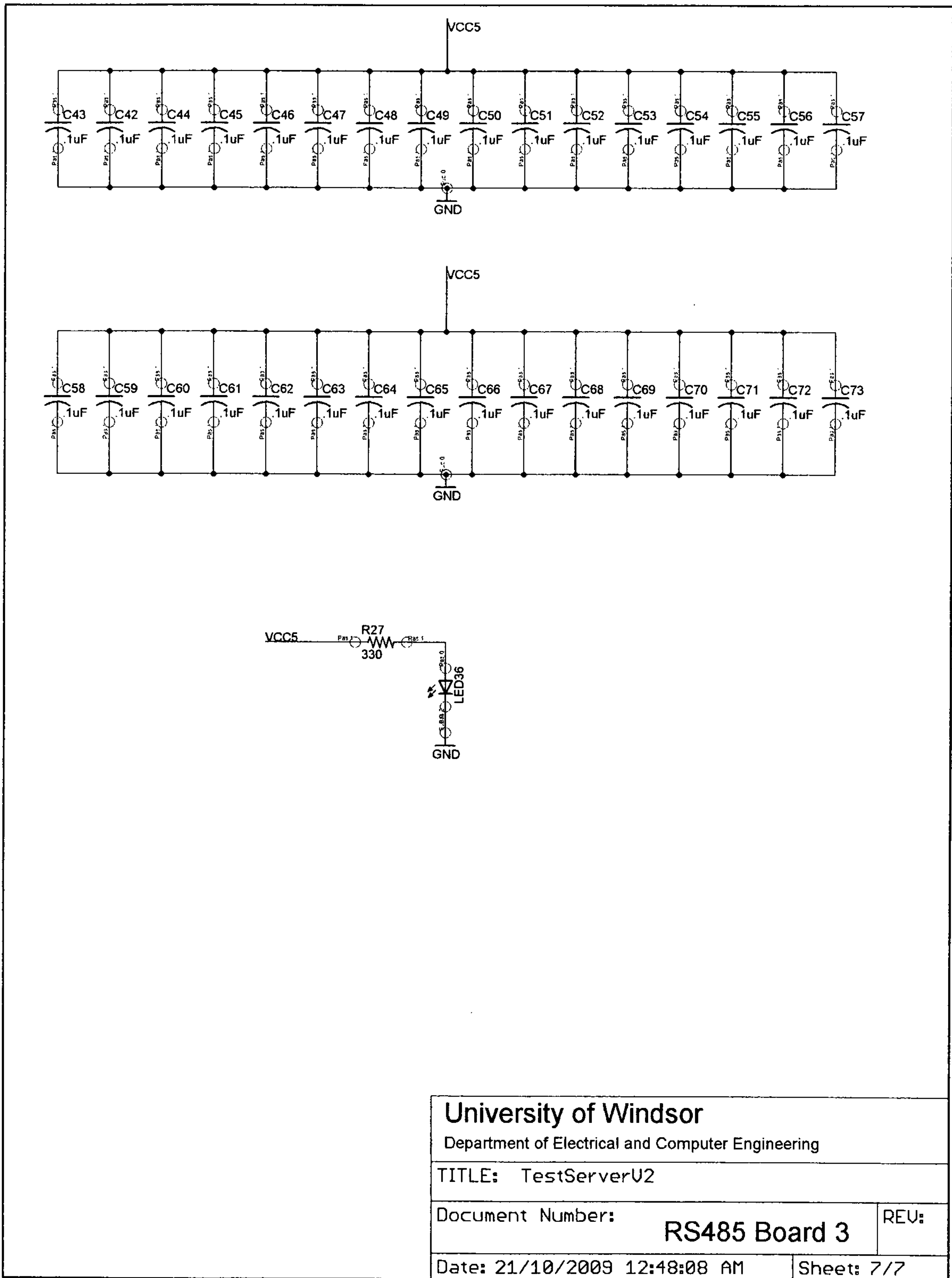

Figure A.7. RS485 Transceiver Network Schematic – Power

A.2 Bill of Materials

A.2.1 Bill of Materials – Prototyping Stage

Table A.1. Bill of Materials - Prototyping Stage

Part List			
Name	Digikey Part Number	Manufacturer	Quantity Used
Ethernet Microcontroller Dev Kit	DM163024-ND	Microchip	1
Power Supply for MCU Kit	AC162039-ND	Microchip	1
FPGA (UART) Dev Kit	P0304A-ND	Terasic	1
RS485 Transceiver	LTC485CN8#PBF-ND	Linear	8
RJ45 Connectors	A31451-ND	Tyco	1
100 ohm Terminating Resistors	100H-ND	Yageo	8
.1 uF Capacitors	399-4454-1-ND	Kemet	8
Prototyping Board	V1256-ND	Vector	1

A.2.2 Bill of Materials – Final Test Server

Part	Value	Device	Package	Library	Sheet
C1	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C2	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C3	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C4	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C5	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C6	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C7	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C8	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C9	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C10	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C11	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C12	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C13	.1 uF	C-US025-024X044	C025-024X044	resistor	1
C14	33 pF	C-US025-024X044	C025-024X044	resistor	2
C15	33 pF	C-US025-024X044	C025-024X044	resistor	2
C16	.1 uF	C-US025-024X044	C025-024X044	resistor	2
C17	.1 uF	C-US025-024X044	C025-024X044	resistor	2
C18	.1 uF	C-US025-024X044	C025-024X044	resistor	2
C19	.1 uF	C-US025-024X044	C025-024X044	resistor	2
C20	220 uF	C-US025-024X044	C025-024X044	resistor	2
C21	.1 uF	C-US025-024X044	C025-024X044	resistor	2
C22	47 uF	C-US025-024X044	C025-024X044	resistor	2
C23	47 uF	C-US025-024X044	C025-024X044	resistor	2
C24	.1 uF	C-US025-024X044	C025-024X044	resistor	2
C25	.1uF	C-US025-024X044	C025-024X044	resistor	4
C26	.1uF	C-US025-024X044	C025-024X044	resistor	4
C27	.1uF	C-US025-024X044	C025-024X044	resistor	4
C28	.1uF	C-US025-024X044	C025-024X044	resistor	4
C29	.1uF	C-US025-024X044	C025-024X044	resistor	4
C30	.1uF	C-US025-024X044	C025-024X044	resistor	4
C31	.1uF	C-US025-024X044	C025-024X044	resistor	4
C32	.1uF	C-US025-024X044	C025-024X044	resistor	4

Part	Value	Device	Package	Library	Sheet
C33	.1uF	C-US025-024X044	C025-024X044	resistor	4
C34	.1uF	C-US025-024X044	C025-024X044	resistor	4
C35	100uF	C-US025-024X044	C025-024X044	resistor	3
C36	.1uF	C-US025-024X044	C025-024X044	resistor	4
C37	.1uF	C-US025-024X044	C025-024X044	resistor	4
C38	.1uF	C-US025-024X044	C025-024X044	resistor	4
C39	.1uF	C-US025-024X044	C025-024X044	resistor	4
C40	100uF	C-US025-024X044	C025-024X044	resistor	4
C41	100uF	C-US025-024X044	C025-024X044	resistor	4
C42	.1uF	C-US025-024X044	C025-024X044	resistor	7
C43	.1uF	C-US025-024X044	C025-024X044	resistor	7
C44	.1uF	C-US025-024X044	C025-024X044	resistor	7
C45	.1uF	C-US025-024X044	C025-024X044	resistor	7
C46	.1uF	C-US025-024X044	C025-024X044	resistor	7
C47	.1uF	C-US025-024X044	C025-024X044	resistor	7
C48	.1uF	C-US025-024X044	C025-024X044	resistor	7
C49	.1uF	C-US025-024X044	C025-024X044	resistor	7
C50	.1uF	C-US025-024X044	C025-024X044	resistor	7
C51	.1uF	C-US025-024X044	C025-024X044	resistor	7
C52	.1uF	C-US025-024X044	C025-024X044	resistor	7
C53	.1uF	C-US025-024X044	C025-024X044	resistor	7
C54	.1uF	C-US025-024X044	C025-024X044	resistor	7
C55	.1uF	C-US025-024X044	C025-024X044	resistor	7
C56	.1uF	C-US025-024X044	C025-024X044	resistor	7
C57	.1uF	C-US025-024X044	C025-024X044	resistor	7
C58	.1uF	C-US025-024X044	C025-024X044	resistor	7
C59	.1uF	C-US025-024X044	C025-024X044	resistor	7
C60	.1uF	C-US025-024X044	C025-024X044	resistor	7
C61	.1uF	C-US025-024X044	C025-024X044	resistor	7
C62	.1uF	C-US025-024X044	C025-024X044	resistor	7
C63	.1uF	C-US025-024X044	C025-024X044	resistor	7
C64	.1uF	C-US025-024X044	C025-024X044	resistor	7
C65	.1uF	C-US025-024X044	C025-024X044	resistor	7
C66	.1uF	C-US025-024X044	C025-024X044	resistor	7
C67	.1uF	C-US025-024X044	C025-024X044	resistor	7
C68	.1uF	C-US025-024X044	C025-024X044	resistor	7
C69	.1uF	C-US025-024X044	C025-024X044	resistor	7
C70	.1uF	C-US025-024X044	C025-024X044	resistor	7
C71	.1uF	C-US025-024X044	C025-024X044	resistor	7
C72	.1uF	C-US025-024X044	C025-024X044	resistor	7
C73	.1uF	C-US025-024X044	C025-024X044	resistor	7
D1	1N4004	1N4004	DO41-10	diode	2
DIS1	TUXGR_16X2_R2	TUXGR_16X2_R2	TUXGR_16X2_R2	display-lcd	2
IC1	PIC18F97J60	PIC18F97J60	TQFP100	microchip	1
IC2	EP2C15AF256	EP2C15AF256	FBGA256	altera-cyclone-II	3
IC3	MAGICJACK	MAGICJACK	MAGJACK	magicjack	2
IC4	RJ11	RJ11	RJ11	magjack	2
IC5	REG1117F	REG1117F	DD-3	burr-brown	2
IC6	REG1117F	REG1117F	DD-3	burr-brown	2
IC7	4010D	4010D	SO16	40xx	4
IC8	REG1117	REG1117	SOT223	burr-brown	4
J1		JACK-PLUG0	SPC4077	con-jack	2
L1		WE-CBF_0805	0805	wuerth-elektronik	2
LED1		LED	SMARTLED-TTW	led	2
LED2		LED	SMARTLED-TTW	led	2
LED3		LED	SMARTLED-TTW	led	2
LED4		LED	SMARTLED-TTW	led	2
LED5		LED	SMARTLED-TTW	led	2
LED6		LED	SMARTLED-TTW	led	2
LED7		LED	SMARTLED-TTW	led	2
LED8		LED	SMARTLED-TTW	led	2

Part	Value	Device	Package	Library	Sheet
LED9		LED	SMARTLED-TTW	led	2
LED10	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED11	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED12	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED13	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED14	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED15	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED16	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED17	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED18	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED19	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED20	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED21	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED22	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED23	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED24	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED25	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED26	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED27	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED28	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED29	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED30	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED31	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED32	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED33	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED34	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED35	LG_T679-E1F1-1	LG_T679-E1F1-1	PLCC2	led	4
LED36		LED	SMARTLED-TTW	led	7
PORT1	44520-0001	44520-0001	44520-0001	con-molex	5
Q1	25 MHz	CRYSTALSM49	SM49	crystal	2
Q2	ASF1	ASF1	ASF	crystal	4
R1	2.26K 1%	R-US_0204/2V	0204V	resistor	1
R2	1K	R-US_0204/2V	0204V	resistor	1
R3	4.7K	R-US_0204/2V	0204V	resistor	1
R4	1K	R-US_0204/2V	0204V	resistor	1
R5	4.7K	R-US_0204/2V	0204V	resistor	1
R6	1M	R-US_0204/2V	0204V	resistor	2
R7	470	R-US_0204/2V	0204V	resistor	2
R8	470	R-US_0204/2V	0204V	resistor	2
R9	470	R-US_0204/2V	0204V	resistor	2
R10	470	R-US_0204/2V	0204V	resistor	2
R11	100	R-US_0204/5	0204/5	resistor	5
R12	100	R-US_0204/5	0204/5	resistor	5
R13	100	R-US_0204/5	0204/5	resistor	5
R14	100	R-US_0204/5	0204/5	resistor	5
R15	100	R-US_0204/5	0204/5	resistor	5
R16	100	R-US_0204/5	0204/5	resistor	5
R17	100	R-US_0204/5	0204/5	resistor	5
R18	100	R-US_0204/5	0204/5	resistor	5
R19	100	R-US_0204/5	0204/5	resistor	5
R20	100	R-US_0204/5	0204/5	resistor	5
R21	100	R-US_0204/5	0204/5	resistor	5
R22	100	R-US_0204/5	0204/5	resistor	5
R23	100	R-US_0204/5	0204/5	resistor	5
R24	100	R-US_0204/5	0204/5	resistor	5
R25	100	R-US_0204/5	0204/5	resistor	5
R26	100	R-US_0204/5	0204/5	resistor	5
R27	330	R-US_0204/2V	0204V	resistor	7
R28	100	R-US_0204/5	0204/5	resistor	5
R29	100	R-US_0204/5	0204/5	resistor	5
R30	100	R-US_0204/5	0204/5	resistor	5

