

DATA TRANSFER SYSTEM
FOR HOST COMPUTER
AND FPGA COMMUNICATION

Thesis

Submitted to

The School of Engineering of the

UNIVERSITY OF DAYTON

In Partial Fulfillment of the Requirements for

The Degree of

Master of Science in Electrical Engineering

By

Michael T. Barnard

UNIVERSITY OF DAYTON

Dayton, Ohio

December, 2015

DATA TRANSFER SYSTEM FOR HOST COMPUTER AND FPGA
COMMUNICATION

Name: Barnard, Michael T.

APPROVED BY:

Eric Balster, Ph.D.
Advisor Committee Chairman
Associate Professor, Department of
Electrical and Computer Engineering

John G. Weber, Ph.D.
Committee Member
Professor, Department of Electrical and
Computer Engineering

Tarek M. Taha, Ph.D.
Committee Member
Associate Professor, Department of
Electrical and Computer Engineering

John G. Weber, Ph.D.
Associate Dean
School of Engineering

Eddy M. Rojas, Ph.D., M.A., P.E.
Dean, School of Engineering

© Copyright by
Michael T. Barnard
All rights reserved
2015

ABSTRACT

DATA TRANSFER SYSTEM FOR HOST COMPUTER AND FPGA COMMUNICATION

Name: Barnard, Michael T.
University of Dayton

Advisor: Dr. Eric Balster

This Thesis describes a communication system to allow for the transmission of data between a host computer and a DE2-115 FPGA board over an Ethernet connection. This is achieved by using a socket between the host computer and a NIOS II embedded processor that accepts the data from the host computer and transfers it to the FPGA fabric. The host computer uses a C++ program to open a file and send the data over the socket to the NIOS II processor. The NIOS II acts as memory controller for the Synchronous Dynamic Random Access Memory (SDRAM) on the board with separate input and output data sections for the Hardware Description Language (HDL) processing module. A HDL module then processes the data and sends it back to the NIOS II to be returned to the host computer over the socket. The data transfer system is tested with three basic image processing functions performed on three sample images to verify its functionality. This data transfer system allows for easier testing of digital designs on the DE2-115 board by providing test data to the digital design in an efficient manner.

To Myself...

...for all of your hard work.

ACKNOWLEDGMENTS

I would like to thank everyone in my life during the completion of both the design and documentation; you all helped in your own way. I would like to especially thank the following people for their contributions.

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

I would like to thank you for all of your advice and guidance throughout my time at UD both undergrad and graduate. I can confidently say that you are the reason that I both attempted and completed a Masters Degree.

- **Dr. John Weber, PhD and Dr. Tarek Taha, PhD:**

Thank you to Dr. Weber and Dr. Taha for your advice and input on the writing of my thesis and for being on my thesis committee. I know you have busy schedules, so thank you for allowing me to take up some of your valuable time.

- **David and Judy Barnard:**

I would like to thank you both for allowing me to pursue a Masters degree without balancing a budget on top of course work. Thank you again for all of the love and support you have given me all of my life.

TABLE OF CONTENTS

ABSTRACT	iii
DEDICATION	iv
ACKNOWLEDGMENTS	v
LIST OF FIGURES	viii
LIST OF TABLES	x
I. INTRODUCTION	1
II. FPGA COST	3
III. BACKGROUND TECHNOLOGIES	6
3.1 Sockets and Socket Communication	6
3.2 NIOS II Embedded Processor	7
3.2.1 NIOS II/f Features	8
3.2.2 NIOS II/s Features	8
3.2.3 NIOS II/e Features	8
IV. SYSTEM DESIGN AND IMPLEMENTATION	10
4.1 Socket Client Program	11
4.2 Communications Management Module	13
4.3 Hardware Description Language Modules	15
V. RESULTS	20
5.1 Image Inverter	21
5.2 Image Flipper and Inverter	23

5.3	Moving Average Filter	24
5.4	Logic Utilization	26
VI.	CONCLUSION AND FUTURE WORK	34
	BIBLIOGRAPHY	36

LIST OF FIGURES

2.1	DE2-115 Board	4
4.1	Transfer System Data Flow	11
4.2	Example of Console Output With Multiple Files Processed	12
4.3	Map of Hardware and Important Connections	16
4.4	Logic Diagram of Positive Edge Detector	17
4.5	Positive Edge Detector Signals	18
4.6	The Processing Module State Machine	19
5.1	512x512 Grayscale Input Image	21
5.2	1024x1024 Grayscale Input Image	22
5.3	560x420 Color Input Image	23
5.4	Console Output of Image Inverter With All Three Input Files	24
5.5	512x512 Grayscale Inverted Output Image	25
5.6	1024x1024 Grayscale Inverted Output Image	26
5.7	560x420 Color Inverted Output Image	27
5.8	Console Output of Image Inverter and Flipper With All Three Input Files	28
5.9	512x512 Grayscale Inverted and Flipped Output Image	28

5.10	1024x1024 Grayscale Inverted and Flipped Output Image	29
5.11	560x420 Color Inverted and Flipped Output Image	29
5.12	Console Output of Moving Average Filter With All Three Input Files	30
5.13	512x512 Grayscale Moving Average Output Image	30
5.14	1024x1024 Grayscale Moving Average Output Image	31
5.15	560x420 Color Moving Average Output Image	31
5.16	Compilation Report for the Communication System	32
5.17	Compilation Report for the Image Inverter	32
5.18	Compilation Report for the Image Inverter and Flipper	33
5.19	Compilation Report for the Moving Average Filter	33

LIST OF TABLES

2.1	FPGA Boards Owned and Cost	5
-----	--------------------------------------	---

CHAPTER I

INTRODUCTION

The flexibility of Field Programmable Gate Arrays (FPGAs) has allowed them to become prevalent in many fields including digital design and control systems. These devices can be programmed with a wide range of digital logic designs leading to large amounts of possible research topics and design projects utilizing FPGAs [1]. An FPGA is a chip that contains hundreds of thousands of digital logic elements that can be connected in numerous ways to create any digital logic structure. FPGAs can be programmed as many times as needed, so improvements can be made to a design and the design can be updated without replacing any hardware. Designs may be prototyped on one chip and then programmed on to other FPGAs once the design is ready for production; one chip can also be used to prototype many designs one after the other. The main restriction to widespread FPGA prototyping is cost: the cost of the board hosting the FPGA, the cost of the software licenses required to program the board, and the cost of software development kits (SDKs) used for data transfer to and from the board. To be able to transfer data to the board most of the more expensive FPGA models have a dedicated SDK that takes care of connecting with the board and transferring the data while the less expensive models do not. These SDKs allow designers to focus on their specific engineering problem without having to worry about the complexities of data transfer. Less expensive FPGA models typically lack SDKs making them less capable and flexible in solving engineering

problems. This Thesis describes a communication system which allows designers to communicate data to the DE2-115 FPGA development board with ease.

Chapter II discusses the impact that this data transfer design will have on the campus research community. Next, Chapter III is a discussion on the backgrounds of a few technologies significant to the design in addition to FPGAs and HDLs. How the design is created and implemented is explained in Chapter IV; covering all three sections of the design: the host PC program, the NIOS II program, and the HDL modules. Chapter V follows with a section on the results of three test functions on three test images and the data transfer system's overall performance during these tests. Chapter VII concludes by summarizing this paper and discussing future work based on this research.

