

2011

Design and Development of an FPGA-based Distributed Computing Processing Platform

Juliana Su
Bucknell University

Follow this and additional works at: https://digitalcommons.bucknell.edu/masters_theses

Recommended Citation

Su, Juliana, "Design and Development of an FPGA-based Distributed Computing Processing Platform" (2011). *Master's Theses*. 38.
https://digitalcommons.bucknell.edu/masters_theses/38

This Masters Thesis is brought to you for free and open access by the Student Theses at Bucknell Digital Commons. It has been accepted for inclusion in Master's Theses by an authorized administrator of Bucknell Digital Commons. For more information, please contact dcadmin@bucknell.edu.

I, Juliana Su, do grant permission for my thesis to be copied.

DESIGN AND DEVELOPMENT OF AN FPGA-BASED DISTRIBUTED COMPUTING
PROCESSING PLATFORM

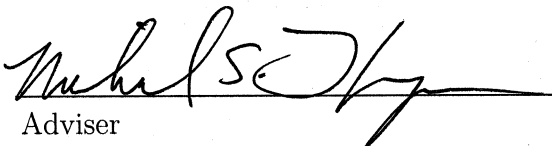
by

Juliana Su

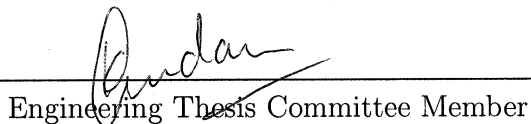
A Thesis

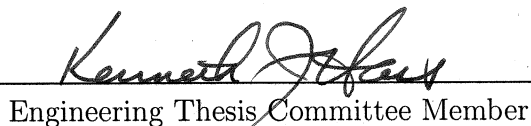
Presented to the Faculty of
Bucknell University
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Approved:


Adviser


Department Chairperson


Engineering Thesis Committee Member


Engineering Thesis Committee Member

5/5/2011
Date

Acknowledgements

To my family, Mom, Dad, and Tina, thank you for all of your support and encouragement over the years. You have always believed in me, no matter what, and for that, I am so grateful.

To my advisor, Professor Thompson, thank you for this wonderful opportunity. I still remember our very first meeting, something required as part of the first ELEC 101 homework assignment. You asked me about what I wanted to do after graduation. As a junior, with graduation being some two years away, I told you I was not sure (maybe grad school?). Well, here we are, almost four years later, with me just over a week away from graduating with a master's and a few months away from starting a PhD program; I think your question has been answered. I cannot thank you enough for your guidance, advice, support, and patience over the past few years.

To the members of my thesis committee, Professors Hass and Nepal, thank you for your thoughtful comments, suggestions, and advice on this work. I appreciate your help, not just with my research, but with my other academic endeavors, as well. Thanks for teaching me the ins and outs of digital design and writing recommendations for me.

Contents

1	Introduction	1
1.1	FPGAs	2
1.2	Distributed Computing	3
1.3	Problem Overview	4
1.4	Contributions	4
1.5	Thesis Organization	5
2	Problem Statement	6
2.1	Challenges of Designing with FPGAs	6
2.1.1	Development Time	6
2.1.2	Hardware/Software Partitioning	7
2.1.3	FPGA Fabric	8
2.2	Design Requirements	8
2.3	Summary	9
3	Related Work	10
3.1	Reconfigurable Computing Systems	10
3.1.1	Splash/Splash 2	11
3.1.2	PRISM	11
3.1.3	SLAAC	12
3.1.3.1	Tower of Power	12

3.1.3.2	Adaptive Computing Systems (ACS) Application Programming Interface (API)	13
3.1.4	Baylor University Cluster	14
3.1.5	The Reconfigurable Computing Cluster (RCC) Project	15
3.2	Types of Applications	16
3.2.1	Digital Signal Processing	17
3.2.2	Bioinformatics	18
3.2.3	Cryptography	18
3.3	Summary	19
4	Design and Implementation	20
4.1	System Overview	20
4.2	Software Framework	22
4.2.1	Conventions	23
4.2.2	Initialization	24
4.2.2.1	Communication	24
4.2.2.2	Data Queues	24
4.2.3	Processing Data	25
4.2.3.1	Reading and Formatting Input Data	25
4.2.3.2	Receiving Results	26
4.3	Hardware Core Manager Framework	27
4.3.1	Conventions	28
4.3.2	Operating System	29
4.3.3	Initialization	29
4.3.4	Receiving Requests	29
4.3.5	Processing Core Requests	30
4.3.5.1	Input Data Queue Setup	31
4.3.5.2	Core List Setup	31
4.3.6	Processing Data Requests	32

4.4	Communication	32
4.4.1	Messages	32
4.4.1.1	Message Components	33
4.4.1.2	Types of Messages	35
4.4.1.3	Exchanging Messages	36
4.5	Summary	39
5	Testing and Results	40
5.1	Overview	40
5.2	Experimental Setup	41
5.2.1	System Components	41
5.2.1.1	Host PC	41
5.2.1.2	FPGA Boards	41
5.2.1.3	Hardware Base System	42
5.2.1.4	Network	44
5.2.1.5	Input Data Files	45
5.2.2	Development Tools	45
5.3	Test Application	46
5.3.1	3DES	46
5.3.2	3DES Hardware Core	47
5.3.2.1	Interfacing with the 3DES Core	47
5.3.2.2	Verification	49
5.4	Test Scenarios	50
5.4.1	Single Core, Single Board Configuration	51
5.4.2	Multiple Cores, Single Board Configuration	51
5.4.3	Single Core Per Board, Multiple Boards Configuration	53
5.4.4	Multiple Cores Per Board, Multiple Boards Configuration	55
5.5	Software Implementation	56
5.6	Preliminary Conclusions	57

5.7	Performance Bottleneck	58
5.7.1	Hardware Core	59
5.7.2	File I/O	60
5.7.3	Network Transmission	60
5.7.4	Analysis	62
5.8	Summary	63
6	Conclusion	64
6.1	Summary	64
6.2	Future Work	65
	Bibliography	69
A	Software Framework	70
A.1	wrapper.c	70
A.2	wrapper.h	86
B	Example Software Application	91
B.1	FPGA.c	91
B.2	Script to Run Software Application	97
B.3	Makefile for Software Application	97
C	Hardware Core Manager	99
C.1	main.c	99
C.2	HCM.c	102
C.3	wrapper.c	108
C.4	wrapper.h	119
C.5	memory_map.h	122
D	MATLAB Implementation	124
D.1	Script to Run 3DES	124

D.2 3DES MATLAB Code	125
D.3 DESDecrypt MATLAB Code	127

List of Tables

5.1	Single core, single board configuration throughput results.	51
5.2	Multiple cores, single board configuration throughput results.	52
5.3	Single core per board, multiple boards throughput results.	54
5.4	Multiple cores per board, multiple boards configuration throughput results .	56
5.5	3DES MATLAB performance results.	57
5.6	Hardware core performance comparison.	59
5.7	Processing times excluding time for file I/O	60

List of Figures

1.1	Block diagram of an FPGA architecture	3
2.1	Diagram of hardware/software partitioning	8
3.1	Diagram of the Tower of Power	13
3.2	Block diagram of the Baylor University cluster	15
3.3	Block diagram of the RCC cluster	16
4.1	Graphical representation of the system.	21
4.2	Data flow through the software application.	23
4.3	Data flow through the hardware core manager.	28
4.4	Flowchart depicting the receive request process.	30
4.5	Message components.	33
4.6	Message formats.	36
4.7	Exchanging of messages between the host PC and FPGA board over time.	37
5.1	System Assembly view screenshot.	42
5.2	Graph of number of cores versus throughput.	53
5.3	Graph of number of boards versus throughput (one core per board).	55

Abstract

This thesis presents two frameworks- a software framework and a hardware core manager framework- which, together, can be used to develop a processing platform using a distributed system of field-programmable gate array (FPGA) boards. The software framework provides users with the ability to easily develop applications that exploit the processing power of FPGAs while the hardware core manager framework gives users the ability to configure and interact with multiple FPGA boards and/or hardware cores. This thesis describes the design and development of these frameworks and analyzes the performance of a system that was constructed using the frameworks. The performance analysis included measuring the effect of incorporating additional hardware components into the system and comparing the system to a software-only implementation. This work draws conclusions based on the provided results of the performance analysis and offers suggestions for future work.

Chapter 1

Introduction

Recently, there has been growing interest in high-performance computing using FPGAs due to numerous application areas experiencing an increased demand in processing capability. Research conducted over the past twenty years has demonstrated that hardware acceleration using FPGAs yields considerable performance improvements for certain application areas, such as bioinformatics, digital signal processing, cryptography, and network packet processing.

One method of exploiting the processing power of FPGAs is to cluster them together and distribute computations among them. The concept of dividing up a problem into smaller tasks and distributing these tasks among separate processing elements is called distributed computing. By using FPGAs in a distributed computing environment, performance is increased through parallelization.