Part	Value	Device	Package	Library	Sheet
R31	100	R-US_0204/5	0204/5	resistor	5
R32	100	R-US_0204/5	0204/5	resistor	5
R33	100	R-US_0204/5	0204/5	resistor	5
R34	100	R-US_0204/5	0204/5	resistor	5
R35	100	R-US_0204/5	0204/5	resistor	5
R36	470	R-US_0204/2V	0204V	resistor	2
R37	470	R-US_0204/2V	0204V	resistor	2
R38	470	R-US_0204/2V	0204V	resistor	2
R39	470	R-US_0204/2V	0204V	resistor	2
R40	49.9	R-US_0204/2V	0204V	resistor	2
R41	49.9	R-US_0204/2V	0204V	resistor	2
R42	49.9	R-US_0204/2V	0204V	resistor	2
R43	49.9	R-US_0204/2V	0204V	resistor	2
R44	180	R-US_0204/2V	0204V	resistor	2
R45	180	R-US_0204/2V	0204V	resistor	2
R46	0	R-US_0204/2V	0204V	resistor	2
R47	0	R-US_0204/2V	0204V	resistor	2
R48	100	R-US_0204/2V	0204V	resistor	2
R49	2 ohm 1%	R-US_0204/2V	0204V	resistor	2
R50	2 ohm 1%	R-US_0204/2V	0204V	resistor	2
R51	2 ohm 1%	R-US_0204/2V	0204V	resistor	2
R52	DNI	R-US_0204/2V	0204V	resistor	3
R53	DNI	R-US_0204/2V	0204V	resistor	3
R54	1K	R-US_0204/2V	0204V	resistor	3
R55	1K	R-US_0204/2V	0204V	resistor	3
R56	0	R-US_0204/2V	0204V	resistor	3
R57	0	R-US_0204/2V	0204V	resistor	3
R58	330	R-US_0204/2V	0204V	resistor	4
R59	330	R-US_0204/2V	0204V	resistor	4
R60	330	R-US_0204/2V	0204V	resistor	4
R61	330	R-US_0204/2V	0204V	resistor	4
R62	330	R-US_0204/2V	0204V	resistor	4
R63	330	R-US_0204/2V	0204V	resistor	4
R64	330	R-US_0204/2V	0204V	resistor	4
R65	330	R-US_0204/2V	0204V	resistor	4
R66	330	R-US_0204/2V	0204V	resistor	4
R67	330	R-US_0204/2V	0204V	resistor	4
R68	330	R-US_0204/2V	0204V	resistor	4
R69	330	R-US_0204/2V	0204V	resistor	4
R70	330	R-US_0204/2V	0204V	resistor	4
R71	330	R-US_0204/2V	0204V	resistor	4
R72	330	R-US_0204/2V	0204V	resistor	4
R73	330	R-US_0204/2V	0204V	resistor	4
R74	330	R-US_0204/2V	0204V	resistor	4
R75	330	R-US_0204/2V	0204V	resistor	4
R76	330	R-US_0204/2V	0204V	resistor	4
R77	330	R-US_0204/2V	0204V	resistor	4
R78	330	R-US_0204/2V	0204V	resistor	4
R79	330	R-US_0204/2V	0204V	resistor	4
R80	330	R-US_0204/2V	0204V	resistor	4
R81	330	R-US_0204/2V	0204V	resistor	4
R82	330	R-US_0204/2V	0204V	resistor	4
R83	330	R-US_0204/2V	0204V	resistor	4
R84	100K	R-US_0204/2V	0204V	resistor	4
R85	100K	R-US_0204/2V	0204V	resistor	4
R86	100K	R-US_0204/2V	0204V	resistor	4
R87	10m	R-US_0204/2V	0204V	resistor	4
R88	10m	R-US_0204/2V	0204V	resistor	4
R89	100	R-US_0204/2V	0204V	resistor	6
R90	100	R-US_0204/2V	0204V	resistor	6
R91	100	R-US_0204/2V	0204V	resistor	6

Part	Value	Device	Package	Library	Sheet
R92	100	R-US_0204/2V	0204V	resistor	6
R93	100	R-US_0204/2V	0204V	resistor	6
R94	100	R-US_0204/2V	0204V	resistor	6
R95	100	R-US_0204/2V	0204V	resistor	6
R96	100	R-US_0204/2V	0204V	resistor	6
R97	100	R-US_0204/2V	0204V	resistor	6
R98	100	R-US_0204/2V	0204V	resistor	6
R99	100	R-US_0204/2V	0204V	resistor	6
R100	100	R-US_0204/2V	0204V	resistor	6
R101	100	R-US_0204/2V	0204V	resistor	6
R102	100	R-US_0204/2V	0204V	resistor	6
R103	100	R-US_0204/2V	0204V	resistor	6
R104	100	R-US_0204/2V	0204V	resistor	6
R105	100	R-US_0204/2V	0204V	resistor	6
R106	100	R-US_0204/2V	0204V	resistor	6
R107	100	R-US_0204/2V	0204V	resistor	6
R108	100	R-US_0204/2V	0204V	resistor	6
R109	100	R-US_0204/2V	0204V	resistor	6
R110	100	R-US_0204/2V	0204V	resistor	6
R111	100	R-US_0204/2V	0204V	resistor	6
R112	100	R-US_0204/2V	0204V	resistor	6
R113	100	R-US_0204/2V	0204V	resistor	6
R114	100	R-US_0204/2V	0204V	resistor	6
R115	100	R-US_0204/2V	0204V	resistor	6
R116	100	R-US_0204/2V	0204V	resistor	6
R117	100	R-US_0204/2V	0204V	resistor	6
R118	100	R-US_0204/2V	0204V	resistor	6
R119	100	R-US_0204/2V	0204V	resistor	6
R120	100	R-US_0204/2V	0204V	resistor	6
R121	200	R-US_0204/2V	0204V	resistor	6
R122	200	R-US_0204/2V	0204V	resistor	6
R123	200	R-US_0204/2V	0204V	resistor	6
R124	200	R-US_0204/2V	0204V	resistor	6
R125	200	R-US_0204/2V	0204V	resistor	6
R126	200	R-US_0204/2V	0204V	resistor	6
R127	200	R-US_0204/2V	0204V	resistor	6
R128	200	R-US_0204/2V	0204V	resistor	6
R129	100	R-US_0204/5	0204/5	resistor	5
R130	100	R-US_0204/5	0204/5	resistor	5
R131	100	R-US_0204/5	0204/5	resistor	5
R132	100	R-US_0204/5	0204/5	resistor	5
R133	100	R-US_0204/5	0204/5	resistor	5
R134	200	R-US_0204/5	0204/5	resistor	5
R135	200	R-US_0204/5	0204/5	resistor	5
R136	200	R-US_0204/5	0204/5	resistor	5
R137	200	R-US_0204/5	0204/5	resistor	5
R138	200	R-US_0204/5	0204/5	resistor	5
R139	100	R-US_0204/5	0204/5	resistor	5
R140	100	R-US_0204/5	0204/5	resistor	5
R141	100	R-US_0204/5	0204/5	resistor	5
R142	200	R-US_0204/5	0204/5	resistor	5
R143	200	R-US_0204/5	0204/5	resistor	5
R144	200	R-US_0204/5	0204/5	resistor	5
REC1	LTC485N	LTC485N	SOIC8	linear-technology	5
REC2	LTC485N	LTC485N	SOIC8	linear-technology	5
REC3	LTC485N	LTC485N	SOIC8	linear-technology	5
REC4	LTC485N	LTC485N	SOIC8	linear-technology	5
REC5	LTC485N	LTC485N	SOIC8	linear-technology	5
REC6	LTC485N	LTC485N	SOIC8	linear-technology	5
REC7	LTC485N	LTC485N	SOIC8	linear-technology	5

Part	Value	Device	Package	Library	Sheet
REC8	LTC485N	LTC485N	SOIC8	linear-technology	5
REC9	LTC485S	LTC485S	SOIC8	linear-technology	6
REC10	LTC485S	LTC485S	SOIC8	linear-technology	6
REC11	LTC485S	LTC485S	SOIC8	linear-technology	6
REC12	LTC485S	LTC485S	SOIC8	linear-technology	6
REC13	LTC485S	LTC485S	SOIC8	linear-technology	6
REC14	LTC485S	LTC485S	SOIC8	linear-technology	6
REC15	LTC485S	LTC485S	SOIC8	linear-technology	6
REC16	LTC485S	LTC485S	SOIC8	linear-technology	6
S1	PVA1	PVA1	PVA1F	switch-misc	1
S2	PVA1	PVA1	PVA1F	switch-misc	1
S3	PVA1	PVA1	PVA1F	switch-misc	4
S4	PVA2F	PVA2F	PVA2F	switch-misc	2
TRANS1	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS2	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS3	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS4	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS5	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS6	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS7	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS8	LTC485N	LTC485N	SOIC8	linear-technology	5
TRANS9	LTC485S	LTC485S	SOIC8	linear-technology	6
TRANS10	LTC485S	LTC485S	SOIC8	linear-technology	6
TRANS11	LTC485S	LTC485S	SOIC8	linear-technology	6
TRANS12	LTC485S	LTC485S	SOIC8	linear-technology	6
TRANS13	LTC485S	LTC485S	SOIC8	linear-technology	6
TRANS14	LTC485S	LTC485S	SOIC8	linear-technology	6
TRANS15	LTC485S	LTC485S	SOIC8	linear-technology	6
TRANS16	LTC485S	LTC485S	SOIC8	linear-technology	6
X1	44520-0001	44520-0001	44520-0001	con-molex	6

Appendix B. Test Server VHDL Code

B.1 server.vhd

```
----------------------------------
--File Name: server.vhd
--Description: Test Server Module
--Author: Christopher Rennick
--Date: April 30, 2009
--Simulator: Altera Quartus II
----------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity server is
	port(CLK, RESET: in std_logic;
	 req_mcu_buf, req_mcu_output: in std_logic;		--req signals from MCU
	MCU_IN: in std_logic_vector(7 downto 0);		--input from MCU
	in_port: in std_logic_vector(15 downto 0);		--input from rs485 network
	ack_mcu_buf, ack_mcu_output: out std_logic;		--ack signals to MCU
	rs485out: out std_logic_vector(15 downto 0);		--output to rs485 network
	MCU_OUT: out std_logic_vector(7 downto 0);		--output to MCU
	LEDG: out std_logic_vector(7 downto 0);		--LED signals for dev. kit
	LEDR: out std_logic_vector(17 downto 0));
end server;

architecture behavioural of server is
signal div_CLK: std_logic;
begin
	--instantiation of rs485 to MCU
	rs485_to_mcu_entity: entity work.rs485_to_mcu
	port map(CLK => div_CLK, RESET => not(RESET),
	req => req_mcu_output,
	in_port => in_port,
	ack => ack_mcu_output,
	data_out => MCU_OUT,
	LEDR => LEDR(8 downto 0));

	--instantiation of MCU to rs485
	mcu_to_485_entity: entity work.mcu_to_485
	port map(CLK => div_CLK, RES => not(RESET),
	data_in => MCU_IN,
	req => req_mcu_buf,
	ack => ack_mcu_buf, rs485out => rs485out,
	LEDG => LEDG, LEDR => LEDR(17 downto 9));

	--The following is a clock divider to get the Dev. Kit clock speed of 50MHz
	-- down to the desired speed of 25MHz
	process(CLK, RESET)
	begin
		if (RESET='0') then
			div_CLK<='0';
		elsif (CLK'event and CLK='1') then
			div_CLK<=not(div_CLK);
		end if;
	end process;
end behavioural;
```

B.2 mcu_to_485.vhd

```
-------------------------------------------------------------------
--File Name: mcu_to_485.vhd
--Description: Test Server Module that takes parallel MCU data in
--and outputs the RS485 serial data
--Author: Christopher Rennick
--Date: April 29, 2009
--Simulator: Altera Quartus II
-------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mcu_to_485 is
	port(CLK, RES: in std_logic;
	data_in: in std_logic_vector(7 downto 0);		--input from MCU
	req: in std_logic;
	ack: out std_logic;
	rs485out: out std_logic_vector(15 downto 0);		--output to rs485 network
	LEDG: out std_logic_vector(7 downto 0);
	LEDR: out std_logic_vector(8 downto 0));
end mcu_to_485;

architecture behavioural of mcu_to_485 is
signal mem_wr, mem_en, data_avail1: std_logic;
signal data_bus: std_logic_vector(7 downto 0);
begin
	--instantiation of mcu_buf_mem module
	mcu_buf_mem1: entity work.mcu_buf_mem
	port map(CLK => CLK,
	RESET => RES,
	req => req,
	busy => mem_wr,
	MCU_in => data_in,
	mem1_en => mem_en,
	data_avail => data_avail1,
	ack => ack,
	data_out => data_bus,
	LEDG => LEDG,
	LEDR => LEDR(8 downto 5));

	--instantiation of rs485_output module
	rs485_output1: entity work.rs485_output
	port map(CLK => CLK,
	RESET => RES,
	data_avail => data_avail1,
	data_in => data_bus,
	WR => mem_wr,
	rs485out => rs485out,
	LEDR => LEDR(4 downto 0));
end behavioural;
```

B.2.1 mcu_buf_mem.vhd

```
-------------------------------------------------------------------
--File Name: mcu_buf_mem.vhd
--Description: Test Server Module that takes parallel MCU data in
--and stores it in a FIFO RAM buffer
--Author: Christopher Rennick
--Date: March 25, 2009
--Simulator: Altera Quartus II
-------------------------------------------------------------------
```

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mcu_buf_mem is
		port(CLK, RESET, req: in std_logic;
		busy: in std_logic;
		MCU_IN: in std_logic_vector(7 downto 0);		--input from MCU
		mem1_en: inout std_logic;
		data_avail: out std_logic;
		ack: out std_logic;
		data_out: out std_logic_vector(7 downto 0);		--output to rs485_output module
		LEDG: out std_logic_vector(7 downto 0);
		LEDR: out std_logic_vector(3 downto 0));
end mcu_buf_mem;

architecture behavioural of mcu_buf_mem is
signal mem1_wr, mem1_busy, FIFO_full_sig: std_logic;
signal mem1_datain, mem1_dataout: std_logic_vector(7 downto 0);
begin
		--instantiation of memory for incoming MCU and outgoing MCU data
		mem1: entity work.fifo
		port map(CLK => CLK,
		RES => RESET,
		EN => mem1_en,
		WR => busy,
		data_in => mem1_datain,
		data_avail => data_avail,
		FIFO_full => FIFO_full_sig,
		data_out => data_out,
		LEDR => LEDR(1 downto 0));

		--instantiation of input buffer for MCU data
		-- ***busy signal is tied to WR for mem1: if WR is 0 it is busy,
		-- *** if WR is 1, it is free***
		inmcu_buf: entity work.mcu_buf
		port map(CLK => CLK,
		RESET => RESET,
		req => req,
		busy => busy,
		FIFO_full => FIFO_full_sig,
		MCU_IN => MCU_IN,
		ack => ack,
		mem_en => mem1_en,
		data_out => mem1_datain,
		LEDR => LEDR(3 downto 2),
		LEDG => LEDG);
end behavioural;
```

B.2.2 rs485_output.vhd

```
----------------------------------------------------------------
--File Name: rs485_output.vhd
--Description: Test Server Module that outputs RS485 Serial Data
--Author: Christopher Rennick
--Date: April 14, 2009
--Simulator: Altera Quartus II
----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity rs485_output is
```

```
	port(CLK, RESET, data_avail: in std_logic;
	data_in: in std_logic_vector(7 downto 0);				--input from FIFO module
	WR: out std_logic;
	rs485out: out std_logic_vector(15 downto 0);			--output to RS485 network
	LEDR: out std_logic_vector(4 downto 0));
end rs485_output;

architecture behavioural of rs485_output is
type mem_type is array (0 to 31) of std_logic_vector(7 downto 0);
signal outbuff: mem_type;								--output buffer
signal addr: std_logic_vector (3 downto 0);		--used to determine the bus to transmit to