CHAPTER II

FPGA COST

Widespread FPGA prototyping would be more prevalent if the costs could be reduced: cost of the board hosting the FPGA, cost of software licenses required to program the board, and cost of software development kits (SDKs) for data transfer to and from the board. Most of the more expensive FPGA models have a dedicated SDK to transfer data to the FPGA board while less expensive models do not. These SDKs allow designers to focus on their specific engineering problem without having to worry about the complexities of data transfer. The SDKs for data transfer can cost upwards of \$4,000 per year alone, and Altera's Quartus II software can also cost \$2,000 per year [2]. That \$6,000 a year allows for the use of the FPGA but does not include the cost of the FPGA itself. The FPGAs with this type of support software can cost up to \$50,000 per board. Alternatively, there are inexpensive boards, like the DE2-115 shown in Figure 2.1, that can be programmed with license free software but do not have any SDKs to transfer data [3]. These inexpensive boards are limited to switches and buttons for data input which are not realistic input methods for large scale data transfer. The development of this data transfer system allows for communications between the inexpensive board and a host PC. The development of such a system allows for the possible \$56,000 investment to be replaced with a \$300 investment for prototyping purposes.

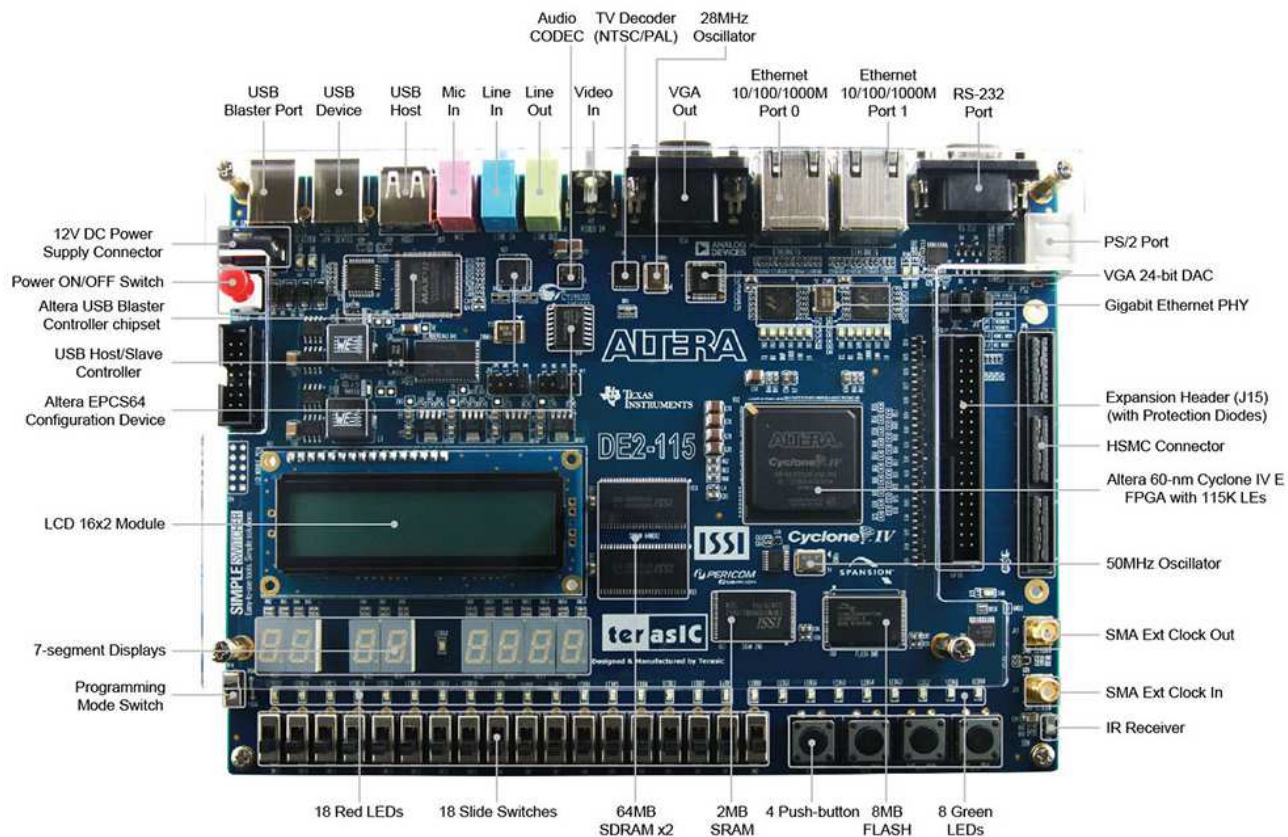


Figure 2.1: DE2-115 Board

FPGA Board NAME	Number Owned	Unit Cost	Total Cost
Gidel Proce V [5]	57	\$12,450	\$709,650
Gidel PROCStar III 110 [6]	3	\$7,500	\$22,500
Gidel PROCStar III 150 [6]	32	\$14,150	\$452,800
Total	92		\$1,184,950

Table 2.1: FPGA Boards Owned and Cost

This data transfer design will serve as the basis for other research projects. It also allows for more projects to occur simultaneously because for the cost of one high end FPGA board approximately 180 inexpensive boards can be purchased. The DE2-115 board is slightly less powerful than more expensive models, but the scope of research may not be affected by that difference. By using the less expensive boards in research, the projects would be completed less expensively. The projects that would benefit from this data transfer system are not limited to research. There are numerous college-level courses in Electrical and Computer Engineering that use FPGAs and could incorporate assignments based off of this research to better illustrate how certain digital designs operate on larger sets of input data [4].

There are already some high end FPGA boards owned by the University of Dayton used by various faculty in the Electrical and Computer Engineering Department and researchers at the University of Dayton Research Institute. Table 2.1 shows the number of each type of FPGA board owned by the University and the total cost of all of the boards. Some of the research performed on these boards could be moved to the DE2-115 board so the University would not have to invest in so many expensive FPGAs. The money saved could be used to facilitate other research topics by hiring more researchers or purchasing specific equipment required by specialized research topics.

CHAPTER III

BACKGROUND TECHNOLOGIES

The data transfer system relies on two other major technologies in addition to the hardware description language (HDL) language and FPGA chips. These technologies are socket communication and the NIOS II embedded-processor. Socket communication allows for data transfer between two separate applications and even two separate computing machines. The NIOS II architecture is an embedded processor that can be instantiated on an FPGA chip and execute the C programming language.

3.1 Sockets and Socket Communication

The term socket describes an endpoint for either interprocess or interapplication communication [7]. Two processes can communicate over a network by using two separate sockets. Every socket is identified by an IP address concatenated with a port number i.e. (146.86.5.20:1625) which describes the IP address, 146.86.5.20, and the port number, 1625. Specific services listen to well known ports; ports numbered below 1024 are considered well known but not all are used. Sockets are generally used in client-server architecture to ensure that communication will occur easily and error free. To be able to communicate over sockets all connections must be unique.

The communication capabilities of sockets allow applications to send and receive packets of information over the connection when it is established. Applications can also use sockets to connect

a local socket to a remote address of a socket created by a different application. Sockets listen for any remote connections that connect to a local socket to allow for communication between the remote application and a local application.

There are five software functions used in the operations of socket for both servers and clients: socket, bind, listen, accept, and connect. The socket function creates a communication endpoint for either the server or client. To associate an endpoint a specific point the server uses the bind function. The listen function makes the socket a passive listener and the accept function is used to accept a connection request from the client. The connect function is used by the client to request a connection from the server.