1.1 FPGAs

FPGAs are integrated circuits designed to be reconfigured and reprogrammed an unlimited amount of times after the manufacturing process. They serve as the building blocks of reconfigurable computing, a computing paradigm that focuses on dividing applications into parallel, application-specific pipelines. The beauty of reconfigurable computing is that it combines the speed of hardware with the flexibility of software, essentially the best characteristics of hardware and software.

The basic architecture of an FPGA, shown in Figure 1.1, consists of configurable logic blocks (CLBs), routing channels, and input/output blocks. Each CLB, which is embedded in a general routing structure, consists of look-up tables and flip-flops that can be configured to perform either combinational or sequential logic. CLBs are surrounded by input/output blocks (IOBs) for interfacing with external devices. The general routing structure allows for arbitrary wiring, so designers can connect the logic elements however necessary. Designs can be implemented on an FPGA using a hardware description language (HDL), such as Verilog or VHDL, or a schematic.

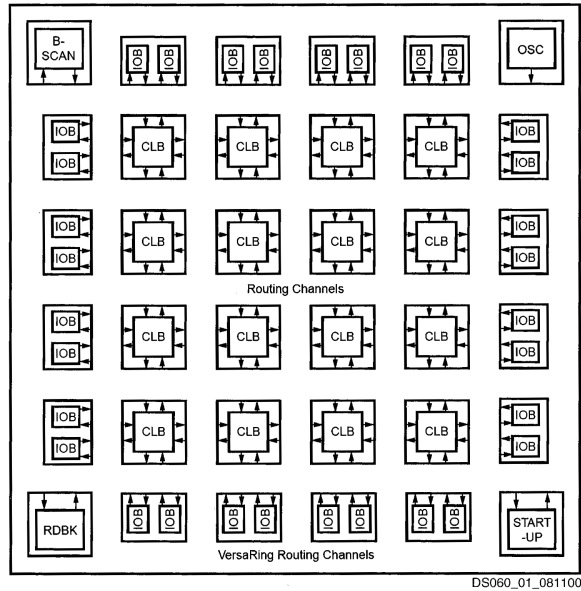


Figure 1.1: Block diagram of an FPGA architecture [1].

In general, the amount of exploitable parallelism is the key factor in determining the suitability of an application for FPGAs. FPGAs can only outperform modern processors by exploiting huge amounts of parallelism.

1.2 Distributed Computing

A distributed system is defined as a collection of individual processing elements that communicate through a network. The concept of using a distributed system to solve computational problems is known as distributed computing. In distributed computing, a problem is first broken up into smaller tasks. These sub-tasks are then distributed among the separate processing elements within the distributed system. The processing elements work together to solve a problem, even though each processing element may be responsible for a different portion of the problem.

1.3 Problem Overview

With FPGAs, the benefits of high performance and reconfigurability come at the cost of added complexity. When designing for FPGAs, one challenge that designers face is the need to work with both hardware and software components, each of which possesses its own design methods. Using FPGAs in a distributed computing environment adds another level of complexity since it introduces the element of networking. For the user, determining how to configure and interact with hardware, software, and networking components in order to develop an application for an FPGA can be a tedious and time-consuming task.

1.4 Contributions

The contributions of this work are as follows:

- We provide a software framework that provides users with a set of functionalities to develop a software application that interfaces with multiple FPGA boards.
- We provide a hardware core manager framework that gives users the ability to configure and interact with multiple FPGA boards and/or hardware cores.
- We perform an analysis of various system configurations, using an application developed with the frameworks, to observe what effect incorporating additional hardware components (FPGA boards and hardware cores) into a system has on performance.
- We compare the performance of the application developed with our frameworks (a hardware/software-based solution) to the performance of a software-based implementation of the application.

1.5 Thesis Organization

Chapter 2 defines the problem statement for this work. Chapter 3 explores related work. Chapter 4 describes the design and implementation of the system. Chapter 5 provides the results and analysis of our performance analysis. Chapter 6 summarizes this work and discusses potential future work.

Chapter 2

Problem Statement

Unlike general-purpose computing systems, which separate the design of hardware and software, embedded systems involve the simultaneous design of hardware and software. The challenge of creating a system which clusters FPGAs in a distributed computing environment requires designers to be knowledgeable in hardware, software, and networking concepts.

2.1 Challenges of Designing with FPGAs

Exploiting huge amounts of parallelism using FPGAs is no trivial matter. This can be attributed to three aspects of FPGA design: 1) lengthy development time; 2) complex hardware/software partitioning; and 3) limited size.

2.1.1 Development Time

One of the major difficulties of FPGA design lies in the manner that designers must approach a problem. Designing for FPGAs involves simultaneously using multiple resources that are

spread across a chip to achieve a massive amount of parallelism. Software programming, in contrast, is generally aimed at exploiting a microprocessor's ability to sequentially execute instructions. Humans naturally think in a sequential manner, so translating a design into parallel logic takes significantly more time than sequential programming. In addition to the increased time it takes to implement designs in an HDL, there is a steep learning curve associated with learning to use FPGA development tools. These tools are often vendor-specific, depending on the FPGA being used. In summary, the increased time it takes to design and implement parallel logic translates into longer development times.

2.1.2 Hardware/Software Partitioning

Finding the right balance between the flexibility of software and speed of hardware while satisfying design requirements, such as performance, area, designer effort, etc., is a challenge associated with designing for FPGAs. In a process called hardware/software partitioning, shown in Figure 2.1, it is the responsibility of the designer to decide how best to divide an application between a microprocessor component (“software”) and one or more custom coprocessors (“hardware”). The task of partitioning is particularly difficult due to the fact that there are often many ways to partition a design.

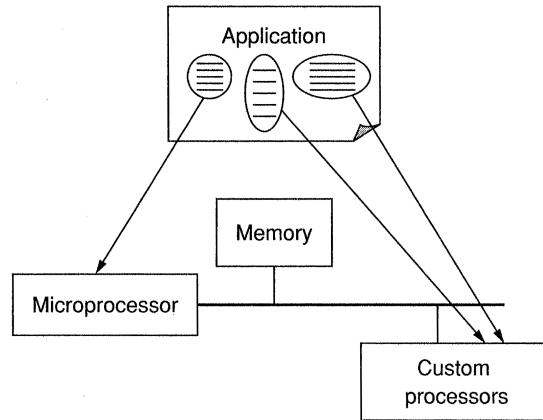


Figure 2.1: Diagram of hardware/software partitioning [2].

2.1.3 FPGA Fabric

Programmable interconnects dominate the FPGA fabric. A large amount of area on an FPGA, roughly 90 percent, is devoted to internal routing, leaving only 10 percent of the fabric for configurable logic [2]. Having a finite amount of area for configurable logic means that there is a limit on how large an FPGA design can be in order to fit on a single chip.

2.2 Design Requirements

As the related work described in Chapter 3 implies, the task of designing a processing platform that incorporates FPGAs is not a novel idea. What makes our particular problem unique from previous research, however, is the combination of design requirements imposed upon it. First, we require the system to be flexible. The system needs to not be application-specific since we want to have the ability to incorporate any type of hardware core into the system. Second, we require the system to be scalable. This means that the user should have the ability to incorporate multiple boards and multiple cores into the system. In our case, specifically, the ability to support multiple cores is crucial since we only have three FPGA

boards available for use. Our system will be very small-scale; however, we will design for potential larger-scale configurations.

2.3 Summary

This chapter defined the problem statement of this work. Summarizing our design goals, we aim to develop a system that decreases development time while being flexible and scalable. The next chapter provides an overview of related work.

Chapter 3

Related Work

The following review of related work covers reconfigurable computing systems, as well as types of FPGA applications. The purpose of reviewing reconfigurable computing systems is to provide an overview of what has already been built and how these existing systems do or do not address portions of the problem statement. A review of applications was conducted to understand what types of application have been applied to FPGAs. These applications provide an understanding of how FPGAs were utilized to run these applications. This information was taken into account when our system was designed.

3.1 Reconfigurable Computing Systems

The majority of related reconfigurable computing systems take the form of computing clusters in which multiple FPGA boards were either networked together or used as accelerators on PCs to achieve increased computational performance.

3.1.1 Splash/Splash 2

Splash and Splash 2 [2] are special-purpose parallel processors that use FPGAs as processing elements. Splash is a reconfigurable linear logic array of XC3000-series Xilinx FPGAs that uses a Peripheral Component Interconnect bus to interface with a host system. Splash 2 consists of two rows of eight Xilinx XC4010 FPGAs, each with a small local memory attached. Both systems are scalable since multiple FPGA boards can be incorporated into a system and multiple systems can be connected together to form even larger systems.

While Splash/Splash 2 addresses scalability, it is a custom-designed system. Since we do not have the resources to design and produce our own boards, it is necessary to use commercial-off-the-shelf (COTS) FPGA boards.