--flags used to determine whether the module is reading, writing or sending the first byte of the packet
signal reading, sending, firstsend: std_logic;

signal clock_div, in_count: integer;				--signal for clock divider and input counter
signal send_count1, send_count2: integer;		--counters for sending packet bytes
begin
	process (CLK, RESET)
	begin
		if (RESET='1') then
			LEDR<="00000";
			in_count<=0;
			clock_div<=0;
			WR<='1';						--memory WR signal
			send_count1<=0;
			send_count2<=0;
			reading<='0';
			sending<='0';
			firstsend<='0';
			rs485out<="1111111111111111";		--initializing all buses to '1'
			addr<="0000";
		elsif (CLK'event and CLK='1') then
			--checking if module is sending
			if (sending='0') then
				--check to see if memory contains any data

				if (data_avail='1') then
					reading<='1';
				end if;

				if (reading='1') then
					--inputting data from FIFO module
					if (in_count>=0 and in_count<30) then
						outbuff(in_count-3)<=data_in;
						--when reading from memory, WR is '0'
						WR <= '0';
						in_count<=in_count+1;
						reading<='1';
					elsif (in_count>=30 and in_count<=32) then
						WR<='1';
						outbuff(in_count-3)<=data_in;
						LEDR<="10000";
						in_count<=in_count+1;
						reading<='1';
					elsif (in_count>32) then
						WR <= '0';
						firstsend<='1';
						in_count<=0;
						sending<='1';
						reading<='0';
						addr<=outbuff(0)(7 downto 4);
					end if;
				end if;

			elsif (sending ='1') then			--now sending data
```

```
					WR <= '1';
					clock_div<=clock_div+1;

					--firstsend is used so there is no wait to send the first byte of the packet
					if (firstsend = '1' or clock_div=249) then
						clock_div<=0;
						firstsend<='0';
						if (send_count1=0) then
							rs485out(to_integer(unsigned(addr))) <= '0';
						elsif (send_count1 > 0 and send_count1 <=8) then
							rs485out(to_integer(unsigned(addr))) <= outbuff(send_count2)(8-
send_count1);
						elsif (send_count1 = 9) then
							rs485out(to_integer(unsigned(addr))) <= '1';
						end if;


						if (send_count1=9) then		--increase the counters appropriately
							send_count1<=0;		--send_count1 counts bits/byte
							LEDR(2)<='0';
							if (send_count2 = 31) then
								--send_count2 counts bytes/packet
								send_count2<=0;
								sending<='0';
								LEDR(3)<='1';
							else
								send_count2<=send_count2+1;
								sending<='1';
								LEDR(3)<='0';
							end if;
						elsif (send_count1>3 and send_count1<8) then
							LEDR(2)<='1';
							send_count1<=send_count1+1;
						else
							send_count1<=send_count1+1;
							LEDR(2)<='0';
						end if;
					end if;
				end if;
			end if;
		end process;
end behavioural;
```

B.2.3 *fifo.vhd*

```
-----------------------------------------------------------------
--File Name: fifo.vhd
--Description: Test Server Module that stores parallel data that
--came in from the MCU
--Author: Christopher Rennick
--Date: April 14, 2009
--Simulator: Altera Quartus II
-----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- memory size is based on an assumption of 32 byte packets

entity FIFO is
	port(CLK, RES, EN, WR: in std_logic;
	data_in: in std_logic_vector(7 downto 0);		--input from MCU_buf module
	data_avail, FIFO_full: out std_logic;
	data_out: out std_logic_vector(7 downto 0);		--output to RS485_output module
```

```
		LEDR: out std_logic_vector(1 downto 0));
end FIFO;

architecture behavioural of FIFO is
signal read_addr, write_addr: std_logic_vector(39 downto 0);
signal address_a_sig, address_b_sig: std_logic_vector(12 downto 0);
signal wren_a_sig: std_logic;
signal q_a_sig: std_logic_vector(7 downto 0);
begin
	--instantiation of dual port RAM megafunction created by MegaWizard in Quartus II
	fifo_ram_inst : entity work.fifo_ram
	PORT MAP (
		address_a  => address_a_sig,
		address_b  => address_b_sig,
		clock      => CLK,
		data_a     => data_in,
		data_b     => "00000000",
		wren_a     => wren_a_sig,
		wren_b     => '0',
		q_a        => q_a_sig,
		q_b        => data_out
	);

	process (CLK, RES)
	begin
		if (RES='1') then
			read_addr<=X"0000000000";
			write_addr<=X"0000000000";
			address_a_sig<="0000000000000";
			data_avail<='0';
			wren_a_sig<='0';
			LEDR<="00";
			FIFO_full<='0';
		elsif (CLK'event and CLK='1') then
			--setting the data_avail flag to show if there is data available in the FIFO
			if ((to_integer(unsigned(write_addr))-to_integer(unsigned(read_addr)))>=32) then
				data_avail<='1';
			else
				data_avail<='0';
			end if;

			--this is used to set the FIFO_full flag if the write pointer is more than ¼ of the max
			--memory space ahead of the read pointer
			if((to_integer(unsigned(write_addr))-to_integer(unsigned(read_addr)))>=2048) then
				FIFO_full<='1';
			else
				FIFO_full<='0';
			end if;

			--writing to the FIFO
			if (EN='1') then
				address_a_sig<=write_addr(12 downto 0);
				wren_a_sig<='1';
				LEDR(0)<='1';
				write_addr<=std_logic_vector(to_unsigned((to_integer(unsigned(write_addr))+1),40));

			else
				wren_a_sig<='0';
				address_a_sig<="0000000000000";
				LEDR(0)<='0';
				write_addr<=write_addr;
			end if;

			--reading from the FIFO
			if (WR='0') then
				address_b_sig<=read_addr(12 downto 0);
```

```
                    LEDR(1)<='1';
                    read_addr<=std_logic_vector(to_unsigned((to_integer(unsigned(read_addr))+1),40));
                else
                    LEDR(1)<='0';
                    address_b_sig<="0000000000000";
                    read_addr<=read_addr;
                end if;

            end if;
        end process;
end behavioural;
```

B.2.4 mcu_buf.vhd

```
----------------------------------------------------------------
--File Name: mcu_buf.vhd
--Description: Test Server Module that buffers parallel data that
--came in from the MCU before sending it to the FIFO RAM
--Author: Christopher Rennick
--Date: April 14, 2009
--Simulator: Altera Quartus II
----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity mcu_buf is
        port(CLK, RESET, req: in std_logic;
        busy, FIFO_full: in std_logic;
        MCU_IN: in std_logic_vector(7 downto 0);                --input from MCU
        ack, mem_en: out std_logic;
        data_out: out std_logic_vector(7 downto 0);             --output to FIFO module
        LEDR: out std_logic_vector(1 downto 0);
        LEDG: out std_logic_vector(7 downto 0));
end mcu_buf;

architecture behavioural of mcu_buf is
type buf_type is array(0 to 31) of std_logic_vector(7 downto 0);
type mem_type is array(0 to 3) of buf_type;

signal buff: mem_type;
signal int_ack, write_sig: std_logic;
signal switch, writing: std_logic_vector(1 downto 0);
signal count, send_count: integer;
begin
        process(CLK, RESET, req)
        begin
            if (RESET='1') then
                ack <= '0';             --ack signal to MCU
                int_ack <= '0';                 --copy of ack signal
                count<=0;               --counts incoming bytes from MCU
                switch <= "00";                 --stores which buffer is being written to
                writing <= "00";                --stores which buffer is being read from
                send_count<=0;                  --counts the bytes being sent to the FIFO
                mem_en <= '0';                  --signal to FIFO
                LEDR<="00";                     --LED control for dev. kit
                LEDG <="00000001";
            elsif (CLK'event and CLK='1') then
                LEDG<="00000000";
                --ack signal must be brought low after every byte is received
                if (int_ack='1') then
                    LEDG(7)<='1';
                    ack<='0';
```

```
		int_ack<='0';
	elsif (int_ack='0') then
		LEDG(7)<='0';
		--if req=1 then MCU has data to send
		if (req='1') then
			--checking whether a full packet has been received
			if (count>=0 and count <32) then
				--if FIFO is full, do not accept data from MCU
				if (FIFO_full = '0') then
					buff(to_integer(unsigned(switch)))(count) <= MCU_IN;
					count<=count+1;
					ack<='1';
					int_ack<='1';
					LEDG(6)<='0';
				else
					LEDG(6)<='1';
					ack<='0';
					int_ack<='0';
				end if;

			--when a full packet has been received, switch input buffer pointer
			--and set write buffer pointer to last input buffer
			elsif (count=32) then
				LEDR<="10";
			switch<=std_logic_vector(to_unsigned((to_integer(unsigned(switch))+1),2));
				count<=0;
				write_sig<='1';
			end if;

		--if there is no data to be sent, keep ack low
		elsif (req='0') then
			ack <= '0';
			int_ack<='0';
		end if;
	end if;

	--after a packet has been received, switch to next buffer and start
	--sending first buffer to FIFO
	if (write_sig='1') then
		--if memory is not busy (busy=1), start sending data
		if (busy='1') then
			--sending data
			if (send_count=0) then
				mem_en<='1';
				send_count<=send_count+1;
				LEDR<="00";
				write_sig<='1';
			elsif (send_count<32 and send_count>0) then
				mem_en<='1';
				data_out<=buff(to_integer(unsigned(writing)))(send_count-1);
				send_count<=send_count+1;
				LEDR<="00";
				write_sig<='1';
			--when done sending, turn off mem1 enable
			elsif (send_count=32) then
				data_out<=buff(to_integer(unsigned(writing)))(send_count-1);
				send_count<=send_count+1;
				mem_en<='0';
				LEDR<="01";
				write_sig<='1';
			else
				send_count<=0;
				mem_en<='0';
				LEDR<="00";
				write_sig<='0';
			writing<=std_logic_vector(to_unsigned((to_integer(unsigned(writing))+1),2));
```

```
                                        end if;
                                else
                                        mem_en<='0';
                                        LEDR<="00";
                                        write_sig<=write_sig;
                                        send_count<=send_count;
                                        writing<=writing;
                                end if;
                        end if;
                end if;
        end process;
end behavioural;
```

B.3 rs485_to_mcu.vhd

```
----------------------------------------------------------------------
--File Name: rs485_to_mcu.vhd
--Description: Test Server Module that accepts data from the RS485
--network, buffers it, and sends it to the MCU
--Author: Christopher Rennick
--Date: April 28, 2009
--Simulator: Altera Quartus II
----------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rs485_to_mcu is
        port(CLK, RESET, req: in std_logic;
        in_port: in std_logic_vector(15 downto 0);              --input from RS485 network
        ack: out std_logic;
        data_out: out std_logic_vector(7 downto 0);             --output to MCU
        LEDR: out std_logic_vector(8 downto 0));
end rs485_to_mcu;

architecture behavioural of rs485_to_mcu is
signal busy_sig, data_avail_sig: std_logic;
signal data_out_sig: std_logic_vector(7 downto 0);
begin
        --instantiation of rs485_buf_mem module
        in485_buf_mem: entity work.rs485_buf_mem
        port map(CLK => CLK, RESET => RESET,
        busy => busy_sig,
        in0 => in_port(0), in1=> in_port(1), in2=> in_port(2), in3=> in_port(3),
        in4=> in_port(4), in5=> in_port(5), in6=> in_port(6), in7=> in_port(7),
        in8=> in_port(8), in9=> in_port(9), in10=> in_port(10), in11=> in_port(11),
        in12=> in_port(12), in13=> in_port(13), in14=> in_port(14), in15=> in_port(15),
        data_avail => data_avail_sig,
        LEDR => LEDR(4 downto 0),
        data_out => data_out_sig);

        --instantiation of mcu_output module
        mcu_output_inst: entity work.mcu_output
        port map(CLK => CLK, RESET => RESET,
        data_avail => data_avail_sig,
        req => req, data_in => data_out_sig,
        WR => busy_sig, ack => ack,
        LEDR => LEDR(8 downto 5),
        data_out => data_out);
end behavioural;
```

B.3.1 rs485buf_mem.vhd

```
--------------------------------------------------------------------------
--File Name: rs485_buf_mem.vhd
--Description: This module consists of the 16 rs485_buf
-- Modules Combined With the FIFO RAM Block
--Author: Christopher Rennick
--Date: April 13, 2009
--Simulator: Altera Quartus II
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity rs485_buf_mem is
        port(CLK, RESET, busy: in std_logic;
        in0, in1, in2, in3, in4, in5, in6, in7: in std_logic;            --inputs from RS485 network
        in8, in9, in10, in11, in12, in13, in14, in15 : in std_logic;
        data_avail: out std_logic;
        LEDR: out std_logic_vector(4 downto 0);
        data_out: out std_logic_vector(7 downto 0));                     --output to mcu_output module
end rs485_buf_mem;

architecture behavioural of rs485_buf_mem is
signal mem2_wr, mem2_en: std_logic;
signal mem2_datain, mem2_dataout: std_logic_vector(7 downto 0);
signal data_avail_sig: std_logic_vector(15 downto 0);
signal EN_sig: std_logic_vector(15 downto 0);
begin
        --instantiation of memory for incoming and outgoing test node data
        mem2: entity work.rs485_fifo
        port map(CLK => CLK,  EN => mem2_en, WR => busy, RES => RESET, data_in => mem2_datain,
        data_avail => data_avail, LEDR => LEDR(4 downto 2), data_out => data_out);

        --instantiation of rotating priority arbiter
        arb1: entity work.arbiter
        port map(CLK => CLK, RES => RESET, busy => busy, data_avail => data_avail_sig,
                LEDR => LEDR(1 downto 0), EN => EN_sig);

        --instantiation of 16 input buffers for RS485 serial inputs
        in485_buf0: entity work.rs485_buf
        port map(CLK => CLK, RESET => RESET, busy => EN_sig(0), data_in => in0,
        data_avail => data_avail_sig(0), mem_en => mem2_en, data_out => mem2_datain);

        in485_buf1: entity work.rs485_buf
        port map(CLK => CLK, RESET => RESET, busy => EN_sig(1), data_in => in1,
        data_avail => data_avail_sig(1), mem_en => mem2_en, data_out => mem2_datain);

        in485_buf2: entity work.rs485_buf
        port map(CLK => CLK, RESET => RESET, busy => EN_sig(2), data_in => in2,
        data_avail => data_avail_sig(2), mem_en => mem2_en, data_out => mem2_datain);

        in485_buf3: entity work.rs485_buf
        port map(CLK => CLK, RESET => RESET, busy => EN_sig(3), data_in => in3,
        data_avail => data_avail_sig(3), mem_en => mem2_en, data_out => mem2_datain);

        in485_buf4: entity work.rs485_buf
        port map(CLK => CLK, RESET => RESET, busy => EN_sig(4), data_in => in4,
        data_avail => data_avail_sig(4), mem_en => mem2_en, data_out => mem2_datain);

        in485_buf5: entity work.rs485_buf
        port map(CLK => CLK, RESET => RESET, busy => EN_sig(5), data_in => in5,
        data_avail => data_avail_sig(5), mem_en => mem2_en, data_out => mem2_datain);
```

```
            in485_buf6: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(6), data_in => in6,
            data_avail => data_avail_sig(6), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf7: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(7), data_in => in7,
            data_avail => data_avail_sig(7), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf8: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(8), data_in => in8,
            data_avail => data_avail_sig(8), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf9: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(9), data_in => in9,
            data_avail => data_avail_sig(9), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf10: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(10), data_in => in10,
            data_avail => data_avail_sig(10), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf11: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(11), data_in => in11,
            data_avail => data_avail_sig(11), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf12: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(12), data_in => in12,
            data_avail => data_avail_sig(12), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf13: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(13), data_in => in13,
            data_avail => data_avail_sig(13), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf14: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(14), data_in => in14,
            data_avail => data_avail_sig(14), mem_en => mem2_en, data_out => mem2_datain);

            in485_buf15: entity work.rs485_buf
            port map(CLK => CLK, RESET => RESET, busy => EN_sig(15), data_in => in15,
            data_avail => data_avail_sig(15), mem_en => mem2_en, data_out => mem2_datain);
end behavioural;
```

B.3.2 mcu_output.vhd