3.2 NIOS II Embedded Processor

The NIOS II is an embedded processor architecture designed to be implemented using Altera FPGAs [8]. An embedded processor, also called a soft microprocessor, softcore microprocessor, or a soft processor, is a microprocessor core that can be wholly implemented using logic synthesis. The processor can be programmed onto any device containing programmable logic; the processor can also be instantiated multiple times on the same device to mimic a multi-core processor.

The NIOS II processor is a 32 bit RISC architecture with little endianness. There are three versions of the NIOS II processor: the NIOS II/f, the NIOS II/s, and the NIOS II/e. The NIOS II/f core is designed for maximum performance at the expense of core size. The NIOS II/s core is designed for a balance between performance and cost. The NIOS II/e core uses the smallest possible logical footprint so that smaller FPGAs are able to use the NIOS II technology.

3.2.1 NIOS II/f Features

The features of the NIOS II/f include separate instruction and data caches that range in size from 512 bytes to 64 kilobytes [9]. An optional Memory Management Unit (MMU) or Memory Protection Unit (MPU) is also available in this version. The processor has access to up to 2 GB of external address space. The option for tightly coupled memory for instructions and data is also available. The processor has a six stage pipeline to achieve maximum Dhrystone Million Instructions per Second per MegaHertz. Some peripherals that are included are single cycle hardware multiply, a barrel shifter, optional hardware divide option, and dynamic branch prediction. The core also supports up to 256 custom instructions and unlimited hardware accelerators. A Joint Test Action Group (JTAG) debug module is also present along with optional JTAG debug module enhancements, including hardware breakpoints, data triggers, and a real time trace.

3.2.2 NIOS II/s Features

The features of the NIOS II/s include an instruction cache and up to 2 GB of external address space [9]. The option for tightly coupled memory for instructions is present. The five stage pipelined processor also has static branch prediction, hardware multiply, divide and shift options. Up to 256 custom instructions can be supported by the processor. The JTAG debug module is also present with the optional JTAG module enhancements, including hardware breakpoints, data triggers, and real time trace.

3.2.3 NIOS II/e Features

The features of the NIOS II/e core include up to 2 GB of external address space [9]. The complete processor system only occupies a maximum of 700 logic elements. The JTAG debug module is also present with some optional debug enhancements. The core supports up to 256 custom

instructions. This version of the NIOS II processor does not require a license unlike the other two versions of the NIOS II processor.

CHAPTER IV

SYSTEM DESIGN AND IMPLEMENTATION

The proposed data transfer system is split into three major components: the Hardware Description Language (HDL) modules to process the data, the Communications Management Module (CMM), which runs on the NIOS II processor, that moves the data from the socket to the processing module and back again, and the Socket Client used by the host computer to communicate with the NIOS II processor [10]. The processing module uses the CMM as a memory controller for the Synchronous Dynamic Random Access Memory (SDRAM) reading and writing the data that it processes. The CMM reads that data from the socket and unpacks it byte by byte and sends the individual bytes down to the processing module. When the CMM processor receives data from the processing module it puts the data into an output buffer that is sent across the socket when all of the data has been processed. The host computer's Socket Client opens the file to be processed and sends it across the socket as one buffer; when the processed data is received from the socket it is written out to a newly created file. Figure 4.1 shows how the data moves around the various components. The host computer program is written in C++, the CMM is written in C, running on a NIOS II processor and the processing module is written in Verilog, a hardware description language. The NIOS II/s variant is used because of its balance between computing power and logic size.

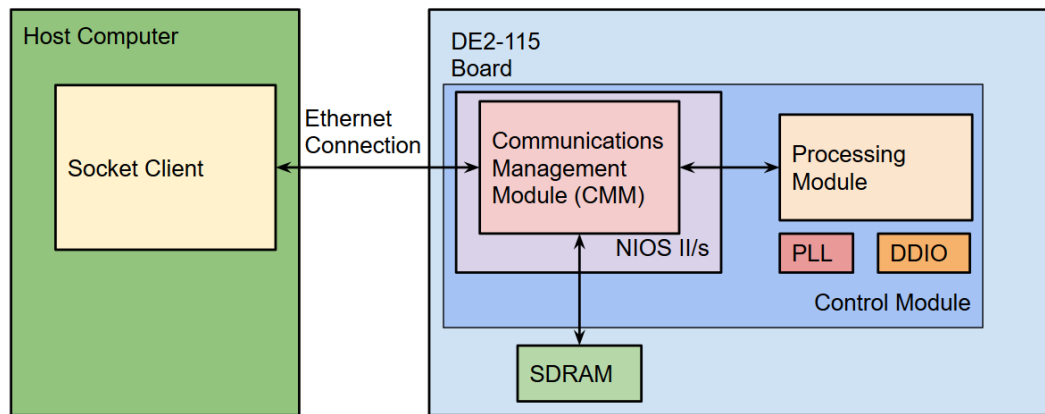


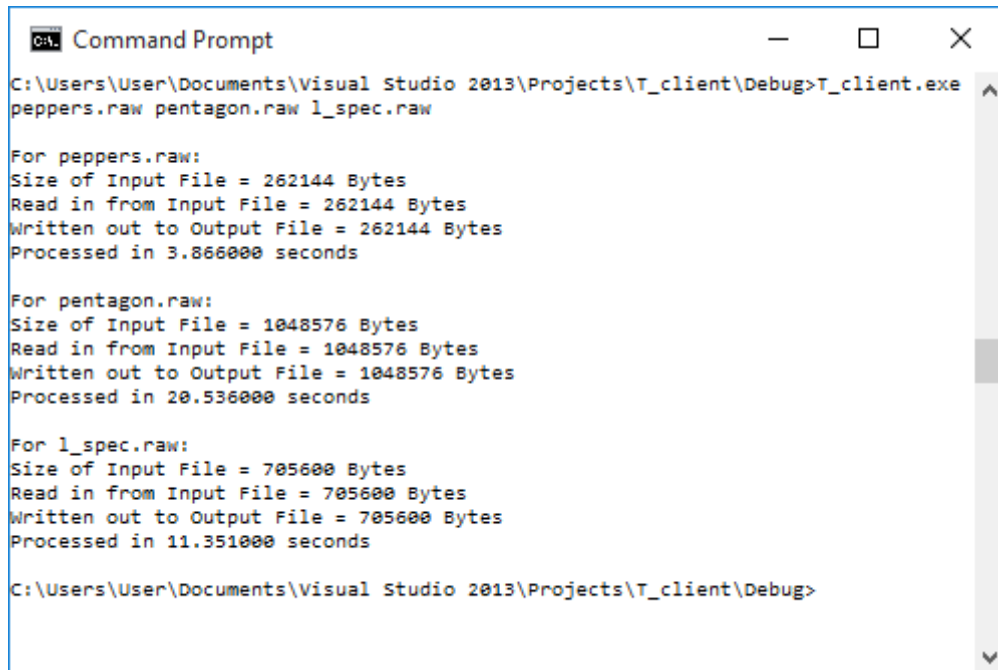
Figure 4.1: Transfer System Data Flow

4.1 Socket Client Program

The Socket Client is designed to open the input and output files and communicate to the DE2-115 board over the Ethernet connection. The socket server is created by the CMM on the board and connected to the host computer when a file is ready to be processed. The input data is written to the socket and the processed data is read from the socket upon completion. The processed data is written to a new file with a name that can be edited before run time but is defaulted to "processed_" followed by the name of the input file. The time taken to process the data is also calculated and output for the user's convenience. The size of the file, number bytes read from the input file, and the number of bytes written to the output file are also printed to the console when the file has been processed to ensure that the entire file is processed and there are no errors in transmitting the data across the socket.