3.1.2 PRISM

Unlike the large-scale Splash systems, PRISM [3] is a small-scale proof-of-concept system, which augments individual general-purpose core processors with FPGAs. In the system, an FPGA, which serves as a coprocessor, is configured to execute the smaller, more frequently executed sections of a program while the general-purpose core processor is responsible for executing the less frequently accessed sections.

PRISM addresses the issue of hardware/software partitioning by assigning the most frequently executed portions of code to the FPGA and the less frequently executed portions of the code to the general-purpose processor. However, the issue with this generalized method of partitioning a program is that it may not improve performance for all types of applications. The most frequently executed portions of a program may actually be better suited for a general-purpose processor. In order for the system to execute a program more efficiently, the FPGA must execute its computations faster than a general-purpose processor.

Otherwise, if a general-purpose processor can execute the same computations faster than the FPGA, there is no benefit to using the FPGA over the general-purpose processor.

3.1.3 SLAAC

The objective of the Systems Level Applications of Adaptive Computing (SLAAC) project [4] is to define an open and scalable heterogeneous distributed adaptive computing systems architecture standard. Part of SLAAC's mission includes creating a COTS reference platform implementation. One such platform, the SLAAC Research Reference Platform (RRP), is defined as a high-speed network cluster of desktop PCs where each PC is enhanced with a reconfigurable accelerator, such as an FPGA board.

3.1.3.1 Tower of Power

An example of an existing RRP is Virginia Tech's Tower of Power (ToP) [4, 5], a system comprised of sixteen Pentium II PCs that are individually equipped with a WildForce board and connected together using an Ethernet and a Myrinet network. A diagram of the ToP is shown in Figure 3.1. In total, the platform contains 80 Xilinx XC4062XL FPGAs and memory banks. As part of the SLAAC project, researchers at Virginia Tech designed and developed a common Application Programming Interface (API) which supports application development for the SLAAC system.

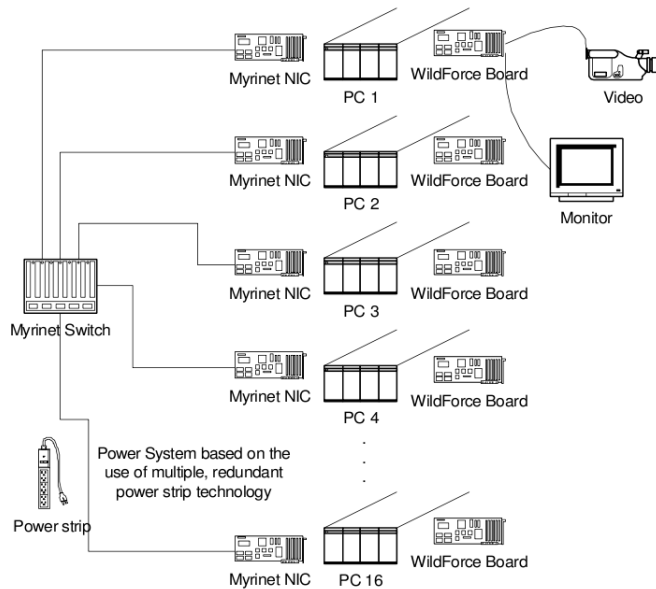


Figure 3.1: Diagram of the Tower of Power [6].

3.1.3.2 Adaptive Computing Systems (ACS) Application Programming Interface (API)

In [5] and [6], the ACS API is described as a development environment for applications on heterogeneous distributed FPGA boards. Its purpose is to provide developers with a simple API for controlling a complex distributed system of interconnected adaptive computing boards. Some applications of the API include HokieGene, a system that implements an enhanced version of a genetic-search algorithm [7] and FIR filters [8].

The ACS API programming model describes a system as a collection of hosts, nodes, and channels. Hosts are user application-level processes that allocate and control nodes, which are hardware resources (typically FPGA accelerator boards). Hosts and nodes are connected together by channels, which allow data to flow throughout system. This data flow is initiated by system creation, streaming data, and memory access routines. System creation allocates nodes and sets up channels. Streaming data functions allow the host application to insert

streaming data into and receive streaming data from the system. Memory access functions include data transfer operations that read, write, and copy memory, as well as generate interrupt signals at nodes.

The ACS API addresses the FPGA development time issue since it: 1) is independent of a specific FPGA architecture; 2) allows a developer to control multiple boards with a single program; and 3) configures the system's communication network. All these characteristics of the ACS API lessen the time it takes for developers to implement an FPGA-based system.

While the ACS API's programming model and functions have characteristics desirable in our system, the configuration of the cluster it was used for is not ideal for our purposes. ToP is a large-scale configuration that simply incorporates too many PCs and too many FPGAs.

3.1.4 Baylor University Cluster

The reconfigurable computing cluster constructed in [9] by Troy consists of 16 XUPV2P boards that are physically arranged in groups of four and networked to a Mini-ITX host PC. Figure 3.2 shows a block diagram of the cluster. In order to quantify execution speedups over a varying number of nodes, Troy measured the run times of multiple partitioned data sets. Using the cluster, Troy scattered test data among the different nodes, all of which contained a single copy of a hardware core implementation of the Triple Data Encryption Standard (3DES) algorithm. His results showed that, while hardware acceleration improves performance, speedup is limited by node communication and file I/O.

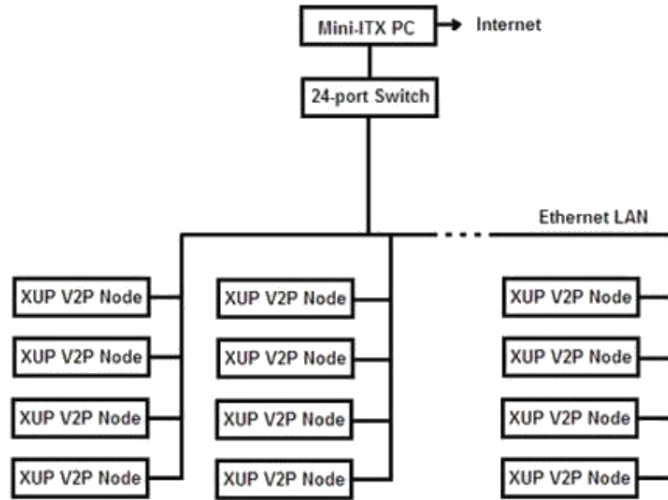


Figure 3.2: Block diagram of the Baylor University cluster [9].

This cluster addresses scalability, one of our design goals, in that FPGA boards can be added or removed from the system. Results included configurations ranging from one board to 15 boards, indicating that this design works for both very small-case to larger-scale configurations. However, while this cluster addresses the finite FPGA fabric issue by utilizing more than one FPGA board, it does not take advantage of all the available FPGA space on each board since each board only contained one 3DES hardware core.

3.1.5 The Reconfigurable Computing Cluster (RCC) Project

The RCC project at the University of North Carolina- Charlotte is a multi-institution, multi-disciplinary project currently investigating the use of Xilinx ML-410 development boards to build cost-effective petascale computers [10, 11]. As of 2008, the project has constructed a prototype cluster consisting of 64 Xilinx Virtex-4 (V4P60) ML-410 Development boards. Figure 3.3 shows a block diagram of the RCC cluster.

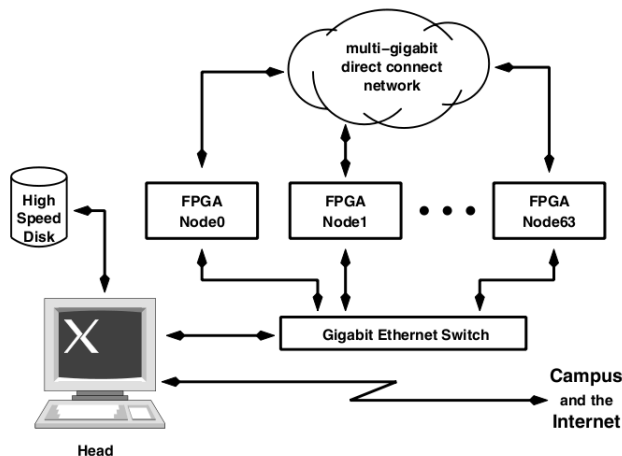


Figure 3.3: Block diagram of the RCC cluster [10].

While petascale computing is outside the scope of this work, we do find the configuration of the RCC cluster to be ideal. It is similar to the design of the Baylor University cluster in that it consists of a host PC and multiple FPGA board nodes connected together via a network switch. Despite the fact that these two clusters incorporate a large number of FPGA boards, they do not use a large number of PCs, as in the ToP system, which is ideal in terms of resources. Like the Baylor University cluster, the RCC cluster addresses scalability since FPGA nodes can be added or removed from the system.