```
----------------------------------------------------------------
--File Name: mcu_output.vhd
--Description: Output Module for Sending Parallel Data to MCU
--Author: Christopher Rennick
--Date: April 10, 2009
--Simulator: Altera Quartus II
----------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity mcu_output is
            port(CLK, RESET: in std_logic;
            data_avail, req: in std_logic;
            data_in: in std_logic_vector(7 downto 0);          --data from FIFO
            WR: out std_logic;
            ack: out std_logic;
            LEDR: out std_logic_vector(3 downto 0);
            data_out: out std_logic_vector(7 downto 0));       --data going to MCU
end mcu_output;

architecture behavioural of mcu_output is
```

```vhdl
type mem_type is array (0 to 31) of std_logic_vector(7 downto 0);
signal outbuff: mem_type;
signal reading, sending, once: std_logic;
signal send_count, in_count: integer;
begin
	process (CLK, RESET)
	begin
		if (RESET='1') then
--check bit to make sure that send_count is only incremented once per byte sent (req may be asserted for more than one clock cycle)
			once<='0';

			in_count<=0;			--counter for inputting the 100 bytes to outbuff
			send_count<=0;			--counting the number of bytes sent to MCU so far
			ack<='0';			--ack pin to MCU
			WR<='1';			--write/read bit for FIFO
			reading<='0';			--signals that data is currently being read in from FIFO
			sending<='0';			--signals that data is in outbuff waiting to be sent
			LEDR(2 downto 0)<="000";
			LEDR(3)<='1';
		elsif (CLK'event and CLK='1') then
			LEDR(3)<='0';
			--not sending mode
			if (sending='0') then
				if (data_avail='1') then
					reading<='1';
					LEDR(0)<='1';
				else
					reading<='0';
					LEDR(0)<='0';
				end if;

				--reading mode (from FIFO)
				if (reading='1') then
					if (in_count>=0) then
						WR <= '0';
					else
						WR <= '1';
					end if;

					if (in_count>=0 and in_count<=31) then
						outbuff(in_count)<=data_in;
						in_count<=in_count+1;
						reading<='1';
						sending<='0';
						LEDR(0)<='1';
					else
						in_count<=0;
						sending<='1';
						reading<='0';
						LEDR(0)<='0';
					end if;
				end if;

			--sending mode
			elsif (sending ='1') then
				WR <= '1';
				if (send_count<=31) then
					LEDR(1)<='1';
					if (req = '1') then
						once<='1';
						if (once='0') then
							send_count<=send_count+1;
						end if;
						ack<='0';
						LEDR(2)<='0';
```

```
                        else
                                once<='0';
                                ack <= '1';
                                LEDR(2)<='1';
                                data_out<=outbuff(send_count);
                        end if;
                        --when done sending, turn off ack
                elsif(send_count>31) then
                        send_count<=0;
                        ack<='0';
                        LEDR(2)<='0';
                        sending<='0';
                        LEDR(1)<='0';
                end if;
            end if;
        end if;
    end process;
end behavioural;
```

B.3.3 rs485_fifo.vhd

```
--------------------------------------------------------------------------
--File Name: rs485_fifo.vhd
--Description: FIFO RAM Module for the RS485 Buffers
--Author: Christopher Rennick
--Date: April 12, 2009
--Simulator: Altera Quartus II
--------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

-- memory size is based on an assumption of 100 byte packets
-- capability to store 128 packets is desired


entity rs485_fifo is
        port(CLK, EN, WR, RES: in std_logic;
        data_in: in std_logic_vector(7 downto 0);          --data in from rs485_buf modules
        data_avail: out std_logic;
        LEDR: out std_logic_vector(2 downto 0);
        data_out: out std_logic_vector(7 downto 0));       --data output to the mcu_output module
end rs485_fifo;

architecture behavioural of rs485_fifo is
--the current address being read from and written to
signal read_addr, write_addr: std_logic_vector(39 downto 0);

--the signals for the address lines going to the RAM megafunction
signal address_a_sig, address_b_sig: std_logic_vector(9 downto 0);

signal wren_a_sig: std_logic;                   --the write enable function for the 'a' port of the RAM megafunction
signal q_a_sig: std_logic_vector(7 downto 0);           --the 'a' output port of the RAM megafunction
begin
        --instantiation of Quartus II derived dual-port RAM megafunction
        fifo_ram_inst : entity work.rs485_fifo_ram
        PORT MAP (
                address_a  => address_a_sig,
                address_b  => address_b_sig,
                clock      => CLK,
                data_a     => data_in,
                data_b     => "00000000",
                wren_a     => wren_a_sig,
                wren_b     => '0',
                q_a        => q_a_sig,
```

```
		q_b		=> data_out
	);

	process (CLK, RES)
	begin
		if (RES='1') then
		--the reset command instructions
			read_addr<=X"0000000000";
			write_addr<=X"0000000000";
			address_a_sig<="0000000000";
			data_avail<='0';
			wren_a_sig<='0';
			LEDR<="000";
		elsif (CLK'event and CLK='1') then

			--determines if the data_avail flag should be set or not
			if ((to_integer(unsigned(write_addr))-to_integer(unsigned(read_addr)))>=32) then
				data_avail<='1';
				LEDR(2)<='1';
			else
				data_avail<='0';
				LEDR(2)<='0';
			end if;

			--write
			if (EN='1') then
				address_a_sig<=write_addr(9 downto 0);
				wren_a_sig<='1';
				write_addr<=std_logic_vector(to_unsigned((to_integer(unsigned(write_addr))+1),40));
				LEDR(1 downto 0)<="01";
			else
				wren_a_sig<='0';
				address_a_sig<="0000000000";
			end if;

			--read
			if (WR='0') then
				address_b_sig<=read_addr(9 downto 0);
				read_addr<=std_logic_vector(to_unsigned((to_integer(unsigned(read_addr))+1),40));
				LEDR(1 downto 0)<="10";
			end if;

		end if;
	end process;
end behavioural;
```

B.3.4 arbiter.vhd

```
------------------------------------------------
--File Name: arbiter.vhd
--Description: Rotating Priority Arbiter Module
--Author: Christopher Rennick
--Date: April 12, 2009
--Simulator: Altera Quartus II
------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity arbiter is
	port(CLK, RES, busy: in std_logic;
	data_avail: in std_logic_vector(15 downto 0);		--data_avail signals from rs485_buf modules
	LEDR: out std_logic_vector(1 downto 0);
	EN: out std_logic_vector(15 downto 0));			--EN bus output to rs485_buf modules
```

```
end arbiter;

architecture behavioural of arbiter is
signal en_array: bit_vector(15 downto 0);
signal count: integer;
begin
	process(CLK, RES)
	begin
		if (RES='1') then
			count<=0;
			en_array<=X"0001";
			LEDR<="00";
		elsif (CLK'event and CLK='1') then
			--busy is tied to WR signal of rs485_fifo module, if busy=0, the module is being read from
			if (busy='1') then
				--checking if no one has data available to send to fifo, rotate en_array
				if ((en_array and To_bitvector(data_avail))=X"0000") then
					count<=0;
					EN<=X"0000";
					en_array<=en_array rol 1;
				else
					--when count>34, disable any active rs485_buf modules
					if (count>34) then
						count<=0;
						EN<=X"0000";
						en_array<=en_array rol 1;
					--enable appropriate rs485_buf module
					else
						count<=count+1;
						en_array<=en_array;
						EN<=To_StdLogicVector(en_array and To_bitvector(data_avail));
					end if;
				end if;
			--if rs485_fifo module is busy, do nothing
			else
				EN<=X"0000";
				en_array<=en_array;
				count<=count;
			end if;

		end if;
	end process;
end behavioural;
```

B.3.5 rs485_buf.vhd

```
-----------------------------------------------------------------
--File Name: rs485_buf.vhd
--Description: This Module Buffers Serial Data From The RS485
--Connection Before Sending It To The FIFO RAM Block
--Author: Christopher Rennick
--Date: March 17, 2009
--Simulator: Altera Quartus II
-----------------------------------------------------------------


library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;

entity rs485_buf is
	port(CLK, RESET: in std_logic;
	busy: in std_logic;
	data_in: in std_logic;			--input from one of the rs-485 network buses
	data_avail: out std_logic;
	mem_en: out std_logic;
```

```
		data_out: out std_logic_vector(7 downto 0));		--tri-state output to the rs485_fifo module
end rs485_buf;

architecture behavioural of rs485_buf is
signal buff: std_logic_vector(7 downto 0);
signal go_clk, data_avail_sig: std_logic;
signal buffswitch, send_buffswitch: std_logic_vector(1 downto 0);
signal clock_div, clock_div_count: integer;
signal count, count1, send_count: integer;
signal data_a_sig, q_a_sig, q_b_sig: std_logic_vector(7 downto 0);
signal address_a_sig, address_b_sig: std_logic_vector(6 downto 0);
signal wren_a_sig: std_logic;
signal check: integer;
begin

	--instantiation of RAM megafunction with 128 byte capacity (4 32 byte packages)
	rs485_ram_inst : entity work.rs485_ram
	PORT MAP (
		address_a	=> address_a_sig,
		address_b	=> address_b_sig,
		clock		=> CLK,
		data_a		=> data_a_sig,
		data_b		=> "00000000",
		wren_a		=> wren_a_sig,
		wren_b		=> '0',
		q_a		=> q_a_sig,
		q_b		=> q_b_sig
	);

	process(CLK, RESET, data_in)
	begin
		if (RESET='1') then
			send_buffswitch<="00";		--dictates which half of memory is being read from
			buffswitch<="00";		--dictates which half of memory are being written to
			--used to calculate an average of all sampled inputs to ensure no bounces in signal were read
			check<=0;

			go_clk <= '0';		--used to indicate we're looking for the middle of the incoming data pulse
			count1 <= 0;				--counting out incoming bits being saved to 8 bit buffer
			count <= 0;				--counting bytes being saved to 100 byte packet buffer
			send_count <= 0;		--counter for sending data out of module to FIFO
			clock_div <= 0;				--clock divider for syncing with incoming RS485 signal
			data_avail <= '0';		--signal to arbiter to say data is ready to be sent to FIFO
			data_avail_sig <= '0';
			data_out<="ZZZZZZZZ";
			mem_en <= 'Z';
			clock_div_count<=0;

			--the following are for testing memory write performance in simulation to save time
			--count <=29;

		elsif (CLK='1' and CLK'event) then
			if (data_in='0' or go_clk='1' or clock_div_count=10) then
				--Baud rate is 100K Baud
				--Examining 10 divisions of 25 clock cycles, find average value
				--of these to ensure it wasn't triggered by a 'bounce'
				if (count<32) then
					wren_a_sig<='0';
					if (clock_div=24) then
						clock_div<=0;
						clock_div_count<=clock_div_count+1;
						if (data_in='1') then
							--count of 'ones' in the input data stream
							--used to find the "average" value of data_in
							check<=check+1;
						end if;
```

```
					elsif (clock_div_count/=10 or go_clk='1') then
						clock_div<=clock_div+1;
					end if;

					--clock_div_count=10 when we reach a full 250 cycles (10*25)
					if (clock_div_count=10) then
						clock_div_count<=0;

						--go_clk=0 when we are looking at the start bit, for all other data
						--bits, go_clk=1
						if (go_clk='0') then
							buff<="00000000";
							if (check<5) then	--check if average value is 0
								go_clk<='1';
								check<=0;
							end if;
						else
							--checking if we are still looking at a data bit or the
							--stop bit
							if (count1<=7) then
								if (check<4) then
									--average input value is 0
									buff(7-count1) <= '0';
									check<=0;
								else
									--average input value is 1
									buff(7-count1) <= '1';
									check<=0;
								end if;
								count1<=count1+1;
							else
							--we are now at the stop bit
								check<=0;
								go_clk<='0';
								address_a_sig<=buffswitch &
							std_logic_vector(to_unsigned(count,5));
								wren_a_sig<='1';
								data_a_sig<=buff;
								count1<=0;
								count<=count+1;
							end if;
						end if;
					end if;
				else
					count<=0;
					data_avail<='1';
					data_avail_sig<='1';
					wren_a_sig<='0';
					data_a_sig<="00000000";
				buffswitch<=std_logic_vector(to_unsigned((to_integer(unsigned(buffswitch))+1),2));
				end if;
			end if;

			--this is to send the data serially, byte by byte to the FIFO RAM block
			if (busy='1') then
				if (send_count =0) then
				address_b_sig<=send_buffswitch & std_logic_vector(to_unsigned(send_count,5));
					send_count<=send_count+1;
				elsif (send_count =1) then
					send_count<=send_count+1;
					mem_en<='1';
				address_b_sig<=send_buffswitch & std_logic_vector(to_unsigned(send_count,5));
				elsif (send_count >1 and send_count<=31) then
					mem_en<='1';
				address_b_sig<=send_buffswitch & std_logic_vector(to_unsigned(send_count,5));
					data_out<=q_b_sig;
```

```
							send_count<=send_count+1;
						elsif (send_count >31 and send_count<=32) then
							data_out<=q_b_sig;
							mem_en<='1';
							send_count<=send_count+1;
						elsif (send_count=33) then
							data_out<=q_b_sig;
							mem_en<='Z';
							send_count<=send_count+1;
						else			--done sending, get everything ready for next packet
							send_count<=0;
							count<=0;
							mem_en<='0';
							data_avail<='Z';
							data_avail_sig<='0';
							data_out<="ZZZZZZZZ";
							send_buffswitch<=std_logic_vector(to_unsigned((to_integer(unsigned
						(send_buffswitch))+1),2));
						end if;
					end if;
				end if;
			end process;
end behavioural;
```

B.4 Server_test.vhd

```
------------------------------------------------------------
--File Name: server_test.vhd
--Description: Test Server Verification Wrapper Module
--This Module Simulates The Server Being Connected To The MCU
--Author: Christopher Rennick
--Date: May 10, 2009
--Simulator: Altera Quartus II
------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity server_test is
		port(CLK, RESET: in std_logic;
		in_port: in std_logic_vector(15 downto 0);
		rs485out: out std_logic_vector(15 downto 0);
		MCU_OUT: out std_logic_vector(7 downto 0);
		LEDG: out std_logic_vector(7 downto 0);
		LEDR: out std_logic_vector(17 downto 0));
end server_test;

architecture behavioural of server_test is
signal req_mcu_buf_sig, req_mcu_output_sig, ack_mcu_buf_sig, ack_mcu_output_sig: std_logic;

signal addr: std_logic_vector (3 downto 0);
signal sending, firstsend: std_logic;

signal send_count1, send_count2: integer;
type mem_type is array (0 to 31) of std_logic_vector(7 downto 0);
signal outbuff: mem_type;

signal data_in: std_logic_vector(7 downto 0);
signal reading: std_logic;
signal read_count: integer;

signal switch: std_logic;

signal DCLK: std_logic;
```

```
signal clock_div: integer;

begin
	server_entity1: entity work.server
	port map(CLK => CLK, RESET => RESET, req_mcu_buf => req_mcu_buf_sig,
	req_mcu_output => req_mcu_output_sig, MCU_in => data_in,
	in_port => in_port,
	ack_mcu_buf => ack_mcu_buf_sig, ack_mcu_output => ack_mcu_output_sig,
	rs485out => rs485out,
	MCU_OUT => MCU_OUT,
	LEDG => LEDG, LEDR => LEDR);

	process(CLK, RESET)
	begin
		if (RESET='0') then
			DCLK<='0';
			clock_div<=0;
		elsif (CLK'event and CLK='1') then
			DCLK<=not(DCLK);