The Socket Client has the ability to process more than one file each time it is called by including multiple file names in the command line arguments. The only command line arguments needed are the application name and the name of a file to be processed. The IP address and port number are written in the Socket Client program because the location of the server will not change unless it

is changed by the user; these two values are used to connect to the socket and communicate with the board. The file name is used to open both the input and output files so that if multiple files are processed the output files are easily paired with their respective input file. When multiple files are processed, a separate socket connection is made for each file and the console output is repeated for each file as they are processed. An example of the console output with three files being processed can be seen in Figure 4.2.



```
C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>T_client.exe
peppers.raw pentagon.raw l_spec.raw

For peppers.raw:
Size of Input File = 262144 Bytes
Read in from Input File = 262144 Bytes
Written out to Output File = 262144 Bytes
Processed in 3.866000 seconds

For pentagon.raw:
Size of Input File = 1048576 Bytes
Read in from Input File = 1048576 Bytes
Written out to Output File = 1048576 Bytes
Processed in 20.536000 seconds

For l_spec.raw:
Size of Input File = 705600 Bytes
Read in from Input File = 705600 Bytes
Written out to Output File = 705600 Bytes
Processed in 11.351000 seconds

C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>
```

Figure 4.2: Example of Console Output With Multiple Files Processed

The Socket Client starts by checking for correct usage to ensure that all of the needed information is present. The number of file names given is then obtained by subtracting one from the number of command line arguments present to loop the correct number of times. After opening the input file the size of the file is determined by using the C functions `fseek()` and `ftell()` to find the value of the pointer to the last byte in the file. The buffers for the input, output, and data to be sent across

the socket are allocated in memory by using malloc; the size is determined by the size of the input file. The buffers are dynamically allocated to eliminate the need for the user to hard code the size of the file into the program and to allow multiple large files to be processed without taking up huge chunks of the Socket Clients' available memory. The socket connection is established using an IP address and port number declared in the Socket Client.

The data from the input file is not the only data written to the socket. The size of the file occupies the first four bytes of the buffer that is written down to the board so that the other parts of the design are aware of how much data is present to process. The rest of this buffer is filled with the data from the input file and written to the socket. The program then waits for the socket to be written to by the NIOS II processor and reads in the processed data from the socket until the number of bytes read is the same as the size of the input file. The output buffer is written to the output file and the socket, the input file, and the output file are closed and the three buffers are deallocated to conserve memory. If there are more files to be processed then the program determines the new file's size and reopens the socket as many times as needed.

4.2 Communications Management Module

The CMM is written in C and run on the NIOS II processor that is instantiated on the FPGA chip. The bulk of the code is dedicated to creating and monitoring the socket and providing error messages to the user in case any errors arise. When a connection is made and data is available to be read the socket is read and the size of the file is separated from the body of data. When all of the data has been read from the socket the input data is placed into a buffer that is sent to a function to communicate with the processing module along with the size of the file and the socket details to allow for communication.

When a connection to the socket is detected the program calls one of two functions: if this is the first connection then `SSS_handle_accept` is called or if the program has accepted the connection then `SSS_handle_receive` is called to receive the data. When the first section of data is read from the socket the number of bytes of data is separated from the rest of the data from the socket so that the program knows when to stop reading the socket. If all of the data is received in the first read then the function to move the data to the processing module is called; if there is still more data to be read then the program reads the socket until all the data has been received. When all of the data is received then a payload buffer is allocated and filled with the input data and sent with the file size and socket details to the function that communicates with the processing module.

`SSS_exec_command` is the function that acts as a memory controller for the processing module and also sends the processed data back over the socket. The first task it performs is to allocate memory for the output buffer based on the size passed from `SSS_handle_receive`. The HDL modules are then enabled by setting their reset high; the size of the file is written to a Parallel Input Output (PIO) port to be used by the processing module to know how much data needs to be processed. The start signal is toggled on and off to start the processing module moving through its state machine to process the data. The code then enters a loop that will exit when it receives a signal from the processing module that all of the data has been processed. This loop constitutes the memory interface to the SDRAM reading from a specific address when a signal is received and writing to a specific address when a signal is received. Within the loop there are two if statements: one for reading data from the data buffer and one for writing to the output buffer. When the signal to read data is received the index of the data to be read is read from a PIO port; if the index references a location outside the buffer sends back a zero instead of garbage data. The selected value is sent back to the module and a signal that the data is ready is toggled to allow the module to continue to process the data. The process for writing data to the output buffer is nearly the same; the address is obtained from a

PIO port and the data is written into that address of the buffer. A signal to continue processing is also used to allow the module to continue through its state machine. When all of the data has been processed the output buffer is sent back to the host computer over the socket. Finally the reset for the processing module is set low to ensure that they start from their initial states the next time they are used.

4.3 Hardware Description Language Modules

There are three main HDL modules that are instantiated on the board: the control module that instantiates the other modules, the NIOS II processor module, and a processing module as seen in Figure 4.1. These modules are responsible for most of the space on the FPGA and most of the work in processing the data. There are two more specialized modules that are present: a Phase Locked Loop (PLL) and a Double Data Rate Input/Output (DDIO) core. The PLL is used to create the three clocks used with the three possible Ethernet speeds: 10 Megabit, 100 Megabit, and 1 Gigabit. The DDIO module transmits data on both edges of the reference clock and creates an accurate edge-aligned clock-data relationship with the Ethernet port. The HDL modules and the major connections between them are shown in Figure 4.3; the arrows on the left side of the figure are connected to the pins of the FPGA chip. The PLL and DDIO modules are megafunctions provided by Altera while the NIOS II module is created using the Qsys tool. The control and process modules are hand written. The control module instantiates the other four module and also includes some digital logic to set various parameters for the use of the Ethernet port.

The NIOS II system is created by using the Qsys system integration tool and is instantiated in the control module. The processor module is made up of various components that are compiled together by the Qsys tool. The processor module includes a clock source, JTAG UART for communicating with the computer that programs it, and a system ID peripheral to simplify programing the