3.2 Types of Applications

In terms of applications often designed for FPGAs, digital signal processing, bioinformatics, and cryptography are typical target applications due to inherent characteristics that are conducive to exploiting parallelism. This section provides a few examples of viable target applications that were considered for our performance analysis. Since our system needs to be flexible and not application-specific, this review revealed what variable application features, such as different types and sizes of input data, need to be accommodated.

3.2.1 Digital Signal Processing

One of the earliest reconfigurable computing applications was signal processing. Digital signal processing (DSP) is concerned with the processing of signals that have been converted into a digital format. DSP technology is used in a variety of areas, such as speech processing, imaging processing, audio processing, information systems, control systems, and instrumentation.

DSP applications possess characteristics that increase the amount of parallelism that can be exploited by reconfigurable computing. First, the operations in many DSP functions follow rather regular schedules. This predictability reduces the amount of control logic needed in the design and allows hardware to be customized to extract a significant amount of parallelism. Second, many DSP applications use small word data widths. Requiring less hardware and less routing, smaller word widths allow more hardware units to fit on a chip and result in higher clock rates. Finally, fixed coefficients or constants are often used in DSP computations. Hardware customized for a given coefficient or constant uses less area and processes operations more quickly.

Discrete Fourier Transforms (DFTs) are typically computed using the Fast Fourier Transform (FFT), an algorithm that efficiently computes a DFT by recursively dividing a DFT into smaller DFTs. In [12], a speed-up of 23 times over a Sparc-10 workstation was achieved for an FFT algorithm implementation on Splash-2, an FPGA-based array processor. Also, in [13], an FPGA implementation of a radix-4 FFT achieved speedup factors of 9.4, 10, and 12.5 over a TMS320C5x DSP processor for FFTs of length 64, 256, and 1024 points, respectively.

3.2.2 Bioinformatics

One of the main focuses of bioinformatics, an interdisciplinary field which combines biology, computer science, and information technology, is analyzing and interpreting nucleotide and amino acid sequences, protein domains, and protein structures. The process of analyzing and interpreting biological data sets is known as computational biology.

Computational biology works with incredibly large sets of data. For example, in 2008, the National Center for Biotechnology Information's GenBank had more than 98 million sequences on record [14]. In addition, many of the algorithms applied to these large data sets are computationally intensive. The complexity of comparison algorithms, for instance, is quadratic with respect to sequence length. The fact that the number of sequences on record continues to grow exponentially every year along with the fact that computational biology algorithms are computationally intensive makes the area an ideal candidate for high-performance computing.

A specific area of study in bioinformatics is sequence alignment, which analyzes similarities between DNA or protein sequences to assess the genetic relationship between organisms. Several studies have highlighted the vast speedups of FPGA sequence alignment implementations. In [15], an FPGA implementation of the Smith-Waterman algorithm was about 330 times faster than a desktop computer with a 1GHz Pentium-III. In another study, mentioned in [16], a global sequence alignment calculation between two DNA sequences on a Splash-2 board performed 1000 times better than a comparable SPARC-1 workstation.

3.2.3 Cryptography

A key element in achieving computer system security is the use of cryptography, the science of encrypting and decrypting data. Since cryptographic algorithms are computationally-

intensive, easily pipelined, and frequently implemented with hardware components, such as shift registers and permutation networks, they are an ideal application for reconfigurable computing. Hardware implementations are well-suited for cryptographic algorithms since they provide high performance and significant resistance to attacks. In particular, modular arithmetic, which is an important element of cryptography, is more efficiently implemented in hardware than with fixed-width microprocessor arithmetic logic units.

The RSA algorithm is a public-key encryption scheme that was developed by Ron Rivest, Adi Shamir, and Len Adleman. The challenge of RSA lies in the factoring of large numbers, a problem which becomes exponentially more difficult to solve as the size of the numbers increases. A fast and efficient factoring algorithm for large numbers has yet to be discovered. The best factoring algorithm runs in sub-exponential time, which is greater than polynomial time, but less than exponential time.

In [17], the 0.8 millisecond decryption time for an FPGA-implemented RSA was about 11 times faster than the 9.1 millisecond decryption time for a 512-bit software implementation on a 150MHz Alpha. Also, the fastest 1024-bit software implementation of 43.3 milliseconds running on a PPro200 based PC was shown to be about 14 times slower than their best result of 3.1 milliseconds.

3.3 Summary

This chapter presented an overview of reconfigurable computing systems and a review of typical FPGA applications. The next chapter describes the design and implementation of our frameworks.

Chapter 4

Design and Implementation

The design goal of the software and hardware core manager frameworks is to provide users with an API to develop applications for, as well as build and control, a complex distributed system consisting of a host PC and multiple FPGA boards.

4.1 System Overview

The system consists of the following components:

- A host PC
- A software framework
- A software application
- FPGA boards
- Hardware core managers
- Hardware cores
- A network switch

Together these components form a distributed system of multiple FPGA boards that parallelizes computations. A graphical representation of the system is illustrated in Figure 4.1.

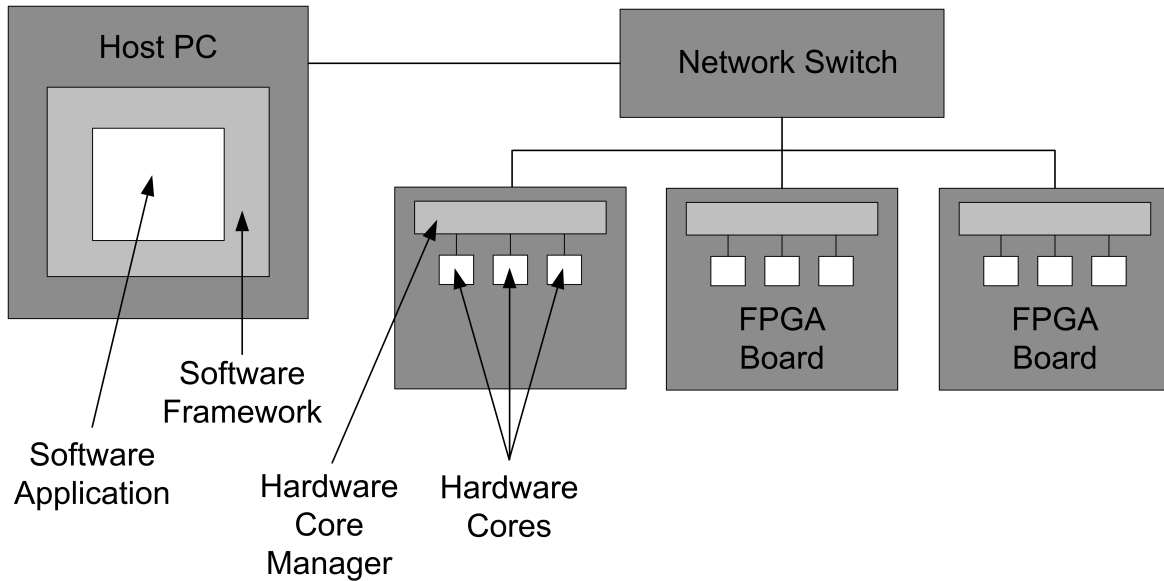


Figure 4.1: Graphical representation of the system.

In terms of operation, the *software framework*, which resides on a *host PC*, provides the user with a library of functions for interacting with the *FPGA boards*. Functionalities provided by the software framework can be grouped into the following categories: initialization, core information set up, reading data, formatting data, sending data, and receiving data. Using these functions, the user has the tools to develop a *software application* that uses the FPGA boards to process data.

When input data read by the software application is sent out to the FPGA boards, it is, more specifically, being sent to the *hardware core managers* running on the FPGA boards. The hardware core manager takes the form of a software application that runs on top the soft core processor implemented on the FPGA. The framework for the hardware core managers includes a set of initialization, core information set up, read data, format data, send data,

and receive data functionalities that are similar to, but independent from, the functionalities defined for the software framework. The hardware core manager serves as the middleman between the user's application and the *hardware cores*, logic blocks that perform a specific computation. This is due to the fact that the hardware core manager has the ability to access the hardware cores directly, writing data sent by the user application to the hardware core and then reading the result computed by the hardware core. This result is returned to the software application, where it is sorted and either read to a file or used by the user in some other manner.

The host PC and FPGA boards are connected together via a *network switch* to allow for data communication between the software application and hardware core managers over a network. Ethernet and lwIP, a implementation of the TCP/IP protocol suite, facilitate network communication.

4.2 Software Framework

The software framework takes the form of an API software library that provides a set of data structures and functions that users can use to build an application that sends data to and receives data from the FPGA boards.

This section describes the components that make up the software framework and provides a description of their their functionalities and responsibilities. For more details on the software framework, see Appendix A, which provides the software framework source code.

Figure 4.2 illustrates the flow of data through the software application created using the software framework.