			--clock divider circuitry used to verify functionality on physical board
			--Note: to use clock divider, CLK signal to server_entity1 needs to be changed
			--to DCLK signal

			--clock_div<=clock_div+1;
			--if (clock_div=32) then
			--		DCLK<=not(DCLK);
			--		clock_div<=0;
			--end if;
		end if;
	end process;

	process(DCLK, RESET)
	begin
		if (RESET='0') then
			req_mcu_buf_sig<='0';
			req_mcu_output_sig<='0';
			reading<='0';
			read_count<=0;
			data_in<="00000001";
			switch<='0';
		elsif (DCLK'event and DCLK='1') then
			if (reading='0') then
				--writing to FPGA and out rs485
				if (read_count=33) then
					read_count<=0;
					if (switch='1' and ack_mcu_output_sig='1') then
						reading<='1';
					end if;

					switch<=not(switch);
					req_mcu_buf_sig<='0';
				elsif (read_count=0) then
					data_in<="00000001";
					req_mcu_buf_sig<='1';
					read_count<=read_count+1;
				end if;

				if (ack_mcu_buf_sig='1') then
					if (switch='0') then
						--packet to turn on motherboard power
						if (read_count=1) then
							data_in<="00010000";
							read_count<=read_count+1;
						elsif (read_count=30) then
							data_in<=X"DC";
```

```
										read_count<=read_count+1;
								elsif (read_count=31) then
										data_in<=X"D5";
										read_count<=read_count+1;
								elsif (read_count=32) then
										read_count<=read_count+1;
										req_mcu_buf_sig<='0';
								else
										data_in<="00000000";
										read_count<=read_count+1;
								end if;
						else
								--packet to turn off motherboard power
								if (read_count=1) then
										data_in<="00010001";
										read_count<=read_count+1;
								elsif (read_count=30) then
										data_in<=X"2C";
										read_count<=read_count+1;
								elsif (read_count=31) then
										data_in<=X"33";
										read_count<=read_count+1;
								elsif (read_count=32) then
										read_count<=read_count+1;
										req_mcu_buf_sig<='0';
								else
										data_in<="00000000";
										read_count<=read_count+1;
								end if;
						end if;
				end if;
		else
				--writing from FPGA to MCU
				if (ack_mcu_output_sig='1') then
						if (read_count=32) then
								read_count<=0;
								reading<='0';
								req_mcu_output_sig<='0';
						else
								req_mcu_output_sig<='1';
								read_count<=read_count+1;
								reading<='1';
						end if;
				else
						req_mcu_output_sig<='0';
						read_count<=read_count;
						reading<=reading;
				end if;
		end if;
	end if;
  end process;
end behavioural;
```

Appendix C. Test Server C Code

```
/*********************************************************************
 *
 *  Main Application Entry Point
 *  Module for Microchip TCP/IP Stack and Test Server Functionality
 *   -Communicates over TCP/IP socket at IP address as given on LCD of
 *   Development Kit
 *   -Calculates the CRC check value for all outgoing and incoming
 *   packets
 *  -Transmits to the FPGA and accepts transmissions from the FPGA
 *
 *  Code is based off of TCPIP Demo App created by Microchip and
 *  available for free with their TCP/IP Stack Software
 *********************************************************************
 * FileName:        Main.c
 * Dependencies:    TCPIP.h
 * Processor:       PIC18
 * Author:         Christopher Rennick
 *
 * Version of Microchip TCP/IP Stack in Use:
 * Author              Date          Comment
 *~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
 * Howard Schlunder            07/10/09     Rev. 5.10
 ********************************************************************/
/*
 * This macro uniquely defines this file as the main entry point.
 * There should only be one such definition in the entire project,
 * and this file must define the AppConfig variable as described below.
 */
#define THIS_IS_STACK_APPLICATION

// Include all headers for any enabled TCPIP Stack functions
#include "TCPIP Stack/TCPIP.h"

// Include functions specific to this stack application
#include "Main.h"

#include <stdio.h>
#include <string.h>

// Declare AppConfig structure and some other supporting stack variables
APP_CONFIG AppConfig;
BYTE AN0String[8];

// Use UART2 instead of UART1 for stdout (printf functions).  Explorer 16
// serial port hardware is on PIC UART2 module.
#if defined(EXPLORER_16)
        int __C30_UART = 2;
#endif
/*****************************************************************************/
//Initialize constants for Berkeley Sockets
#define PORTNUM 9764
// Maximum number of simultanous connections accepted by the server.
#define MAX_CLIENT (1)
#define PACKETLENGTH 30
/*****************************************************************************/

/*****************************************************************************/
//These variables are the storage for outgoing and incoming packets
#pragma udata my_section_1
unsigned char sending_data_buffer[30];
#pragma udata my_section_2
unsigned char receiving_data_buffer[30];

//This variable is the hex character string conversion of receiving_data_buffer
```

```
#pragma udata my_section_3
char dataString[PACKETLENGTH];

int CRC_value;
/*******************************************************************************/

/*******************************************************************************/
// Function declarations.
static void InitAppConfig(void);
static void InitializeBoard(void);
int BerkeleyTCPServer(char *bfr, char *trans_bfr, int total);
void Initialize(void);
void crc16(unsigned char data);
void FPGA_transmit(void);
void FPGA_receive(void);
/*******************************************************************************/

// PIC18 Interrupt Service Routines
#if defined(__18CXX)
        #if defined(HI_TECH_C)
        void interrupt low_priority LowISR(void)
        #else
        #pragma interruptlow LowISR
        void LowISR(void)
        #endif
        {
            TickUpdate();
        }

        #if defined(HI_TECH_C)
        void interrupt HighISR(void)
        #else
        #pragma interruptlow HighISR
        void HighISR(void)
        #endif
        {
            #if defined(STACK_USE_UART2TCP_BRIDGE)
                        UART2TCPBridgeISR();
                #endif

                #if defined(ZG_CS_TRIS)
                        zgEintISR();
                #endif // ZG_CS_TRIS
        }

        #if !defined(HI_TECH_C)
        #pragma code lowVector=0x18
        void LowVector(void){_asm goto LowISR _endasm}
        #pragma code highVector=0x8
        void HighVector(void){_asm goto HighISR _endasm}
        #pragma code // Return to default code section
        #endif

// C30 and C32 Exception Handlers
// If your code gets here, you either tried to read or write
// a NULL pointer, or your application overflowed the stack
// by having too many local variables or parameters declared.
#elif defined(__C30__)
        void _ISR __attribute__((__no_auto_psv__)) _AddressError(void)
        {
            Nop();
                Nop();
        }
        void _ISR __attribute__((__no_auto_psv__)) _StackError(void)
        {
            Nop();
                Nop();
        }

#elif defined(__C32__)
        void _general_exception_handler(unsigned cause, unsigned status)
```

```
        {
                Nop();
                Nop();
        }
#endif

/*********************************************************************
  Function:
    void main(void)

  Description:
    This is the main function where all other functions are called from

  Precondition:
    None

  Parameters:
    None - None

  Returns:
    None

  Remarks:
    This function was originally part of the Microchip TCP/IP Stack Software
  *********************************************************************/
void main(void)
{
        static TICK t = 0;
        static DWORD dwLastIP = 0;
        int i;
        int total;
        char bfr[67];

        // Initialize application specific hardware
        InitializeBoard();
        Initialize();

        #if defined(USE_LCD)
        // Initialize and display the stack version on the LCD
        LCDInit();
        DelayMs(100);
        strcpypgm2ram((char*)LCDText, "CTCPStack " VERSION "   "
                "                 ");
        LCDUpdate();
        #endif

        // Initialize stack-related hardware components that may be
        // required by the UART configuration routines
    TickInit();

        // Initialize Stack and application related NV variables into AppConfig.
        InitAppConfig();

    // Initiates board setup process if button is depressed
        // on startup
    if(BUTTON0_IO == 0u)
    {
                #if defined(EEPROM_CS_TRIS) || defined(SPIFLASH_CS_TRIS)
                // Invalidate the EEPROM contents if BUTTON0 is held down for more than 4
                //seconds
                TICK StartTime = TickGet();
                LED_PUT(0x00);

                while(BUTTON0_IO == 0u)
                {
                        if(TickGet() - StartTime > 4*TICK_SECOND)
                        {
                                #if defined(EEPROM_CS_TRIS)
                            XEEBeginWrite(0x0000);
                            XEEWrite(0xFF);
                            XEEEndWrite();
```

```
                        #elif defined(SPIFLASH_CS_TRIS)
                        SPIFlashBeginWrite(0x0000);
                        SPIFlashWrite(0xFF);
                        #endif

                            #if defined(STACK_USE_UART)
                            putrsUART("\r\n\r\nBUTTON0 held for more than 4
seconds.\r\n\r\n");
                            #endif

                            LED_PUT(0x0F);
                            while((LONG)(TickGet() - StartTime) <=
(LONG)(9*TICK_SECOND/2));
                            LED_PUT(0x00);
                            while(BUTTON0_IO == 0u);
                            Reset();
                            break;
                    }
                }
                #endif
    }

        // Initialize core stack layers (MAC, ARP, TCP, UDP) and
        // application modules (HTTP, SNMP, etc.)
    StackInit();

        // This infinite loop will continuously execute all
        // stack-related tasks and also application functions.
    // Note that this is a "co-operative mult-tasking" mechanism
    // where every task performs its tasks (whether all in one shot
    // or part of it) and returns so that other tasks can do their
    // job.
    // If a task needs very long time to do its job, it must be broken
    // down into smaller pieces so that other tasks can have CPU time.
    total=0;
        while(1)
    {
        // Blink LED0 (right most one) every second.
        if(TickGet() - t >= TICK_SECOND/2ul)
        {
            t = TickGet();
            LED0_IO ^= 1;
        }

        // This task performs normal stack task including checking
        // for incoming packet, type of packet and calling
        // appropriate stack entity to process it.
        StackTask();

        // This task invokes each of the core stack application tasks
        StackApplications();

                // Process application specific tasks here.
                if (PACKETLENGTH-total>0 && (i=
BerkeleyTCPServer(&(receiving_data_buffer[total]), &(sending_data_buffer[total]),
(PACKETLENGTH-total)))>0)
                        total+=i;

                if(total==PACKETLENGTH)
        {
                        CRC_value=0xFFFF;
                        //Calculate the CRC value for the outgoing data
                        for (i=0;i<=29; i++)
                        {
                                crc16(receiving_data_buffer[i]);
                        }

                        //Transmit the outgoing data to the FPGA
                        FPGA_transmit;

                        //Check to see if data is waiting to be input from the FPGA and
```

```
                        //input any data that is waiting
                        FPGA_receive;

                        total=0;
                }

                // If the local IP address has changed (ex: due to DHCP lease change)
                // write the new IP address to the LCD display, UART, and Announce
                // service
                if(dwLastIP != AppConfig.MyIPAddr.Val)
                {
                        dwLastIP = AppConfig.MyIPAddr.Val;

                        #if defined(STACK_USE_UART)
                                putrsUART((ROM char*)"\r\nNew IP Address: ");
                        #endif

                        DisplayIPValue(AppConfig.MyIPAddr);

                        #if defined(STACK_USE_UART)
                                putrsUART((ROM char*)"\r\n");
                        #endif


                        #if defined(STACK_USE_ANNOUNCE)
                                AnnounceIP();
                        #endif
                }
        }
}

/*********************************************************************
  Function:
    static void Initialize(void)

  Description:
    This routine initializes the MCU I/O port directionality as well as
        initialize the request lines to 0

  Precondition:
    None

  Parameters:
    None - None

  Returns:
    None

  Remarks:
    None
  ********************************************************************/
void Initialize(void)
{
        CRC_value=0;

        //Initialize IO ports for input/output to FPGA
        //B4 is ack1, B5 is req1, B6 is ack2, B7 is req2
        ack1_en=1;
        ack2_en=1;
        req1_en=0;
        req2_en=0;
        //initialize request lines to zero so unwanted data is not transmitted to the FPGA
        req1=0;
        req2=0;

        //PORTC is used for outputting the 8 bit parallel data to the FPGA
        mcu_out_en=0x00;
        mcu_out=0x00;

        //PORTD is used for accepting the 8 bit parallel data from the FPGA
        mcu_in0_en=1;
```

```
        mcu_in1_en=1;
        mcu_in2_en=1;
        mcu_in3_en=1;
        mcu_in4_en=1;
        mcu_in5_en=1;
        mcu_in6_en=1;
        mcu_in7_en=1;
}

/**************************************************************************
  Function:
    static void crc16(int data)

  Description:
        This function updates the CRC16 value for the outgoing packets by
        accepting the output data one byte at a time

  Precondition:
        CRC_value must be initialized to 0xFFFF before function call

  Parameters:
    Inputs:    integer data to be added to the CRC value
        Outputs: None

  Returns:
    None

  Remarks:
    None
  *************************************************************************/
void crc16(unsigned char data)
{
   // y: Temporary calculation for 16-bit arithmetic
   // x: Temporary calculation for 8-bit arithmetic
   long y;
   int x;

   x = (CRC_value >> 8) ^ data;
   x ^= x >> 4;
   y = (long)x << 4;
   // Update the CRC based on the new byte
   CRC_value = (CRC_value << 8) ^ (y << 8) ^ (y << 1) ^ x;
}

/**************************************************************************
  Function:
    void FPGA_transmit(void)

  Description:
        This function transmits the data to the FPGA one byte at a time, including
        the CRC value.  It does this using a simple request/acknowledge handshake
        with the FPGA hardware.