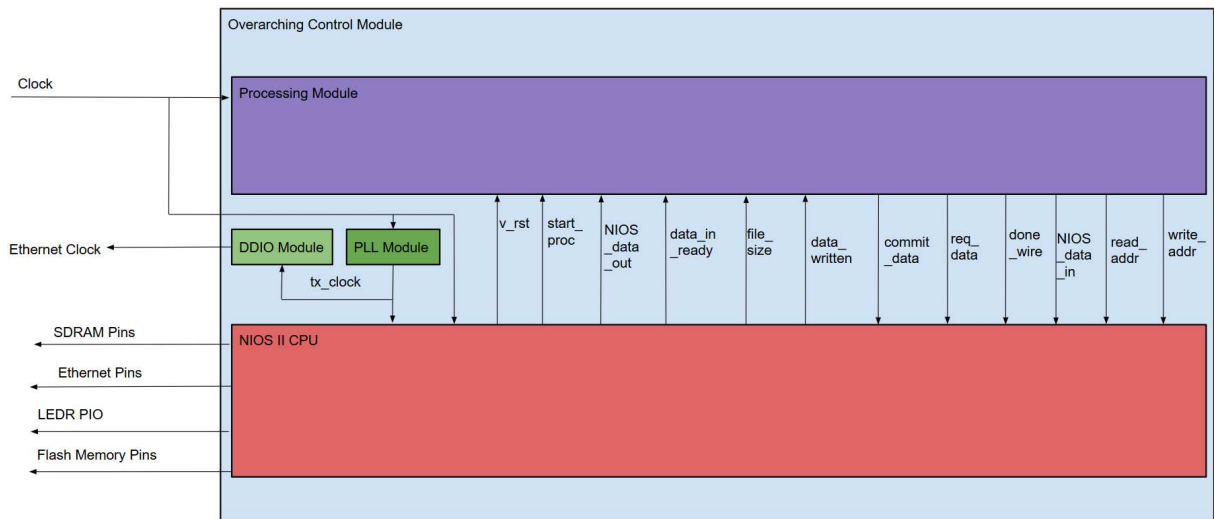


Figure 4.3: Map of Hardware and Important Connections

processor. There is also a NIOS II core that is the basis of the processor; the type used is the NIOS/s to maximize performance and leave as many logic elements open for the digital design. A memory controller for the SDRAM and a clock signals core are included; the NIOS II processor uses the SDRAM for both instruction and data storage, and the clock signal's core ensures that any communication between the memory and the processor is synchronized. There are two timers present in the processor system: a system timer to schedule tasks and a timestamp timer that can be used to measure the time that any number of lines of code take to complete. There are multiple components that are used to control the Ethernet port on the board; the main one being a direct controller for the Triple-Speed Ethernet. Two Scatter-Gather Direct Memory Access (SGDMA) controllers are used for the transmit and receive functions of the Triple-Speed Ethernet controller; a small memory used for the descriptors of the SGDMAs is also included. There is a core used to control the Flash memory on the board that is used to store details about the Ethernet port in the C code. There are also many PIOs used to allow communication between the CMM and the processing module including the input and output data and addresses and the control signals.

The processing module can be configured to perform any number of operations on the input data but there are still a few features that are consistent across all designs to allow for the correct movement of data. There are two positive edge detectors for the start and data ready signals from the CMM because the CMM is running on the NIOS II processor and is on a different clock domain. An edge detector is made up of two flip flops and a two input AND gate; the input signal is assigned to the input of the first flip flop and the output of the first flip flop is assigned to the second flip flop [11]. The inverted output of the second flip flop output and the output of the first flip flop are ANDed together as seen in Figure 4.4. When the input signal goes high and is stored in the first flip flop the output of the AND gate is also high because the inverted output of the second flip flop was high but goes low the next positive edge of the clock when the high value is read from the first flip flop causing the AND gate to go low; Figure 4.5 shows an example of the signals listed in Figure 4.4. The edge detectors ensure that the input signals from the CMM are high for only one clock cycle so that they are not read multiple times. The data and the addresses to read from and write to are stored in registers to easily allow computations to be performed on them. The data is also placed into a register so that it can be easily manipulated as needed. All three of the registers are continuously assigned to outputs so that they can be read by other modules without any additional work.

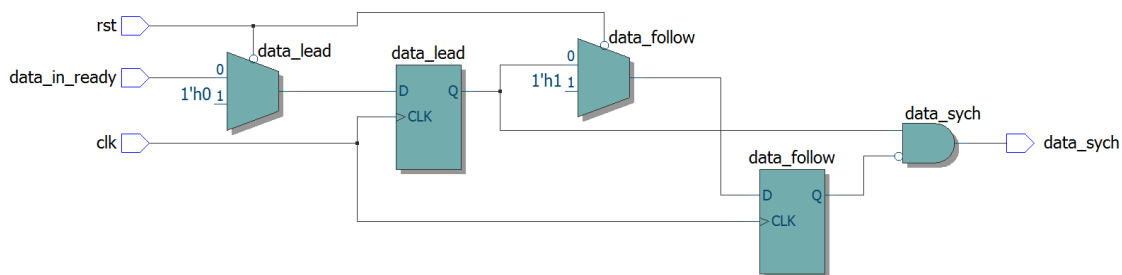


Figure 4.4: Logic Diagram of Positive Edge Detector

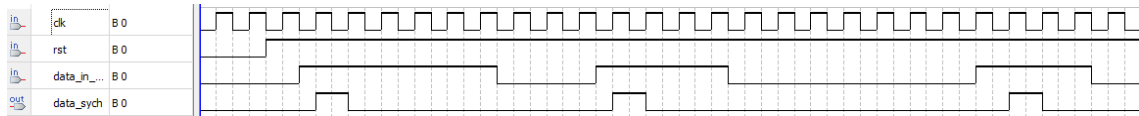


Figure 4.5: Positive Edge Detector Signals

The state machine used to process the data is shown in Figure 4.6; each bubble is a state and the arrows indicate the next state visited. Arrows with text near them are conditional transitions and the other arrows are unconditional transitions. The state machine starts in the initial state and moves to the “wait_for_data” state when the positive edge detector detects an edge in the start signal; when the CMM sends a signal that the data is valid the state machine moves into the “read_data” state. Once the data has been read the state machine then moves to “update_read_addr” that the input data will be read from and moves into the process state. The process state is where the data is processed; it can be one state long or it can contain multiple states depending on the function being performed by the design. After the data has been processed the state machine moves to the “write_data” state that writes the data to the CMM and stays in that state until a signal from the CMM tells it that the data has been written. The state machine then updates the write address in the “update_write_addr” state and moves back to the “wait_for_data” state that waits for the next piece of data from the processor. The state machine moves to the “fin” state from the “wait_for_data” state when the write address is equal to file size plus one meaning that all of the data has been written back to the CMM.