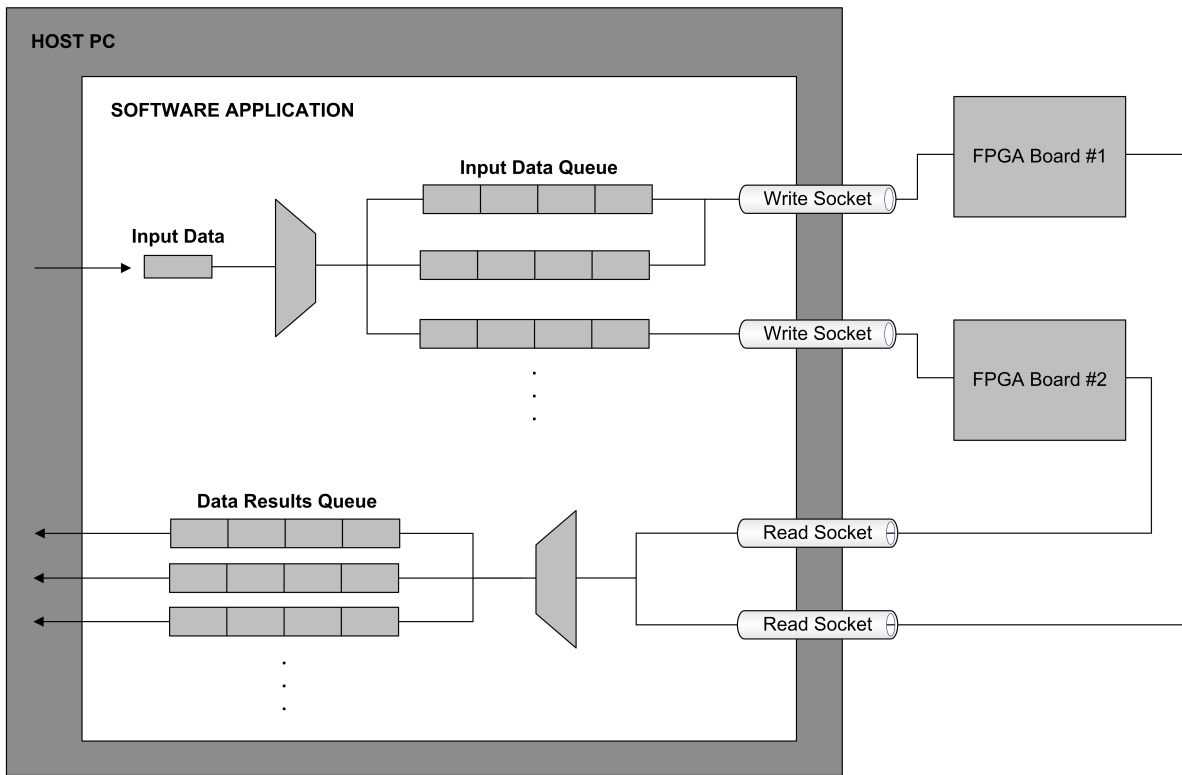


Figure 4.2: Data flow through the software application.

4.2.1 Conventions

The software framework follows the following conventions:

- One thread per input data file
- One thread per data results queue
- One thread per core
- One input data queue per core type
- One data results queue per core type
- One read socket per FPGA board
- One write socket per FPGA board

- One thread per read socket

4.2.2 Initialization

Before an application may begin processing data, communication to the FPGA boards must be established and several data structures must be created and initialized.

4.2.2.1 Communication

The framework provides a `setUpSocket()` function, which is responsible for creating both a read and write socket for each IP address associated with an FPGA board, constructing the FPGA board address structures, and connecting to the FPGA boards. Two separate sockets must be used for sending and receiving due to the fact that the Sockets API of the provided implementation of lwIP is not thread-safe, meaning it is not possible for two different threads to operate on the same lwIP socket without data being corrupted.

4.2.2.2 Data Queues

The software framework provides users with two types of data queues: input data queues and data results queues. The input data queue is a POSIX message queue structure while the data results queue is a linked listed structure. For every different input data file, there is one input data queue. It is assumed that the input data file contains data only intended for one core type. Therefore, an equivalent statement is that there is one input data queue for each different core type in the system. Similarly, there is one data results queue for each different core type in the system. The input data queue is initialized with the `setUpQueue()` function. This function creates the message queues by generating message keys and then initializes the input data queue structure. A separate function called `setUpRList()` constructs the data

results queue.

Finally, there is the `setUpCoreInfo()` function, which requests, collects, and organizes information concerning the number and types of cores available for use in the system. `setUpCoreInfo()` asks an FPGA board to report the number and types of cores it houses. To be aware of the locations of all available hardware cores, the software application must make one call to `setUpCoreInfo()` for every FPGA board present in the system. After collecting core information from every FPGA board in the system, the software application may begin sending data requests to the FPGA boards.

4.2.3 Processing Data

Processing data involves reading input data, formatting data into messages, sending messages, and receiving results.

4.2.3.1 Reading and Formatting Input Data

In terms of input data, the system currently only supports input data read from a file. The `readFile()` function is responsible for reading input data from a file and placing the data into the appropriate input data queue. In the system, there is one input data queue assigned to each different core type present in the system. For example, if the system contained 3DES and FIR filter cores, then there would be two input data queues created, one for 3DES and one for FIR.

A `mapToQueue()` function ensures that data is placed into the right queue. Data is pulled from the input data queue by a core-specific thread. Each core-specific thread is associated with a particular hardware core in the system. Since the software framework is not application-specific, it is the responsibility of the user to develop these core-specific functions.

These core-specific functions are associated with a particular core type in the user-defined `coreMapEntry` data structure. An example of a core-specific function for 3DES is provided in Appendix A.1 and can be used as a guide to creating a core-specific function for another type of core. The core-specific function is responsible for forming and sending the data request message. The process of setting the message type, core type, and job ID portions of the data request message does not vary from core type to core type. The process of sending the data request message out to a particular FPGA board also does not vary from core type to core type. What does vary, however, is the method of constructing the input data portion of the message. For example, 3DES requires five pieces of input data: key 1, key 2, key 3, function select, and data to be processed. These inputs must be gathered from whatever source the user chooses. In our case, all the keys were read in from a user-defined file, function select was assigned from a command-line argument, and the data to be processed was grabbed from an input data file. These pieces of data then had to be concatenated into a collection of bytes and attached to the message type, core type, and job ID to form a data request message. Once the data request message is properly formed, it is sent through to a particular FPGA board through a write socket.

4.2.3.2 Receiving Results

As previously mentioned, there is one read socket for every FPGA board in the system. A thread executing the `recvResults()` function operates on each of these read sockets. `recvResults()` receives a data response message through the socket and checks the core type of the message. Based on the core type, `recvResults()` places the message into the appropriate data results queue using the `addResults()` function. The `addResults()` function places results into a data result queue in order using the job ID. Once all results have been received, the user may either use the `writeResults()` function to write the data to

an output file or use the `removeResults()` function to remove and access results from the queue.

4.3 Hardware Core Manager Framework

The hardware core manager is a software application that controls the data flow into and out of the hardware cores. It runs on the MicroBlaze soft core processor and is responsible for collecting information regarding what types of cores are available for use and sharing this information with the host PC when the information is requested. Another responsibility of the hardware core manager is to route data to and from the hardware cores. Source code for the hardware core manager can be found in Appendix C.

Figure 4.3 shows the flow of data through the hardware core manager.