  Precondition:
        CRC_value must be determined before function call

  Parameters:
    Inputs:    None
        Outputs: None

  Returns:
    None

  Remarks:
    If no response is heard upon initial setting of the request line after a
        count of 255, the function exits.  The client will notice no response was
        heard for this packet and will re-transmit at a later time.
  *************************************************************************/
void FPGA_transmit(void)
{
        int i;
```

```
	int count;
	short int done;

	count=0;
	done=0;
	i=0;

	while (done==0)
	{
		req1=1;

		//wait for a response from the FPGA, if none is received before
		//count reaches 255, exit function
		while(ack1==0 && done==0)
		{
			if (i==0)
			{
				count++;
				if (count==255)
					done=1;
			}
		}

		//Transmit the data bytes to the FPGA and de-assert req
		if (i<=29 && done==0)
		{
			mcu_out=receiving_data_buffer[i]&255;
			req1=0;
			i++;
		}
		//transmit the high order byte of the CRC
		else if (i==30 && done==0)
		{
			mcu_out=(CRC_value>>8)&255;
			req1=0;
			i++;
		}
		//tranmit the low order byte of the CRC
		else
		{
			mcu_out=CRC_value&255;
			req1=0;
			CRC_value=0xFFFF;
			done=1;
		}
	}
}

/*************************************************************************
  Function:
    void FPGA_receive(void)

  Description:
      This function accepts data from the FPGA using the same acknowledge/request
      handshake method used to transmit data.  Data is accepted one byte at a time
      and when all packet data has been received, the CRC value is checked to
      ensure there were no transmission errors anywhere from the test node to here.
      If the CRC does not match, the ACK value of the packet is changed to 170 in
      decimal.

  Precondition:
    None

  Parameters:
    Inputs:    None
      Outputs: None

  Returns:
    None

  Remarks:
```

```
    This function waits for a count of 256 before leaving the function.  It is
        assumed that if the ack line is not asserted before then that the FPGA
        simply has no data to be sent back to the client program.
  ********************************************************************/
void FPGA_receive(void)
{
        int i;
        int exit, count;

        exit=0;
        count=0;
        i=0;

        //accept data from FPGA
        while (exit=0)
        {
                if (i<=0)
                        count++;
                //if ack is not accepted by a count of 256, exit function
                else if (i==32 || count==256)
                        exit=1;
                else
                {
                        if (ack2)
                        {
                                //sending_data_buffer[i]=mcu_in;
                                sending_data_buffer[i]=mcu_in0+(mcu_in1<<1)+(mcu_in2<<2)+(m
                        cu_in3<<3)+(mcu_in4<<4)+(mcu_in5<<5)+(mcu_in6<<6)+(mcu_in7<<7);
                                i++;
                                req2=1;
                        }
                }
        }

        //initialize CRC value
        CRC_value=0xFFFF;

        //check and compare CRC value, if different, set ACK to 170
        if (i==32)
        {
                for (i=0; i<=29; i++)
                {
                        crc16(sending_data_buffer[i]);
                }
                if ((CRC_value>>8)&255!= sending_data_buffer[30] ||
(CRC_value&255)!=sending_data_buffer[31])
                {
                        sending_data_buffer[1]=170;
                }
        }
}

/*********************************************************************
  Function:
    int BerkeleyTCPServer(char *bfr, char *trans_bfr, int total)

  Description:
        This function handles all socket communication with the PC Client program.

  Precondition:
    Stack and Berkeley API must be initialized prior to function call

  Parameters:
    Inputs:    None
        Outputs: None

  Returns:
    None

  Remarks:
        This function was originally part of the Microchip TCP/IP Stack software
```

```
  Internal Function Call:
    The ConvertandStore function is called from within after data has been
        received over the socket.
  ***********************************************************************/
int BerkeleyTCPServer(char *bfr, char *trans_bfr, int total)
{
    static SOCKET bsdServerSocket;
    static SOCKET ClientSock[MAX_CLIENT];
    struct sockaddr_in addr;
    struct sockaddr_in addRemote;
    int addrlen = sizeof(struct sockaddr_in);
    int length;
    int i, j;
    static enum
    {
            BSD_INIT = 0,
        BSD_CREATE_SOCKET,
        BSD_BIND,
        BSD_LISTEN,
        BSD_OPERATION
    } BSDServerState = BSD_INIT;

    switch(BSDServerState)
    {
            case BSD_INIT:
                // Initialize all client socket handles so that we don't process
                // them in the BSD_OPERATION state
                for(i = 0; i < MAX_CLIENT; i++)
                        ClientSock[i] = INVALID_SOCKET;

                BSDServerState = BSD_CREATE_SOCKET;
                // No break needed

        case BSD_CREATE_SOCKET:
            // Create a socket for this server to listen and accept connections on
            bsdServerSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
            if(bsdServerSocket == INVALID_SOCKET)
                return -1;

            BSDServerState = BSD_BIND;
            // No break needed

        case BSD_BIND:
            // Bind socket to a local port
            addr.sin_port = PORTNUM;
            addr.sin_addr.S_un.S_addr = IP_ADDR_ANY;
            if( bind( bsdServerSocket, (struct sockaddr*)&addr, addrlen ) == SOCKET_ERROR
)
                return -1;

            BSDServerState = BSD_LISTEN;
            // No break needed

      case BSD_LISTEN:
            if(listen(bsdServerSocket, MAX_CLIENT) == 0)
                    BSDServerState = BSD_OPERATION;

                        // No break.  If listen() returns SOCKET_ERROR it could be because
                        // MAX_CLIENT is set to too large of a backlog than can be handled
                        // by the underlying TCP socket count (TCP_PURPOSE_BERKELEY_SERVER
                        // type sockets in TCPIPConfig.h).  However, in this case, it is
                        // possible that some of the backlog is still handleable, in which
                        // case we should try to accept() connections anyway and proceed
                        // with normal operation.

      case BSD_OPERATION:
                        for(i=0; i<MAX_CLIENT; i++)
            {
                // Accept any pending connection requests, assuming we have a place to
                //store the socket descriptor
```

```
                if(ClientSock[i] == INVALID_SOCKET)
                    ClientSock[i] = accept(bsdServerSocket, (struct sockaddr*)&addRemote,
&addrlen);

                // If this socket is not connected then no need to process anything
                if(ClientSock[i] == INVALID_SOCKET)
                      continue;

                    // For all connected sockets, receive and send back the data
                              length = recv( ClientSock[i], bfr, total, 0);

               if (length< 0)
                              {
                                      //close the socket connection
                                      closesocket( ClientSock[i] );
                                      ClientSock[i] = INVALID_SOCKET;
                              }
                              else if(length>0)
                              {
                                      send(ClientSock[i], trans_bfr, length, 0);
                              }
                              return length;
                      }

             break;

        default:
             return 0;
    }
    return 0;
}

/*********************************************************************
  Function:
    void DisplayIPValue(IP_ADDR IPVal)

  Description:
        Writes an IP address to the LCD display and the UART as available.

  Precondition:
    Stack must be initialized prior to function call

  Parameters:
    Inputs:    None
        Outputs: None

  Returns:
    None

  Remarks:
    This function was originally part of the Microchip TCP/IP Stack software
  ***********************************************************************/
void DisplayIPValue(IP_ADDR IPVal)
{
//      printf("%u.%u.%u.%u", IPVal.v[0], IPVal.v[1], IPVal.v[2], IPVal.v[3]);
    BYTE IPDigit[4];
        BYTE i;
#ifdef USE_LCD
        BYTE j;
        BYTE LCDPos=16;
#endif

        for(i = 0; i < sizeof(IP_ADDR); i++)
        {
            uitoa((WORD)IPVal.v[i], IPDigit);

                #if defined(STACK_USE_UART)
                        putsUART(IPDigit);
                #endif

                #ifdef USE_LCD
```

```
                        for(j = 0; j < strlen((char*)IPDigit); j++)
                        {
                                LCDText[LCDPos++] = IPDigit[j];
                        }
                        if(i == sizeof(IP_ADDR)-1)
                                break;
                        LCDText[LCDPos++] = '.';
                #else
                        if(i == sizeof(IP_ADDR)-1)
                                break;
                #endif

                #if defined(STACK_USE_UART)
                        while(BusyUART());
                        WriteUART('.');
                #endif
        }

        #ifdef USE_LCD
                if(LCDPos < 32u)
                        LCDText[LCDPos] = 0;
                LCDUpdate();
        #endif
}


/*****************************************************************************
  Function:
    static void InitializeBoard(void)

  Description:
    This routine initializes the hardware.  It is a generic initialization
    routine for many of the Microchip development boards, using definitions
    in HardwareProfile.h to determine specific initialization.

  Precondition:
    None

  Parameters:
    None - None

  Returns:
    None

  Remarks:
    This function was originally part of the Microchip TCP/IP Stack software
  ***************************************************************************/
static void InitializeBoard(void)
{
        // LEDs
        LED0_TRIS = 0;
        LED1_TRIS = 0;
        LED2_TRIS = 0;
        LED3_TRIS = 0;
        LED4_TRIS = 0;
        LED5_TRIS = 0;
        LED6_TRIS = 0;
//#if !defined(EXPLORER_16)    // Pin multiplexed with a button on EXPLORER_16
        LED7_TRIS = 0;
//#endif
        LED_PUT(0x00);

        // Enable 4x/5x/96MHz PLL on PIC18F87J10, PIC18F97J60, PIC18F87J50, etc.
    OSCTUNE = 0x40;

        ADCON2 = 0xBE;          // Right justify, 20TAD ACQ time, Fosc/64 (~21.0kHz)

        // Enable internal PORTB pull-ups
    INTCON2bits.RBPU = 0;

        // Configure USART
```

```
    TXSTA = 0x20;
    RCSTA = 0x90;

        // See if we can use the high baud rate setting
        #if ((GetPeripheralClock()+2*BAUD_RATE)/BAUD_RATE/4 - 1) <= 255
                SPBRG = (GetPeripheralClock()+2*BAUD_RATE)/BAUD_RATE/4 - 1;
                TXSTAbits.BRGH = 1;
        #else   // Use the low baud rate setting
                SPBRG = (GetPeripheralClock()+8*BAUD_RATE)/BAUD_RATE/16 - 1;
        #endif


        // Enable Interrupts
        RCONbits.IPEN = 1;              // Enable interrupt priorities
    INTCONbits.GIEH = 1;
    INTCONbits.GIEL = 1;

        #if defined(SPIRAM_CS_TRIS)
                SPIRAMInit();
        #endif
        #if defined(EEPROM_CS_TRIS)
                XEEInit();
        #endif
        #if defined(SPIFLASH_CS_TRIS)
                SPIFlashInit();
        #endif
}

/*********************************************************************
 * Function:        void InitAppConfig(void)
 *
 * PreCondition:    MPFSInit() is already called.
 *
 * Input:           None
 *
 * Output:          Write/Read non-volatile config variables.
 *
 * Side Effects:    None
 *
 * Overview:        None
 *
 * Note:            None

 * Remarks:                   This function was originally part of the Microchip TCP/IP
Stack software
 ********************************************************************/
// MAC Address Serialization using a MPLAB PM3 Programmer and
// Serialized Quick Turn Programming (SQTP).
// The advantage of using SQTP for programming the MAC Address is it
// allows you to auto-increment the MAC address without recompiling
// the code for each unit.  To use SQTP, the MAC address must be fixed
// at a specific location in program memory.  Uncomment these two pragmas
// that locate the MAC address at 0x1FFF0.  Syntax below is for MPLAB C
// Compiler for PIC18 MCUs. Syntax will vary for other compilers.
//#pragma romdata MACROM=0x1FFF0
static ROM BYTE SerializedMACAddress[6] = {MY_DEFAULT_MAC_BYTE1, MY_DEFAULT_MAC_BYTE2,
MY_DEFAULT_MAC_BYTE3, MY_DEFAULT_MAC_BYTE4, MY_DEFAULT_MAC_BYTE5, MY_DEFAULT_MAC_BYTE6};
//#pragma romdata

static void InitAppConfig(void)
{
        AppConfig.Flags.bIsDHCPEnabled = TRUE;
        AppConfig.Flags.bInConfigMode = TRUE;
        memcpypgm2ram((void*)&AppConfig.MyMACAddr, (ROM void*)SerializedMACAddress,
sizeof(AppConfig.MyMACAddr));
//      {
//              _prog_addressT MACAddressAddress;
//              MACAddressAddress.next = 0x157F8;
//              _memcpy_p2d24((char*)&AppConfig.MyMACAddr, MACAddressAddress,
sizeof(AppConfig.MyMACAddr));
//      }
```

```
	AppConfig.MyIPAddr.Val = MY_DEFAULT_IP_ADDR_BYTE1 | MY_DEFAULT_IP_ADDR_BYTE2<<8ul
| MY_DEFAULT_IP_ADDR_BYTE3<<16ul | MY_DEFAULT_IP_ADDR_BYTE4<<24ul;
	AppConfig.DefaultIPAddr.Val = AppConfig.MyIPAddr.Val;
	AppConfig.MyMask.Val = MY_DEFAULT_MASK_BYTE1 | MY_DEFAULT_MASK_BYTE2<<8ul |
MY_DEFAULT_MASK_BYTE3<<16ul | MY_DEFAULT_MASK_BYTE4<<24ul;
	AppConfig.DefaultMask.Val = AppConfig.MyMask.Val;
	AppConfig.MyGateway.Val = MY_DEFAULT_GATE_BYTE1 | MY_DEFAULT_GATE_BYTE2<<8ul |
MY_DEFAULT_GATE_BYTE3<<16ul | MY_DEFAULT_GATE_BYTE4<<24ul;
	AppConfig.PrimaryDNSServer.Val = MY_DEFAULT_PRIMARY_DNS_BYTE1 |
MY_DEFAULT_PRIMARY_DNS_BYTE2<<8ul  | MY_DEFAULT_PRIMARY_DNS_BYTE3<<16ul  |
MY_DEFAULT_PRIMARY_DNS_BYTE4<<24ul;
	AppConfig.SecondaryDNSServer.Val = MY_DEFAULT_SECONDARY_DNS_BYTE1 |
MY_DEFAULT_SECONDARY_DNS_BYTE2<<8ul  | MY_DEFAULT_SECONDARY_DNS_BYTE3<<16ul  |
MY_DEFAULT_SECONDARY_DNS_BYTE4<<24ul;


	// SNMP Community String configuration
	#if defined(STACK_USE_SNMP_SERVER)
	{
		BYTE i;
		static ROM char * ROM cReadCommunities[] = SNMP_READ_COMMUNITIES;
		static ROM char * ROM cWriteCommunities[] = SNMP_WRITE_COMMUNITIES;
		ROM char * strCommunity;

		for(i = 0; i < SNMP_MAX_COMMUNITY_SUPPORT; i++)
		{
			// Get a pointer to the next community string
			strCommunity = cReadCommunities[i];
			if(i >= sizeof(cReadCommunities)/sizeof(cReadCommunities[0]))
				strCommunity = "";

			// Ensure we don't buffer overflow.  If your code gets stuck here,
			// it means your SNMP_COMMUNITY_MAX_LEN definition in TCPIPConfig.h
			// is either too small or one of your community string lengths
			// (SNMP_READ_COMMUNITIES) are too large.  Fix either.
			if(strlenpgm(strCommunity) >= sizeof(AppConfig.readCommunity[0]))
				while(1);

			// Copy string into AppConfig
			strcpypgm2ram((char*)AppConfig.readCommunity[i], strCommunity);

			// Get a pointer to the next community string
			strCommunity = cWriteCommunities[i];
			if(i >= sizeof(cWriteCommunities)/sizeof(cWriteCommunities[0]))
				strCommunity = "";