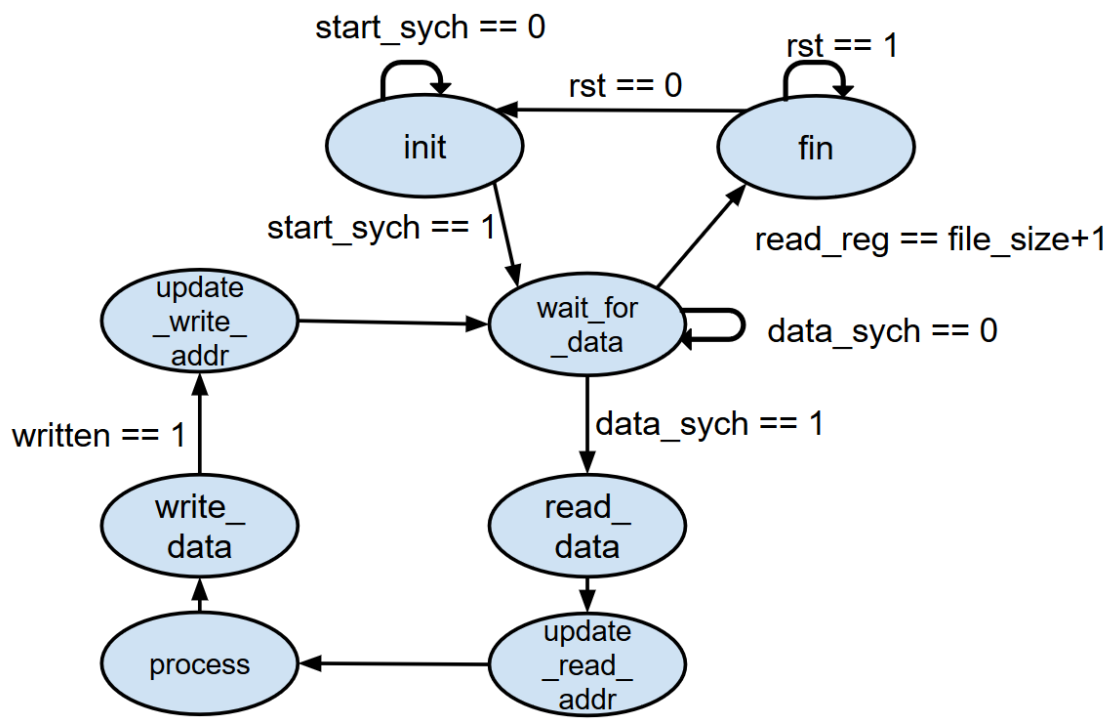


Figure 4.6: The Processing Module State Machine

CHAPTER V

RESULTS

The data transfer system operates as intended while also occupying a small amount of the logic elements on the FPGA chip and possessing a fairly large amount of memory to store data+. The space for the socket server and associated C code is approximately 2 MB and when it is subtracted from the 16 MB total the maximum amount of remaining memory is 14 MB. This means that the maximum file size is 4.6 MB due to the split of available memory into the receive buffer, the input data buffer, and the output data buffer. The data transfer system HDL only takes up around 13,000 logic elements out of the total 114,480 meaning that there is 89% of the chip that can be dedicated to the user's design allowing for fairly complex designs to be tested on the DE2-115 board.

To test the proposed data transfer system, three simple image processing test designs are used to ensure that the processing section can be changed without affecting the memory manager and socket communication sections. These three designs are a negative of the image, a negative image flipped across both the vertical and horizontal axes, and a moving average filter that blurs the image. Three test input images are used for each design; the three input images are shown in Figures 5.1, 5.2, and 5.3. The size of the images as a raw file type are 258 KB, 1026 KB, and 690 KB respectively. The raw file type is used so that no part of the design needs to worry about any header information located in the file itself. The raw file type only contains the pixel values so reading and writing files of the type can be easily done with a single C array. The basic test input is Figure 5.1 because it

is relatively small and grayscale so it does not tax any part of the system and is easily processed. Figure 5.2 is used as a test image because of its size; its large size exposes any faults in the system's handling of data. If the system mishandles 1 MB of data then it may mishandle larger files that it is presented with. The final test image, Figure 5.3, is used because it is a color image; this is just to make sure that the system can process color images without corrupting the data.



Figure 5.1: 512x512 Grayscale Input Image

5.1 Image Inverter

The first test design inverts an image and is given by

$$g[x, y] = 255 - f[x, y] \quad (5.1)$$

where x and y describe pixel locations, f is the input image, and g is the output image. The same equation is used for the color image but instead of a single pixel's intensity as the input and



Figure 5.2: 1024x1024 Grayscale Input Image

output the amount of red, green, or blue in the pixel is used. This leads to an inversion of the color of the pixel and not necessarily an inversion in the pixel's total intensity.

When the data transfer system is run with all three input images given at the same time the console output is shown in Figure 5.4. The output of the design for the first input image is shown in Figure 5.5; the data is processed in 3.87 seconds. Figures 5.6 and 5.7 are the other two outputs of the inverter that are processed in 20.54 seconds and 11.35 seconds respectively. All three images are processed correctly the two grayscale images have their values reversed so that dark areas became light and light areas became dark. Figure 5.7 is not as easily identified in the areas that are not black or white. The areas that are blue in Figure 5.3 are red in Figure 5.7 and vice versa meaning that the image is processed correctly.

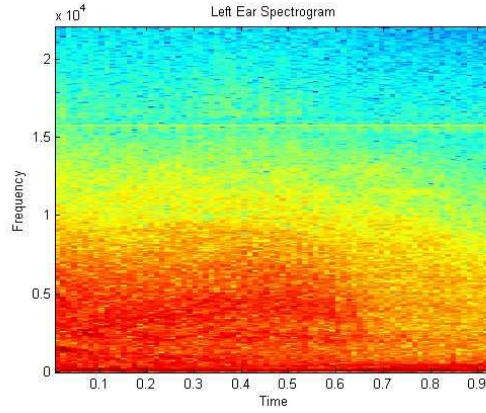


Figure 5.3: 560x420 Color Input Image

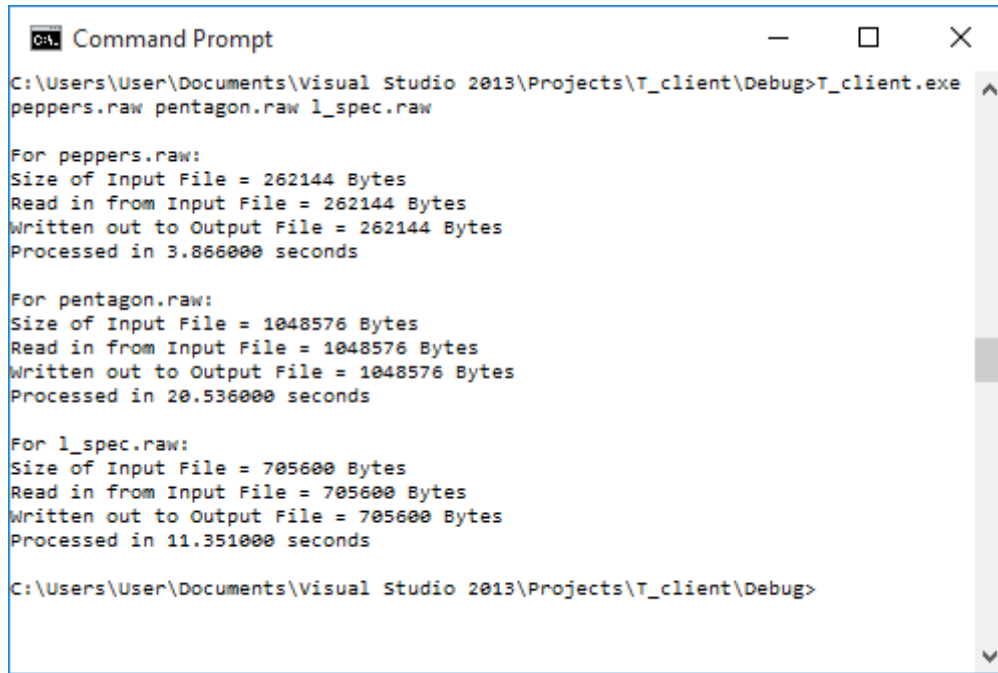
5.2 Image Flipper and Inverter

This second test design processes the data the same as the previous test design however the data is read normally but it is written starting at the end of the buffer and ending at the beginning. By writing to the output buffer in this way the top left pixel ends up switched with the bottom right pixel and the top right pixel is switched with the bottom left pixel. The process used in this test is

$$g[x, y] = 255 - f[C - x, R - y]. \quad (5.2)$$

where C is equal to the number of columns in the image and R is equal to the number of rows in the image. The equation is similar to Equation (5.1) except that the input data is accessed in a different order. This results in the output image being flipped upside down and backwards.