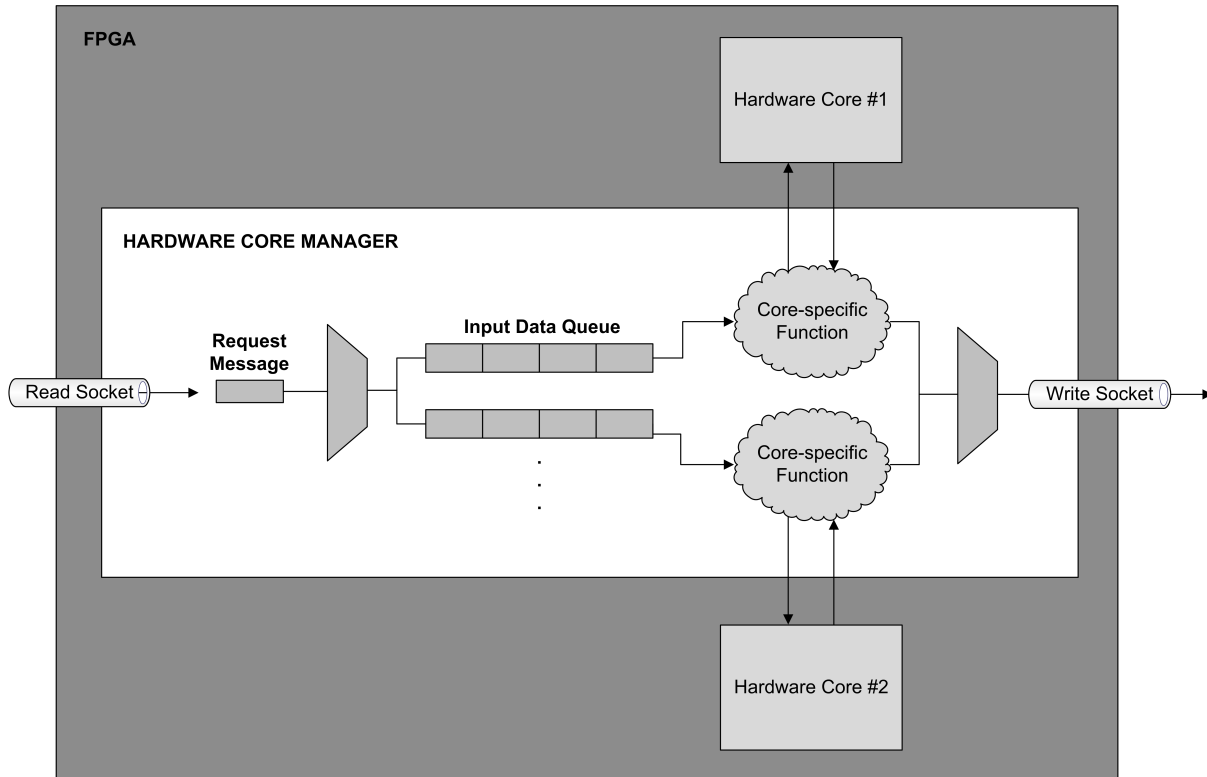


Figure 4.3: Data flow through the hardware core manager.

4.3.1 Conventions

The hardware core manager framework follows the following conventions:

- One read socket
- One write socket
- One thread per read socket
- One thread per core
- One input data queue per core type

4.3.2 Operating System

The hardware core manager runs on top of XilKernel, the operating system for the Xilinx FPGAs. Xilkernel provides such features as threading and message queues.

4.3.3 Initialization

When started, the hardware core manager must first set up its network information. This requires reading the state of the DIP switches present on the FPGA board. The number corresponding to the state of the switches represents the last decimal digit of the IP address to be assigned to the FPGA board. This procedure allows the IP addresses of different FPGA boards to be set by simply changing the DIP switches. It is important, however, that these IP addresses correspond to the IP addresses used by the software application on the host PC.

4.3.4 Receiving Requests

Once the network information has been configured, the hardware core manager must establish a connection to the host PC. This is done by creating two TCP/IP stream sockets, one socket for reading and one socket for writing. Each of these sockets is assigned a different known port number. The hardware core manager then connects to the host PC. Once communication to the host PC has been established, the hardware core manager may begin to receive and process requests, a process that is depicted by the flowchart in Figure 4.4.

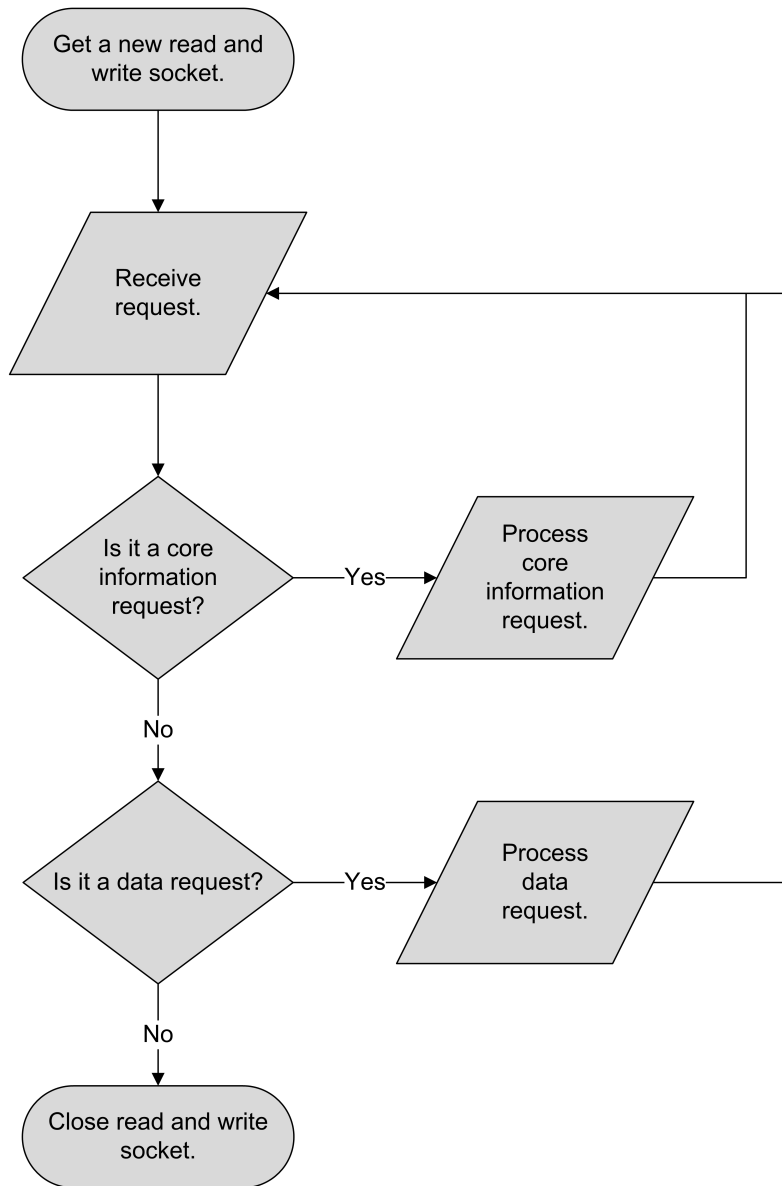


Figure 4.4: Flowchart depicting the receive request process.

4.3.5 Processing Core Requests

A core request is always the first message that the hardware core manager receives from the host PC. When a core request is received, the hardware core manager sets up its core

information, then sends this information to the host PC.

4.3.5.1 Input Data Queue Setup

For every different hardware core type present on the FPGA, an input data queue must be created. Therefore, the convention is that there is one input data queue per core type.

4.3.5.2 Core List Setup

Each FPGA board has a Compact Flash card inserted into its SysACE controller. The Flash card contains a text file which describes each core to be used in the system configuration. An example of the contents of this file is as follows:

```
3DES,0xcea00000,36,8
3DES,0xcea20000,36,8
!
```

Each line of the file contains the following information: core type, core base address, core input data size, and core output data size. In this example, the first line indicates a 3DES core type with a base address of 0xcea00000, a total input data size of 36 bytes, and an output data size of 8 bytes. A stop character, in this case '!', is placed on the last line of the file, so that it is known when the end of the file has been reached.

The process of setting up the core list involves reading the text file containing the core information and parsing it into the necessary data structures. For each core that is set up, a thread is created. This thread is assigned specifically to this core and is therefore responsible to reading/writing data to/from this core. The core information is sent to the host PC so

that the software application is aware of the types and locations of the cores present in the system.

4.3.6 Processing Data Requests

When the hardware core manager receives a data request, it places the message into the input data queue corresponding to the core type of the message. The core thread created in the core list setup procedure checks this queue for new data. When a new piece of data is available, it removes the data and provides the input data to the appropriate core processing function. This function writes the data to the hardware core and reads the data from the hardware core. The result computed by the hardware core is formed into a data response and sent back to the host PC.

4.4 Communication

This section describes the methods with which the software application communicates with the hardware core manager.

4.4.1 Messages

Data is passed between the host PC and FPGA boards as streams of bytes since TCP/IP is a byte-stream oriented communication protocol. Based on this characteristic of TCP/IP, we define a message to be a variable-length packet of bytes that the host PC and FPGA boards use to exchange information.

4.4.1.1 Message Components

Each message, depending on its type, can consist of one or more of the following pieces of information, as illustrated in Figure 4.5: 1) message type; 2) number of cores; 3) core type; 4) job identifier (job ID); 5) input data; and/or 6) output data.