			// Ensure we don't buffer overflow.  If your code gets stuck here,
			// it means your SNMP_COMMUNITY_MAX_LEN definition in TCPIPConfig.h
			// is either too small or one of your community string lengths
			// (SNMP_WRITE_COMMUNITIES) are too large.  Fix either.
			if(strlenpgm(strCommunity) >= sizeof(AppConfig.writeCommunity[0]))
				while(1);

			// Copy string into AppConfig
			strcpypgm2ram((char*)AppConfig.writeCommunity[i], strCommunity);
		}
	}
	#endif

	// Load the default NetBIOS Host Name
	memcpypgm2ram(AppConfig.NetBIOSName, (ROM void*)MY_DEFAULT_HOST_NAME, 16);
	FormatNetBIOSName(AppConfig.NetBIOSName);

	#if defined(ZG_CS_TRIS)
		// Load the default SSID Name
		if (sizeof(MY_DEFAULT_SSID_NAME) > sizeof(AppConfig.MySSID))
		{
		    ZGSYS_DRIVER_ASSERT(5, (ROM char *)"AppConfig.MySSID[] too small.\n");
		}
		memcpypgm2ram(AppConfig.MySSID, (ROM void*)MY_DEFAULT_SSID_NAME,
sizeof(MY_DEFAULT_SSID_NAME));
```

```
        #endif
}
```

Appendix D. PC Client C Code

```
/*****************************************************************************/
/*      Filename: client.c
/*
/*      Description: This is the C code to run the client side of the TCP/IP Socket
/*      Connection
/*
/*      Author: Chris Rennick
/*
/*      Date: August 3, 2009
/*
/*      Notes: Adapted from C code provided in: TCP/IP Sockets in C by Donahoo, M and
/*      Calvert, K
/*
/*****************************************************************************/


//Header file calls
/*****************************************************************************/
//if Windows system
#ifdef WIN32
        #include <winsock.h>
//if Linux/Unix system
#else
        #include <sys/socket.h>
        #include <sys/types.h>
        #include <arpa/inet.h>
        #include <netinet/in.h>
#endif

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
/*****************************************************************************/


//Define system constants and input/output file pointers
/*****************************************************************************/
#define MAXSENDPACKETS 1048576
#define PACKETLENGTH 30

FILE *packets;
FILE *sorted_packets;
FILE *converted_packets;
FILE *received_packets;
/*****************************************************************************/


//Global Variable declarations
/*****************************************************************************/
//sending_data_buffer is for data that is received over ethernet waiting to be sent
//to the test nodes
unsigned int sending_data_buffer[MAXSENDPACKETS][31];
//read_data_buffer is data that has been read from the FPGA waiting to be sent over
//the ethernet connection
unsigned int receiving_data_buffer[MAXSENDPACKETS][30];

//This is the array that holds the time in seconds that a packet was sent (value
//is counted in seconds since January 1, 1970
time_t sending_data_buffer_time[MAXSENDPACKETS];

//this is the pointer that handles the address in the data_buffer for inputting
//new data from the file
int write_pointer_sending;
```

```
//This is the pointer that handles the address in the data_buffer for outputting
//data over the socket to the MCU
int read_pointer_sending;

//This is the pointer that handles the address of the data_buffer used for
//accepting data from the MCU
int write_pointer_receiving;

//This is to keep track of when the last sort on the data was performed
int lastSort;

//These are used in the sorting function.  They are kept global so that from one
//small sort to another, the bus contention issues don't happen.
int lastAddr1, lastAddr2, lastAddr3, lastAddr4;

//This is also used in the sort function to note that the above three variables
//have already been determined once
short int comparisonFull;

//This is to keep track of any nodes that haven't been communicated to in a while
short int nodeLastComm[16][16];
short int nodesConnected[16][16];

//This variable is set to 1 when the system has no new data to send, but is still
//waiting to receive data from the nodes
short int IDLE;

//This variable holds the time when the system last went into IDLE state
time_t IDLE_time;
/*****************************************************************************************/


//Function prototypes
/*****************************************************************************************/
void DieWithError(char *errorMessage);
void Initialize(void);
void InitializeNetwork(void);
void NoResponseCheck(void);
void SortPackets(void);
void ConvertandStore(unsigned char *data_in, short int sendread, int size);
int nodelastcommto(void);
void CheckReceived(void);
/*****************************************************************************************/


/****************************************************************************************
  Function:
    int main(int argc, char *argv[])

  Description:
        This is the main program entry point.  All other functions are called from within
this
        including all socket operations.

  Precondition:
    None

  Parameters:
    Inputs:     Server IP address and port number
        Outputs: None

  Returns:
    None

  Remarks:
    This code was adapted from the example code from the book TCP/IP Sockets in C
        written by Donahoo, M and Calvert, K.
 ****************************************************************************************/
int main(int argc, char *argv[])
{
```

```
	int sock, i, j, k, l;
	struct sockaddr_in dataServAddr;
	unsigned short dataServPort;
	char temp[3];
	char *servIP;
	unsigned char dataString[PACKETLENGTH];
	unsigned char dataBuffer[PACKETLENGTH];
	char data[61];
	int n;
	int dataStringLen;
	int bytesRcvd, totalBytesRcvd;

	#ifdef WIN32
		WSADATA wsaData;
	#endif

	//Openning output files
	packets=fopen("C:/Users/Chris/Desktop/ATI/C_Files/text_files/packets.dat","r");
	sorted_packets=fopen("C:/Users/Chris/Desktop/ATI/C_Files/text_files/sorted_packets
.dat","w");
	converted_packets=fopen("C:/Users/Chris/Desktop/ATI/C_Files/text_files/converted_p
ackets.dat","w");
	received_packets=fopen("C:/Users/Chris/Desktop/ATI/C_Files/text_files/received_pac
kets.dat","w");

	//routine used to initialize global variables
	Initialize();

	//routine used to send packets to check which nodes are connected
	InitializeNetwork();


	//set up socket constants for port number and IP address
	if (argc>=3)
	{
		servIP = argv[1];
		dataServPort = atoi(argv[2]);
	}

	#ifdef WIN32
		//Initialize Winsock
		if (WSAStartup(MAKEWORD(2,0), &wsaData) != 0)
			printf(stderr, "WSAStartup() failed");
	#endif

	//Open Socket
	if ((sock=socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) <0)
		DieWithError("Socket() failed");

	//setup socket constants for IP address and port as well as //the address family
	memset(&dataServAddr, 0, sizeof(dataServAddr));
	dataServAddr.sin_family = AF_INET;
	dataServAddr.sin_addr.s_addr = inet_addr(servIP);
	dataServAddr.sin_port = htons(dataServPort);

	//Connect to Socket
	if (connect(sock, (struct sockaddr *) &dataServAddr, sizeof(dataServAddr)) <0)
		DieWithError("connect() failed");

	while (1)
	{
		//read from the input file until no more data exists in the file
		if (!feof(packets))
		{
			//input the data from the hex file
			n=fread(data,1,sizeof(data),packets);
			data[sizeof(data)]='\0';
			ConvertandStore(data, 0, 60);

			//increment the sort counter
			lastSort++;
```

```
		}
		else
		{
			//if there is no more data to input from the file,
			//start entering IDLE mode
			IDLE=1;
			IDLE_time=time(NULL);
		}

		//if there are 50 new data strings, sort and send them
		if (lastSort==50 || ((time(NULL)-IDLE_time==1) && IDLE==1))
		{
			IDLE=0;

			lastSort=0;

			//Check all sent packets to see if a response was ever received
			NoResponseCheck();

			//50 or more packets are waiting to be transmitted, so sort them
			SortPackets();

			fprintf(converted_packets, "IP address and port number are: %s %s",
argv[1], argv[2]);
			fprintf(converted_packets, "\n Start of data being sent out\n");

			//Send only that data that hasn't been sent before
			while (read_pointer_sending<write_pointer_sending)
			{
				//store the value of the system clock for current
				//transmission packet to verify later that not too much
				//time has past without a response from the node
				sending_data_buffer_time[read_pointer_sending]=time(NULL);

				//convert data back into a hex number character string
				for (j=0; j<=29; j++)
dataString[j]=sending_data_buffer[read_pointer_sending][j]&255;

				read_pointer_sending++;

				dataStringLen = PACKETLENGTH;
				//Send dataString over Socket to Server
				if (send(sock, dataString, dataStringLen, 0) !=
dataStringLen)
					DieWithError("send() sent a different number of
bytes than expected");

				totalBytesRcvd = 0;

				//Receive data back from Server
				printf("%d Received\n", read_pointer_sending);

				//exit=0;

				bytesRcvd=0;

				while ( PACKETLENGTH-bytesRcvd>0 && (n=recv(sock,
dataBuffer+bytesRcvd, PACKETLENGTH-bytesRcvd , 0))>=0 )
				{
					bytesRcvd+=n;
					totalBytesRcvd++;

				};

				for (j=0; j<PACKETLENGTH; j++)
					printf("%02X", dataBuffer[j]);

				printf("\n");

				ConvertandStore(dataBuffer, 1, PACKETLENGTH);
			}
```

```
                }
        }

        #ifdef WIN32
                closesocket(sock);
                WSACleanup();
        #else
                close(sock);
        #endif

        //close output files
        fclose(packets);
        fclose(converted_packets);
        fclose(sorted_packets);

        while(1);
}


/*******************************************************************************
  Function:
    static void Initialize(void)

  Description:
       This function is used to initialize global variables

  Precondition:
    None

  Parameters:
    Inputs:    None
       Outputs: None

  Returns:
    None

  Remarks:
    None
  *****************************************************************************/
void Initialize(void)
{
        //initialize data address pointers
        write_pointer_sending=0;
        read_pointer_sending=0;
        write_pointer_receiving=0;

        //Initialize IDLE
        IDLE=0;

        //initialize sorting global variables
        lastSort=0;
        lastAddr1=0;
    lastAddr2=0;
        lastAddr3=0;
        lastAddr4=0;
        comparisonFull=0;

        //initialize nodesConnected
        for (short int i=0; i<=15; i++)
        {
                for (short int j=0; j<=15; j++) nodesConnected[i][j]=1;
        }
}


/*******************************************************************************
  Function:
    static void InitializeNetwork(void)

  Description:
       This function sends out a packet to all 256 nodes of the test network.
```

```
        It is used to detect which nodes are currently connected to the network

    Precondition:
      None

    Parameters:
      Inputs:    None
          Outputs: None

    Returns:
      None

    Remarks:
      None
  ***********************************************************************************/
void InitializeNetwork(void)
{
        int i, j;

        for (i=0; i<=255; i++)
        {
                for (j=0;j<=29;j++) sending_data_buffer[i][j]=0;
                sending_data_buffer[i][0]=i;
                sending_data_buffer[i][1]=18;
                sending_data_buffer[i][3]=27;

                write_pointer_sending++;
        }
}


/***********************************************************************************
  Function:
    void ConvertandStore(char *data_in, short int sendread, int size)

  Description:
        This function accepts a char format hex file and converts the packet to decimal
integers
        and stores the packet in the sending data buffer for output to the FPGA

  Precondition:
    None

  Parameters:
    Inputs:     - sendread flag which tells the function whether it is a packet waiting to
                be sent, or one thatwas received over the socket that is currently being
                converted
                - data_in character string which is the string to be converted
                - integer size which is the length of the character string to be converted
    Outputs: None

  Returns:
    None

  Remarks:
        Note: at this point, the data still needs to be sorted, which is why it must be in
        integer form
  ***********************************************************************************/
void ConvertandStore(unsigned char *data_in, short int sendread, int size)
{
        short int i;
        int temp_buffer[size];
        int remainder;
        int count;

        if (sendread==0)
        {
                for (i=0;i<size;i++)
                {
                        if ((data_in[i]-55)<=2)
                                temp_buffer[i]=data_in[i]-48;
```

```
                else
                        temp_buffer[i]=data_in[i]-55;
        }

        for (i=0;i<=size; i=i+2)
        {
                remainder=0;
                if (temp_buffer[i+1]>=10)
                {
                        temp_buffer[i+1]-=10;
                        remainder=1;
                }


sending_data_buffer[write_pointer_sending][i/2]=(temp_buffer[i]*16)+(remainder*10)
+temp_buffer[i+1];
        }

        for (i=0; i<=29; i++)
                fprintf(converted_packets, "%d
                ",sending_data_buffer[write_pointer_sending][i]);
        fprintf(converted_packets, "\n");

        write_pointer_sending++;

        return;
    }
    else if(sendread==1)
    {
        for (i=0; i<=size; i++)
        receiving_data_buffer[write_pointer_receiving][i]=data_in[i];


        fprintf(received_packets, "%d    ", write_pointer_receiving);
        for (i=0; i<size; i++)
                fprintf(received_packets, "%02X ",
                receiving_data_buffer[write_pointer_receiving][i]);
        fprintf(received_packets, "\n");

        write_pointer_receiving++;

        CheckReceived();

        return;
    }
}


/*********************************************************************************
  Function:
    int nodelastcommto(void)

  Description:
      This function returns the address for a node that hasn't been communicated to in a
while

  Precondition:
    nodeLastComm and nodesConnected matrices need to be initialized

  Parameters:
    Inputs:     None
      Outputs: None

  Returns:
    Returns the integer address of a node that has not been communicated with in a while

  Remarks:
    None
  ********************************************************************************/
int nodelastcommto(void)
{
```

```
        int i, j;
        int addr, buff_addr;

        for (i=0; i<=15; i++)
        {
                for (j=0; j<=15; j++)
                {
                        if(i==15 && j==15)
                        {
                                for (short int k=0; k<=15; k++)
                                {
                                        for (short int l=0; l<=15; l++)
                                                nodeLastComm[k][l]=nodesConnected[k][l];
                                }

                        }

                        if (nodeLastComm[i][j]==1)
                        {
                                addr=(i*10)+j;
                                buff_addr=addr>>4;

                                if (buff_addr!=lastAddr3 && buff_addr!=lastAddr2 &&
buff_addr!=lastAddr1)
                                {
                                        nodeLastComm[i][j]=0;

                                        return addr;
                                }
                        }
                }
        }
        return lastAddr4<<4;
}


/*****************************************************************************
  Function:
    void SortPackets(void)

  Description:
        This function takes the sending_data_buffer and sorts it so that there is at least
        a three transmission gap between consecutive packets going to the same address bus
        in the test node network

  Precondition:
    Data needs to be present in the sending_data_buffer to be sorted

  Parameters:
    Inputs:     None
        Outputs: None

  Returns:
    None

  Remarks:
    None
  ***************************************************************************/
void SortPackets(void)
{
        int i, j, k;
        int temp_buffer[30];
        short int exit=0;
        int shift_addr1, shift_addr2;

        fprintf(sorted_packets, "write pointer:%d read pointer:%d \n",
        write_pointer_sending, read_pointer_sending);

        for (i=read_pointer_sending;i<write_pointer_sending;i++)
        {
                shift_addr1=sending_data_buffer[i][0]>>4;
```

```
        //If the sort has not been run once yet, the address comparison values
        //(lasAddr1-lastAddr4) do not yet have valid addresses in them and thus
        //must be created
        if (comparisonFull==0)
        {
                //creating lastAddr4
                if (i==0)
                {
                        //setting lastAddr4
                        lastAddr4=shift_addr1;