The console output for the second test design with all three input files can be seen in Figure 5.8. Processing Figure 5.1 with this design yields Figure 5.9 in 2.98 seconds. Figures 5.10 and 5.11 are created by this algorithm from Figures 5.2 and 5.3 in 20.51 seconds and 11.24 seconds respectively. The grayscale images are inverted normally and flipped as expected but because of the fact that



```
Command Prompt
C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>T_client.exe
peppers.raw pentagon.raw l_spec.raw

For peppers.raw:
Size of Input File = 262144 Bytes
Read in from Input File = 262144 Bytes
Written out to Output File = 262144 Bytes
Processed in 3.866000 seconds

For pentagon.raw:
Size of Input File = 1048576 Bytes
Read in from Input File = 1048576 Bytes
Written out to Output File = 1048576 Bytes
Processed in 20.536000 seconds

For l_spec.raw:
Size of Input File = 705600 Bytes
Read in from Input File = 705600 Bytes
Written out to Output File = 705600 Bytes
Processed in 11.351000 seconds

C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>
```

Figure 5.4: Console Output of Image Inverter With All Three Input Files

three bytes of data make up one pixel the colored output of this test is not just a flipped version of Figure 5.7. The black and white portions of Figure 5.11 are the same; the change comes in the center colored portion. Because the colored data is read in a red-green-blue pattern and written in a blue-green-red pattern the processed colors are not the same as the previous test design.

5.3 Moving Average Filter

The final test design is a moving average filter that calculates the output value by averaging a pixel intensity with a certain number of pixel intensities to each side. This design averages the sample with the six values on each side of it to obtain the output value. The equation for the filter is

$$g[x, y] = \frac{1}{2M+1} \sum_{i=-M}^M f[x+i, y] \quad (5.3)$$



Figure 5.5: 512x512 Grayscale Inverted Output Image

where $M = 6$ is the number of pixels on each side that are included in the average, g is the output image, f is the input image, x and y are the pixel's location in the image and i is used to reference the other pixels being averaged. The value of i ranges from $-M$ to M because the pixels in the average are on both sides of the pixel's location. When pixel referenced by i is outside of current row of pixels a value from another row is used; for the pixels to the left it is the previous row and for pixels to the right it is the next row.

The filter processes the three test images and produces the console output seen in Figure 5.12. The outputs for the two grayscale images are shown in Figures 5.13 and 5.14; on both the left and right sides there is some distortion because those pixel values were calculated by using values from the ends of other rows. The blurring effect is not as prevalent in Figure 5.15 because a red, green, or blue value is averaged with values for the other colors instead of the overall pixel intensity; those mixed color averages lead to more detail and the colors seen in the center of the figure. The time to process the grayscale images is 2.98 seconds for Figure 5.13 and 20.51 seconds for Figure 5.14. Figure 5.15 is created by the system in 11.35 seconds.



Figure 5.6: 1024x1024 Grayscale Inverted Output Image

5.4 Logic Utilization

A secondary goal of this Thesis is for the entire Data Transfer System to occupy as few logic elements on the FPGA chip as possible. Figure 5.16 is an example of the compilation report that Quartus II [12] provides at the end of a successful compilation. Some information list includes the number of logic elements used by the design, the number of pins on the FPGA used by the design, the amount of on chip memory reserved for the design, and the number of total PLLs used in the design. The compilation report shows the report for the communications section of the system but not the processing section; the 12,842 logic elements present must be present for the system to operate properly. The other 89% of the chip can be used to create the digital design to be tested. The number of logic elements taken up by the image inverter and the image inverter and flipper do not vary by many logic elements as shown in Figures 5.17 and 5.18. These two designs do not add many logic elements past the basic communication system. Figure 5.19 shows the report

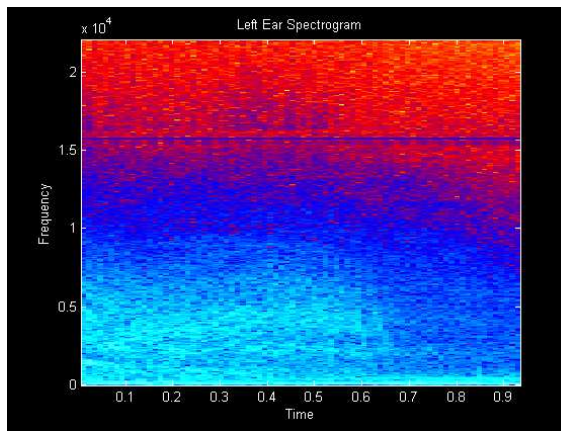


Figure 5.7: 560x420 Color Inverted Output Image

for the most computational intensive design, the moving average filter. Only a few hundred logic elements are added to achieve the design giving a frame of reference for the relationship between the number of logic elements used and the complexity of the digital design being instantiated. With over 100,000 logic elements left on the FPGA chip with the moving average filter present the scope of the design that would occupy the vast majority of the FPGA can be seen as immense.

```
Command Prompt
C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>T_client.exe
peppers.raw pentagon.raw l_spec.raw

For peppers.raw:
Size of Input File = 262144 Bytes
Read in from Input File = 262144 Bytes
Written out to Output File = 262144 Bytes
Processed in 2.977000 seconds

For pentagon.raw:
Size of Input File = 1048576 Bytes
Read in from Input File = 1048576 Bytes
Written out to Output File = 1048576 Bytes
Processed in 20.512000 seconds

For l_spec.raw:
Size of Input File = 705600 Bytes
Read in from Input File = 705600 Bytes
Written out to Output File = 705600 Bytes
Processed in 11.352000 seconds

C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>
```

Figure 5.8: Console Output of Image Inverter and Flipper With All Three Input Files

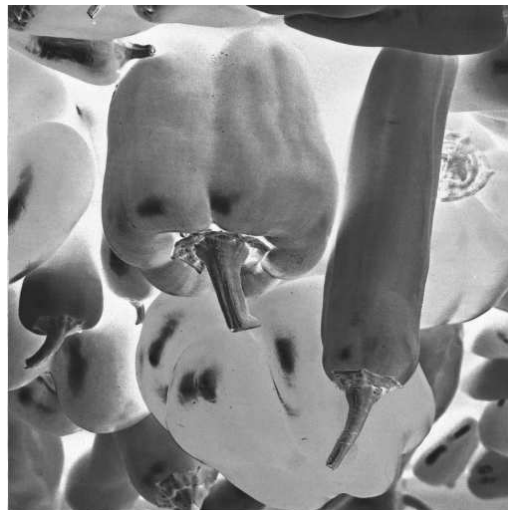


Figure 5.9: 512x512 Grayscale Inverted and Flipped Output Image



Figure 5.10: 1024x1024 Grayscale Inverted and Flipped Output Image

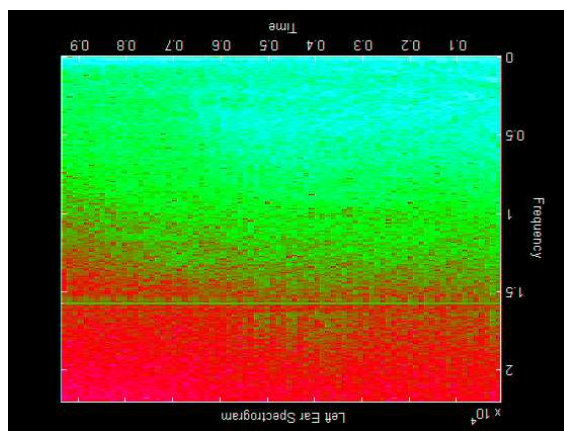


Figure 5.11: 560x420 Color Inverted and Flipped Output Image


```
Command Prompt

Written out to Output File = 705600 Bytes
Processed in 11.355000 seconds

C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>T_client.exe
peppers.raw pentagon.raw l_spec.raw

For peppers.raw:
Size of Input File = 262144 Bytes
Read in from Input File = 262144 Bytes
Written out to Output File = 262144 Bytes
Processed in 2.981000 seconds

For pentagon.raw:
Size of Input File = 1048576 Bytes
Read in from Input File = 1048576 Bytes
Written out to Output File = 1048576 Bytes
Processed in 20.514000 seconds

For l_spec.raw:
Size of Input File = 705600 Bytes
Read in from Input File = 705600 Bytes
Written out to Output File = 705600 Bytes
Processed in 11.353000 seconds

C:\Users\User\Documents\Visual Studio 2013\Projects\T_client\Debug>
```