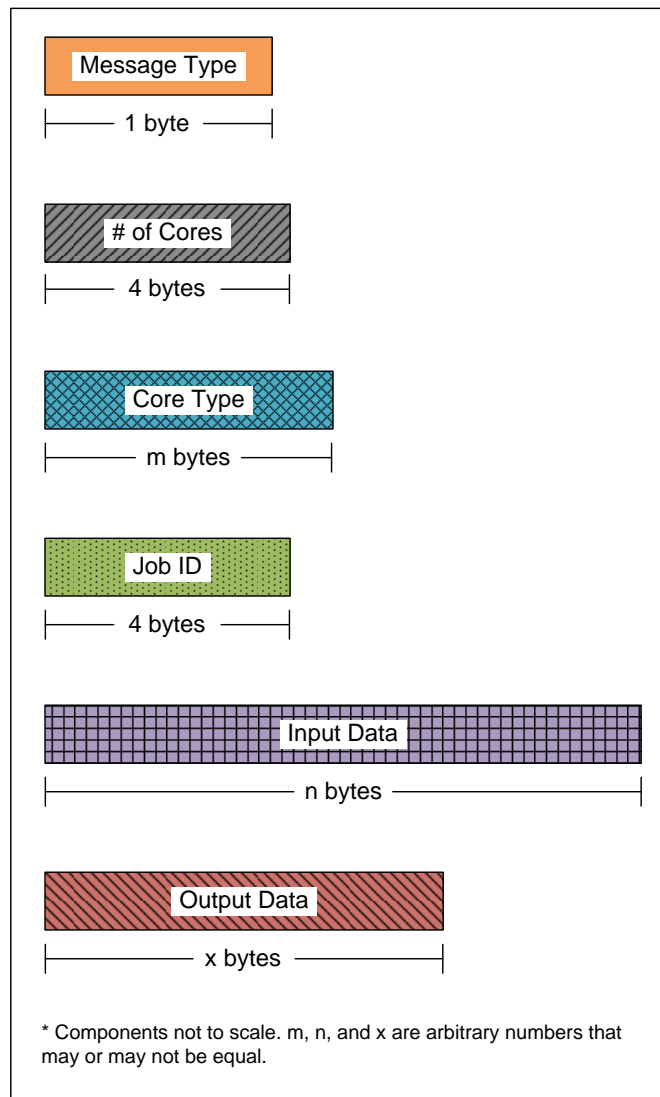


Figure 4.5: Message components.

The *message type* is a one-byte character that enables the hardware core manager to identify and distinguish incoming messages. Our system defines two types of messages types, core

and data. The core request message type is represented by the character, “c”, while the data request message type is represented by the character, “d”. Since message types are defined in a header file (wrapper.h in Appendix A.2), it is possible for the user to define new message types or use a different set of characters to identify messages by simply editing the header file.

Number of cores represents the number of cores present on a particular FPGA board. The system currently defines this component to be four bytes in length, specifically using a C `int` data type. The use of a C `int` data type to store the value for the number of cores means that the maximum number that can be used to represent the number of cores is $2^{31} - 1$. Considering the fact that our system could only handle a maximum of seven hardware cores, four bytes is more than enough bytes to store the value for the number of cores in the system.

The *core type* is a byte array consisting of a user-defined variable-length character string associated with a particular type of core. For our implementation, we defined the core type to be eight bytes long, allowing cores to be identified by a string consisting of a maximum of eight characters. As with the message type, the core type is a flexible message component in that its length can be changed by editing the wrapper.h header file.

The *job ID* is a number representing the order in which the output data corresponding to the message must be arranged in the data results buffer. The length of the job ID is user-defined, but must be large enough to store the maximum number of data requests that will be sent. For our system, four bytes in the form of a C `uint32_t` data type were used to hold the largest possible job ID, meaning the system can support a maximum of $2^{31} - 1$ data requests.

Input data is a byte array containing data supplied to the hardware core. The length of the input data is dependent on the specifications of the hardware core the data is intended for.

For example, input data for the 3DES core consists of 36 bytes that include key 1 (8 bytes), key 2 (8 bytes), key 3 (8 bytes), function select (4 bytes), and data to be processed (8 bytes). In the case where there is more than one input, as in 3DES, all input data are concatenated together when incorporated into a message. The input data are later parsed by a function specific to the core that the data is intended for. For a system consisting of a heterogeneous set of hardware cores, the length of the input data section of the message must be equal to the length of the largest possible input data.

Output data is a byte array holding the result computed by the hardware core. Like the input data, its length is dependent on the specifications of the hardware core the data is intended for. For example, the output data of 3DES is 8 bytes long. Again, for a system consisting of a heterogeneous set of hardware cores, the length of the output data section must be equal to the length of the largest possible output data.

4.4.1.2 Types of Messages

The various message components described in the previous section are used to form the four possible types of messages that can be exchanged between the host PC and the FPGA boards: 1) core information request (core request); 2) core information response (core response); 3) data request; and 4) data response. Requests are messages sent from the host PC to an FPGA board while responses are messages sent from an FPGA board to the host PC. Figure 4.6 shows the format of the different message types. The composition and usage of each of the messages are described in the next section.

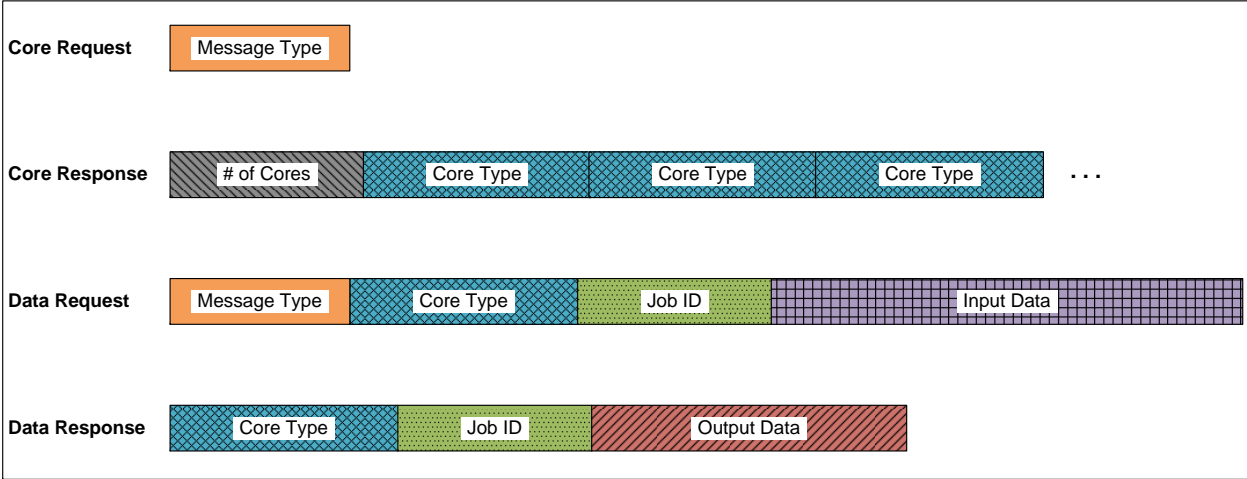


Figure 4.6: Message formats.

4.4.1.3 Exchanging Messages

Figure 4.7 illustrates the exchanging of messages between the host PC (software application) and FPGA board (hardware core manager) over time.

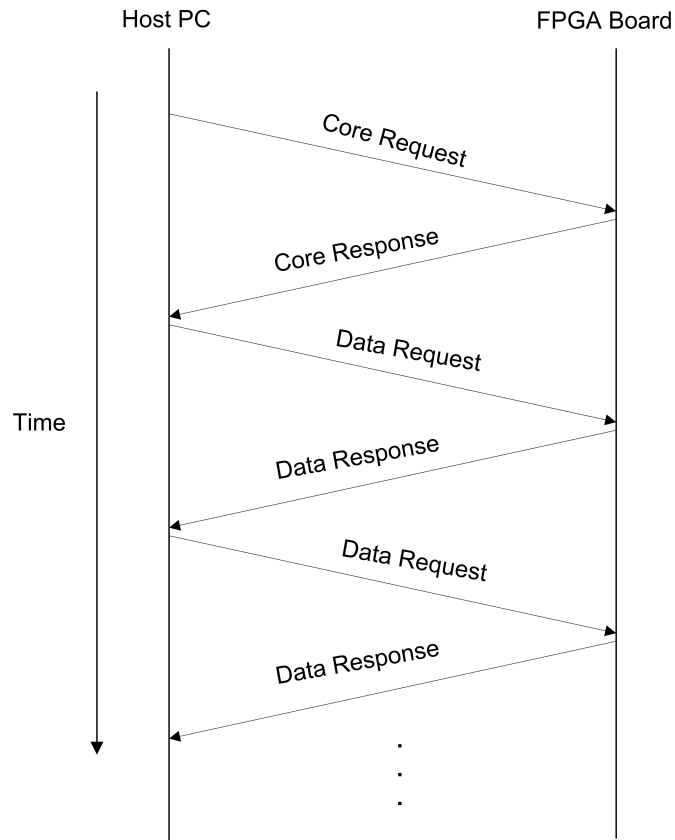


Figure 4.7: Exchanging of messages between the host PC and FPGA board over time.

The first message in a series of exchanges is a *core information request*, a message that only consists of a core request message type. For our system, this is simply the character, “c”. The core information request is only sent once as part of the system initialization of the software application running on the host PC. To be able to distribute input data properly, the application must be aware of what types of cores are present in the system and where (which FPGA board) each core is located. Therefore, before any data can be sent out to an FPGA board, the application must first send a core information request out to each of the FPGA boards.

The hardware core manager checks the message type portion of every message it receives and processes the message based on the message type it sees. If it identifies a core information request, it sends the host PC a *core information response*. In this case, it forms a message that includes the number of cores it has, followed by the core types of these cores.