                        fprintf(sorted_packets," i=0 addr1: %d\t", shift_addr1);

                        //update the nodeLastComm matrix to reflect the address of
                        //the outgoing packet
nodeLastComm[sending_data_buffer[i][0]/16][sending_data_buffer[i][0]%16]=0;

                        for (k=0; k<=29; k++)
                        fprintf(sorted_packets,"%d ",sending_data_buffer[i][k]);
                        fprintf(sorted_packets,"\n");
                }
                //compare to lastAddr4 and create lastAddr3
                else if (i==1)
                {
                        //if the current address does not match the last addresses
                        if (shift_addr1!=lastAddr4)
                        {
                                lastAddr3=shift_addr1;
                                fprintf(sorted_packets,"got to i==1 addr1: %d\t",
                                shift_addr1);
                        }
                        //This 'else' handles moving a packet if it matched
                        //lastAddr4
                        else
                        {
                                for (k=0;k<=29;k++)
                                        temp_buffer[k]=sending_data_buffer[i][k];

                                j=i;
                                //search through all packets after current packets
                                //and find one to trade places
                                while (exit==0)
                                {
                                        j++;
                                        shift_addr2=sending_data_buffer[j][0]>>4;

                                        if (shift_addr2!=lastAddr4)
                                        {
                                                //if no packet exists with an address
                                                //that doesn't match the last address
                                                //then create a filler packet and
                                                //insert it
                                                if (j>=write_pointer_sending)
                                                {
                                                        for (k=0;k<=29;k++)
                                                        {
                                                        sending_data_buffer[i][k]=0;
                                                        //move conflicting packet to
                                                        //end of queue
                sending_data_buffer[write_pointer_sending][k]=temp_buffer[k];
                                                        }
sending_data_buffer[i][0]=nodelastcommto();
                                                        sending_data_buffer[i][1]=18;
                                                        sending_data_buffer[i][3]=27;


shift_addr2=sending_data_buffer[i][0]>>4;
```

```
                                                write_pointer_sending++;
                                        }
                                        else
                                        {
                                                //trade packets with one that
                                                //doesn't match lastaddr4
                                                for (k=0;k<=29;k++)
                                                {
        sending_data_buffer[i][k]=sending_data_buffer[j][k];

        sending_data_buffer[j][k]=temp_buffer[k];
                                                }
                                        }
                                        //update lastAddr3 to reflect the new
                                        //traded packet
                                        lastAddr3=shift_addr2;
                                        exit=1;
                                }
                        }
                        exit=0;
                fprintf(sorted_packets,"i=1 addr2: %d\t", shift_addr2);
                }

                //update the nodeLastComm matrix to reflect transmitted
                //packet

        nodeLastComm[sending_data_buffer[i][0]/16][sending_data_buffer[i][0]%16]=0;

                for (k=0; k<=29; k++)
                fprintf(sorted_packets,"%d ",sending_data_buffer[i][k]);
                fprintf(sorted_packets,"\n");
        }
        //compare to lastAddr4 and lastAddr3 and create lastAddr2
        else if (i==2)
        {
                //no conflict with lastaddr3 or 4
                if (shift_addr1!= lastAddr3 &&  shift_addr1!=lastAddr4)
                {
                        lastAddr2=shift_addr1;
                fprintf(sorted_packets,"i=2 addr1: %d\t", shift_addr1);
                }
                //conflict with either lastaddr3 or 4
                else
                {
                        for (k=0;k<=29;k++)
temp_buffer[k]=sending_data_buffer[i][k];

                        j=i;
                        //search through all packets after current packets
                        //and find one to trade places
                        while (exit==0)
                        {
                                j++;
                                shift_addr2=sending_data_buffer[j][0]>>4;

                                if (shift_addr2!=lastAddr3 &&
shift_addr2!=lastAddr4)
                                {
                                        //if no packet exists with an address
                                        //that doesn't match the last address
                                        //then create a filler packet and
                                        //insert it
                                        if (j>=write_pointer_sending)
                                        {
                                                for (k=0;k<=29;k++)
                                                {
                                                sending_data_buffer[i][k]=0;
                                                //move conflicting packet to
                                                //end of queue
```

```
        sending_data_buffer[write_pointer_sending][k]=temp_buffer[k];
                                                        }

        sending_data_buffer[i][0]=nodelastcommto();
                                                        sending_data_buffer[i][1]=18;
                                                        sending_data_buffer[i][3]=27;


        shift_addr2=sending_data_buffer[i][0]>>4;

                                                        write_pointer_sending++;
                                                }
                                                else
                                                {
                                                //trade packets with one that
                                                //doesn't match lastaddr3 or 4
                                                        for (k=0;k<=29;k++)
                                                        {

        sending_data_buffer[i][k]=sending_data_buffer[j][k];

        sending_data_buffer[j][k]=temp_buffer[k];
                                                        }
                                                }
                                                //update lastAddr2 to reflect the new
                                                //traded packet
                                                lastAddr2=shift_addr2;
                                                exit=1;
                                        }

                                }
                                exit=0;
                        fprintf(sorted_packets,"i=2 addr2: %d\t", shift_addr2);
                        }

                        //update the nodeLastComm matrix to reflect transmitted
                        //packet

        nodeLastComm[sending_data_buffer[i][0]/16][sending_data_buffer[i][0]%16]=0;

                        for (k=0; k<=29; k++)
                        fprintf(sorted_packets,"%d ",sending_data_buffer[i][k]);
                        fprintf(sorted_packets,"\n");
                }
                //compare to lastAddr4, lastAddr3, and lastAddr2 and create
                //lastAddr1
                else if (i==3)
                {
                        //no conflict with lastaddr2, 3 or 4
                        if (shift_addr1!=lastAddr2 && shift_addr1!= lastAddr3 &&
shift_addr1!=lastAddr4)
                        {
                                lastAddr1=shift_addr1;
                                fprintf(sorted_packets,"got to i==2 addr1: %d\t",
                                shift_addr1);
                        }
                        //conflict with one of lastaddr2, 3 or 4
                        else
                        {
                                for (k=0;k<=29;k++)
temp_buffer[k]=sending_data_buffer[i][k];

                                j=i;
                                //search through all packets after current packets
                                //and find one to trade places
                                while (exit==0)
                                {
                                        j++;
                                        shift_addr2=sending_data_buffer[j][0]>>4;
```

```
                                                    if (shift_addr2!=lastAddr2 &&
shift_addr2!=lastAddr3 && shift_addr2!=lastAddr4)
                                                    {
                                                            //if no packet exists with an address
                                                            //that doesn't match the last address
                                                            //then create a filler packet and
                                                            //insert it
                                                            if (j>=write_pointer_sending)
                                                            {
                                                                    for (k=0;k<=29;k++)
                                                                    {

        sending_data_buffer[i][k]=0;
                                                                    //move conflicting
                                                                    //packet to end of queue

        sending_data_buffer[write_pointer_sending][k]=temp_buffer[k];
                                                                    }

        sending_data_buffer[i][0]=nodelastcommto();
                                                                    sending_data_buffer[i][1]=18;
                                                                    sending_data_buffer[i][3]=27;


        shift_addr2=sending_data_buffer[i][0]>>4;

                                                                    write_pointer_sending++;
                                                            }
                                                            else
                                                            {
                                                            //trade packets with one that
                                                            //doesn't match lastaddr2, 3 or 4
                                                                    for (k=0;k<=29;k++)
                                                                    {

        sending_data_buffer[i][k]=sending_data_buffer[j][k];

        sending_data_buffer[j][k]=temp_buffer[k];
                                                                    }
                                                            }
                                                            //update lastAddr1 to reflect the new
                                                            //traded packet
                                                            lastAddr1=shift_addr2;
                                                            exit=1;
                                                    }
                                            }
                                            exit=0;
                                            fprintf(sorted_packets,"got to i==3 addr2: %d\t",
                                            shift_addr2);
                                    }
                                    //update the nodeLastComm matrix to reflect transmitted
                                    //packet

        nodeLastComm[sending_data_buffer[i][0]/16][sending_data_buffer[i][0]%16]=0;

                                    for (k=0; k<=29; k++)
                                    fprintf(sorted_packets,"%d ", sending_data_buffer[i][k]);
                                    fprintf(sorted_packets,"\n");

                                    //all last address values have been set at least once
                                    comparisonFull=1;
                            }
                    }
                    //all lastAddr values have been created, and thus a full comparison and
                    //sort can take place from this point on
                    else
                    {
                            //no conflict with any lastaddr value
                            if (shift_addr1!= lastAddr3 &&  shift_addr1!=lastAddr2 &&
shift_addr1 != lastAddr1)
                            {
```

```
                    lastAddr4=lastAddr3;
                    lastAddr3=lastAddr2;
                    lastAddr2=lastAddr1;
                    lastAddr1=shift_addr1;
                    fprintf(sorted_packets,"got to i==%d addr1: %d\t", i,
shift_addr1);
               }
               //current address matched at least one of the past 3 bus addresses
               else
               {
               for (k=0;k<=29;k++) temp_buffer[k]=sending_data_buffer[i][k];

                    j=i;
                    //search through all packets after current packets and find
                    //one to trade places
                    while (exit==0)
                    {
                         j++;
                         shift_addr2=sending_data_buffer[j][0]>>4;

                         if (shift_addr2!=lastAddr3 && shift_addr2!=lastAddr2
&& shift_addr2 != lastAddr1)
                         {
                              //if no packet exists with an address that
                              //doesn't match the last address
                              //then create a filler packet and insert it
                              if (j>=write_pointer_sending)
                              {
                                   for (k=0;k<=29;k++)
                                   {
                                        sending_data_buffer[i][k]=0;
                                        //move conflicting packet to
                                        //end of queue

     sending_data_buffer[write_pointer_sending][k]=temp_buffer[k];
                                   }

     sending_data_buffer[i][0]=nodelastcommto();
                                   sending_data_buffer[i][1]=18;
                                   sending_data_buffer[i][3]=27;


     shift_addr2=sending_data_buffer[i][0]>>4;

                                   write_pointer_sending++;
                              }
                              else
                              {
                                   //trade packets with one that doesn't
                                   //match any lastaddr value
                                   for (k=0;k<=29;k++)
                                   {

     sending_data_buffer[i][k]=sending_data_buffer[j][k];

     sending_data_buffer[j][k]=temp_buffer[k];
                                   }
                              }
                              //update last address values
                              lastAddr4=lastAddr3;
                              lastAddr3=lastAddr2;
                              lastAddr2=lastAddr1;
                              lastAddr1=shift_addr2;
                              exit=1;
                         }
                    }
                    exit=0;
                    fprintf(sorted_packets," i=%d addr2: %d\t", i,
                    shift_addr2);
               }
```

```
                    //update the nodeLastComm matrix to reflect transmitted packet
        nodeLastComm[sending_data_buffer[i][0]/16][sending_data_buffer[i][0]%16]=0;

                        for (k=0; k<=29; k++)
                                fprintf(sorted_packets,"%d ", sending_data_buffer[i][k]);
                        fprintf(sorted_packets,"\n");
                }

        }
}


/*****************************************************************************************
  Function:
    void NoResponseCheck(void)

  Description:
        This function checks all packets to see if a reply from the node was ever
received.
        The function will re-send a packet three times before marking that a node is not
        connected if no response is ever received.

  Precondition:
    None

  Parameters:
    Inputs:     None
        Outputs: None

  Returns:
    None

  Remarks:
    None
  ***************************************************************************************/
void NoResponseCheck(void)
{
        int i, j;

        //don't bother starting to check until at least 15 packets have been sent
        if (read_pointer_sending>16)
        {
                //check all packets that have been sent more than 15 transmissions ago
                for (i=0; i<=read_pointer_sending-16; i++)
                {
                        //compare current read_pointer to when the data was sent
                        if (time(NULL)-sending_data_buffer_time[i]>=16)
                        {
                                //retry sending packet 3 times
                                if (sending_data_buffer[i][30]<=3)
                                {
                                        for (j=0; j<=29; j++)
sending_data_buffer[write_pointer_sending][j]=sending_data_buffer[i][j];
                                        sending_data_buffer[i][30]+=1;
                                        write_pointer_sending++;
                                        lastSort++;
                                }
                                //all 3 retries failed, mark the packet as not being
connected
                                else if (sending_data_buffer[i][30]=4)
        nodesConnected[sending_data_buffer[i][0]/16][sending_data_buffer[i][0]%16]=0;
                        }
                }
        }

        return;
}
```

```
/*******************************************************************************
  Function:
    void CheckReceived(void)

  Description:
       This function checks recently received packets against sent packets to determine
       which packet received a response.  The function then marks that that sent packet
       received a reply.  This function also verifies the ACK field of the returned
packet.

  Precondition:
    None

  Parameters:
    Inputs:     None
       Outputs: None

  Returns:
    None

  Remarks:
    None
  *******************************************************************************/
void CheckReceived(void)
{
       int i, j;

       //scan through all sent packets for a match to the most recently received
       for (i=0; i<=read_pointer_sending; i++)
       {
              if (receiving_data_buffer[write_pointer_receiving-
1][0]==sending_data_buffer[i][0])
              {
                     if (receiving_data_buffer[write_pointer_receiving-
1][1]==sending_data_buffer[i][2])
                     {
                            if (receiving_data_buffer[write_pointer_receiving-
1][3]==sending_data_buffer[i][3])
                            {
                                   //mark packet as having received a reply
                                   sending_data_buffer[i][30]=8;

                                   if (receiving_data_buffer[write_pointer_receiving-
1][2]==0 || receiving_data_buffer[write_pointer_receiving-1][2]==15 ||
receiving_data_buffer[write_pointer_receiving-1][2]==170)
                                   {
                                          for (j=0; j<=29; j++)
sending_data_buffer[write_pointer_sending][j]=sending_data_buffer[i][j];
                                          write_pointer_sending++;
                                          lastSort++;
                                   }
                            }
                     }
              }
       }
}


/*******************************************************************************
  Function:
    void DieWithError(char *errorMessage)

  Description:
       Function used to print any error messages which may arise when creating socket

  Precondition:
    None

  Parameters:
    Inputs:     System created error messages
       Outputs: None
```

```
  Returns:
    None

  Remarks:
    This code was originally part of the example code from the book TCP/IP Sockets in C
        written by Donahoo, M and Calvert, K.
  ********************************************************************************/
void DieWithError(char *errorMessage)
{
        #ifdef WIN32
                fprintf(stderr,"%s: %d\n", errorMessage, WSAGetLastError());
        #else
                perror(errorMessage);
        #endif
          exit(1);
}
```

Vita Auctoris

Christopher Rennick was born in Regina, Saskatchewan, Canada. He grew up in Stratford, Ontario, Canada where he graduated from Stratford Central Secondary School in 2003. He received his Bachelor of Applied Science in Electrical Engineering from the University of Windsor in Ontario, Canada in 2007. He is currently a candidate for the Master's of Applied Science degree in Electrical Engineering at the University of Windsor and hopes to be completed his degree at the end of calendar year 2009. His research interests include custom hardware and embedded system design.