Figure 5.12: Console Output of Moving Average Filter With All Three Input Files



Figure 5.13: 512x512 Grayscale Moving Average Output Image



Figure 5.14: 1024x1024 Grayscale Moving Average Output Image

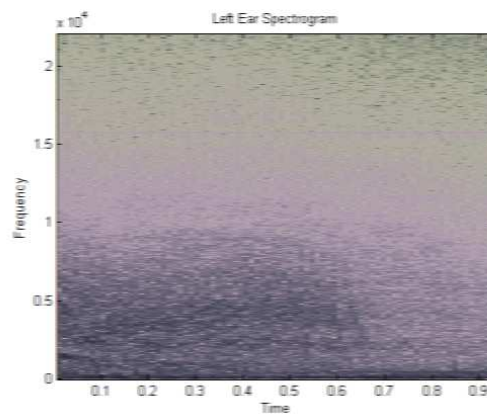


Figure 5.15: 560x420 Color Moving Average Output Image

Flow Status	Successful - Tue Oct 13 11:24:04 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	Pro_2
Top-level Entity Name	Pro_2
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	12,842 / 114,480 (11 %)
Total combinational functions	9,323 / 114,480 (8 %)
Dedicated logic registers	9,441 / 114,480 (8 %)
Total registers	9571
Total pins	130 / 529 (25 %)
Total virtual pins	0
Total memory bits	2,627,364 / 3,981,312 (66 %)
Embedded Multiplier 9-bit elements	4 / 532 (< 1 %)
Total PLLs	2 / 4 (50 %)

Figure 5.16: Compilation Report for the Communication System

Flow Status	Successful - Tue Oct 13 11:07:06 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	Pro_2
Top-level Entity Name	Pro_2
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	13,174 / 114,480 (12 %)
Total combinational functions	9,582 / 114,480 (8 %)
Dedicated logic registers	9,675 / 114,480 (8 %)
Total registers	9805
Total pins	130 / 529 (25 %)
Total virtual pins	0
Total memory bits	2,627,364 / 3,981,312 (66 %)
Embedded Multiplier 9-bit elements	4 / 532 (< 1 %)
Total PLLs	2 / 4 (50 %)

Figure 5.17: Compilation Report for the Image Inverter

Flow Status	Successful - Tue Oct 13 10:33:44 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	Pro_2
Top-level Entity Name	Pro_2
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	13,165 / 114,480 (11 %)
Total combinational functions	9,582 / 114,480 (8 %)
Dedicated logic registers	9,675 / 114,480 (8 %)
Total registers	9805
Total pins	130 / 529 (25 %)
Total virtual pins	0
Total memory bits	2,627,364 / 3,981,312 (66 %)
Embedded Multiplier 9-bit elements	4 / 532 (< 1 %)
Total PLLs	2 / 4 (50 %)

Figure 5.18: Compilation Report for the Image Inverter and Flipper

Flow Status	Successful - Tue Oct 13 11:18:48 2015
Quartus II 64-Bit Version	15.0.0 Build 145 04/22/2015 SJ Web Edition
Revision Name	Pro_2
Top-level Entity Name	Pro_2
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	13,498 / 114,480 (12 %)
Total combinational functions	9,934 / 114,480 (9 %)
Dedicated logic registers	9,784 / 114,480 (9 %)
Total registers	9914
Total pins	130 / 529 (25 %)
Total virtual pins	0
Total memory bits	2,627,364 / 3,981,312 (66 %)
Embedded Multiplier 9-bit elements	4 / 532 (< 1 %)
Total PLLs	2 / 4 (50 %)

Figure 5.19: Compilation Report for the Moving Average Filter

CHAPTER VI

CONCLUSION AND FUTURE WORK

This Thesis presents a system to transfer data between a host computer and a DE2-115 FPGA board. This is accomplished by connecting to a socket over an Ethernet connection and using a NIOS II processor on the board to handle moving the data from the socket to the processing module. The processing module uses the NIOS II processor as a memory controller to process the data; while the processor acts as a socket server for the host computer's socket client. The data is processed and saved to a new file in an efficient manner such that one Mega Byte of data can be processed in twenty seconds with one command line input. The data transfer system uses a relatively small portion of the logic elements on the FPGA chip, approximately 11%, leaving the majority of the chip free for the user's HDL design. The system has a total of 16 MB of the SDRAM available to it so the maximum file size is approximately 4.6 MB because the memory is sectioned into received, input, and output buffers.

The future work of related to this system is varied it ranges from improving the design to be more efficient in chip and memory usage to performing research of digital designs such as creating better methods to segment an image. The improvements to this system aim to reduce the size of its footprint so that more complex designs can be tested on the DE2-115 board and to increase the amount of memory available to the user allowing for larger amounts of test data to be used. Any

research that can be conducted on the DE2-115 board and needs input data to be properly tested can incorporate this system to supply the design with data.

BIBLIOGRAPHY

- [1] R. Hartenstein, *Designing Embedded Processors*. Springer Netherlands, 2007, ch. Basics of Reconfigurable Computing, pp. 451–501.
- [2] Altera. (2015) Buy Design Software. [Online]. Available: <http://www.altera.com/buy/software/buy-software.html>
- [3] Terasic. (2013) Terasic - DE Main Boards - Cyclone - Altera DE2-115 Development and Education Board. [Online]. Available: <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=165&No=502&PartNo=3>
- [4] U. of Dayton. (2015) Electrical and Computer Engineering < Undergraduate. [Online]. Available: <http://catalog.udayton.edu/allcourses/ece/>
- [5] Gidel. ProceV: Stratix IV based PCIe evaluation & system board, DSP Imaging Vision Rapid Prototyping. [Online]. Available: <http://www.gidel.com/ProceV.html>
- [6] ——. ProcStar III: - Stratix III based PCIe evaluation & system board, DSP Imaging Vision Rapid Prototyping. [Online]. Available: <http://gidel.com/PROCStar%20III.htm>
- [7] M. Brain and R. D. Reeves, *Win 32 System Services: The Heart of Windows 98 and Windows 2000*, 3rd ed. Prentice Hall, 2000.
- [8] Altera. (2015) Nios II Processor - Overview. [Online]. Available: <http://www.altera.com/devices/processor/nios2/ni2-index.html>
- [9] ——. Benefits Nios II Processor Cores. [Online]. Available: <https://www.altera.com/products/processors/benefits/nios-ii-processor-cores.html>
- [10] M. M. Mano and M. D. Ciletti, *Digital Design*, 4th ed. Pearson Prentice Hall, 2007.
- [11] FPGACenter. Edge Detector - FPGACenter.com. [Online]. Available: http://fpgacenter.com/examples/basic/edge_detector.php
- [12] Altera. (2015) Quartus II - Overview. [Online]. Available: <https://www.altera.com/products/design-software/fpga-design/quartus-ii/overview.html>