Once the host PC has received the core information response sent by the FPGA board, it may begin sending *data requests*. A data request message consists of: 1) a message type representing a data request; 2) the core type that the data processing request is for; 3) the job ID of the request; and 4) input data. After the FPGA board has determined that the message it has received is a data request, it checks the core type portion of the message to determine which core type input data queue it should place the input data into. This queue is being constantly checked for new messages by the corresponding core threads. When a message is available, a core thread removes the message from the queue and provides the input data from the message to the hardware core function. This hardware core function is user-defined since each hardware core has potentially different structures/requirements, such as number of inputs, size of input data, size of output data, etc. The hardware core function must parse the input data properly and write it to the hardware core. The hardware core function then reads the result from the hardware core. The result is then concatenated with the core type and job ID to form the *data response*, which is sent back to the host PC.

On the host PC side, a single `recvResults()` thread per FPGA board retrieves data responses and places them into a data results queue. After all results have been received and sorted by job ID, the user do whatever they see fit with the data. If the user wishes to write the data to a file, a function to do so is provided. In this case, a `writeResults()` thread removes data from the results queue and writes them to an output file. Whether the user chooses to write the data to a file or not, the main application thread should not exit until all core message queues are removed and all read and write sockets are closed.

4.5 Summary

This chapter described the design and operation of the software and hardware core manager frameworks. The next chapter discusses the experimental setup and the results of the performance analysis.

Chapter 5

Testing and Results

5.1 Overview

In addition to designing and constructing the software and hardware core manager frameworks, our goal was to evaluate the performance of our system design (a hardware/software configuration) with a software-only solution. Throughput, as defined in Equation 5.1, was the metric used to evaluate system performance.

$$\textit{Throughput} = \frac{\textit{Bits}}{\textit{Second}} \quad (5.1)$$

Bits represents input data file size (total amount of data to be processed) and second represents application runtime.

5.2 Experimental Setup

In Section 4.1, the system was described as consisting of seven components: a host PC, software framework, software application, FPGA boards, hardware core managers, hardware cores, and a network switch. Since the software framework and hardware core managers reflect our unique contributions to this research, they were the only components discussed in depth in Chapter 4. This section, however, will describe the components that were not specifically created for this research, but instead configured for use in our system.

5.2.1 System Components

This section describes the host PC, FPGA board, network, and input data file components of the system. The hardware core used in the system is discussed in Section 5.3.2.

5.2.1.1 Host PC

The host PC is a Dell OptiPlex GX280 desktop PC running Ubuntu 10.04 LTS (Lucid Lynx).

5.2.1.2 FPGA Boards

Virtex-5 LXT [18] platform FPGAs were used for this project. The FPGAs contain one or more Microblaze processors, Ethernet MACs, block RAM, and large arrays of reconfigurable logic, among many other features. The Microblaze is a high-performance 32-bit RISC Harvard architecture soft processor core designed with embedded systems application in mind [19]. The configurable logic blocks on the FPGA fabric can be used to implement a variety of logic, including 18×18 multipliers, adders, and accumulators.

5.2.1.3 Hardware Base System

The hardware base system is the starting foundation for any system configuration implemented on the FPGA. This configuration may be customized to include different hardware cores. Figure 5.1 is a screenshot of the hardware base system's System Assembly view in Xilinx Platform Studio. This window displays the various components included in the design. This particular screenshot is for a system that has been customized to include 3DES hardware cores.

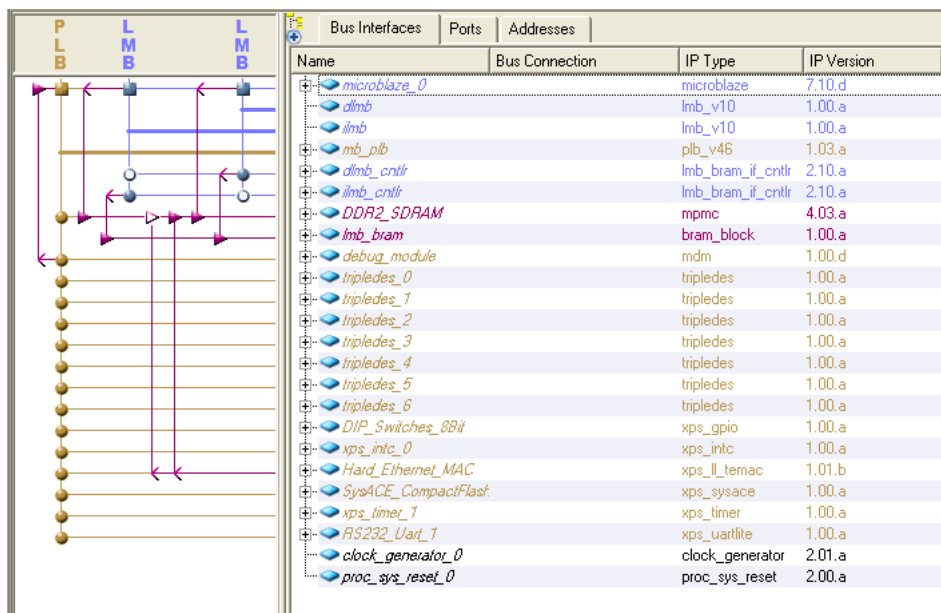


Figure 5.1: System Assembly view screenshot.

The hardware base system consists of the following components:

- MicroBlaze processor
- MicroBlaze Debug Module (MDM)
- Processor Local Bus (PLB)
- Local Memory Bus (LMB)
- Universal Asynchronous Receiver/Transmitter (UART)

- Local Link Tri-Mode Ethernet Media Access Controller (LL_TEMAC)
- Block RAM (BRAM)
- Double Data Rate Synchronous Dynamic Random Access Memory (DDR2 SDRAM)
- DIP Switches
- System ACE Interface Controller (SysACE)
- Interrupt controller
- Timer
- LMB BRAM controller

MicroBlaze The MicroBlaze is a soft 32-bit processor core. On the Virtex 5 FPGA, more than one MicroBlaze may be implemented. However, for our purposes, only one MicroBlaze was included in the design. For our test setup, the MicroBlaze was configured to run at 125 MHz.

MDM The MDM enables JTAG-based debugging for the MicroBlaze processor.

SysACE The SysACE provides an interface to CompactFlash, allowing system configurations, as well as files, to be read off of a Flash card.

PLB The PLB is the system bus that connects the processor to high-speed peripherals. Of the various available CoreConnect peripherals, it offers the highest bandwidth (128-bit data bus or 32-bit address).

LMB The LMB is a 32-bit bus that is designed to have one master and one slave, the MicroBlaze and a memory controller, respectively.

UART The UART core is a low-speed communication core that allows data to be transmitted serially via a RS-232 cable. Since a PC can be easily interfaced to a UART, it is very useful for debugging purposes.

LL_TEMAC The LL_TEMAC is a soft Ethernet core that enables data to be transmitted and received over a network.

BRAM BRAM is on-chip memory within the FPGA.

DDR2 SDRAM DDR2 SDRAM is off-chip memory for the FPGA.

5.2.1.4 Network

In order for the host PC to be able to communicate with the FPGA boards, Ethernet (100 Mbps) and lwIP, a lightweight version of TCP/IP, were used.

Light-weight IP (lwIP) Originally developed by Adam Dunkels at the Swedish Institute of Computer Science, lwIP is a open source, light-weight implementation of the TCP/IP protocol suite [20]. Our particular implementation of lwIP was provided as part of the Xilinx Platform Studio. For the system, the Berkeley Sockets API feature of lwIP was used. It should be noted that lwIP in the sockets API is not thread-safe [21]. One option for guaranteeing that data does not get corrupted is to create two separate sockets- one for reading and one for writing- for each connection.

5.2.1.5 Input Data Files

The input data files used for testing contained random data and were created using the Unix command, `dd`. For example, to generate 1 MB of random data, the following command was used:

```
dd if=/dev/urandom of=1MB.log bs=1024 count=1024
```

`if=/dev/urandom` specifies that data should be read from the file `/dev/urandom`, a special file that, when read, returns random bytes. `of=1MB.log` indicates that the data should be written to a file named `1MB.log`. `bs=1024` is the block size of the data to be written (in bytes) and `count=1024` is the number of blocks to write.

5.2.2 Development Tools

Given that the system was comprised of various different components, several tools were used to build and debug the system. For portions of the system built on top of the FPGAs, including the hardware core manager framework and the hardware cores, the Xilinx Design Suite was used. This set of tools includes the Xilinx Embedded Development Kit, Integrated Software Environment, Platform Studio, Software Development Kit, and ChipScope. For the software framework, the freely-available Eclipse Integrated Development Environment [22] was used for C program development. Finally, for the networking portion of the system, Wireshark [23], a free open-source network protocol analyzer, was used for both network troubleshooting and observing network traffic.

5.3 Test Application

5.3.1 3DES

3DES is an encryption algorithm that applies the DES algorithm to a block of data three times. 3DES uses three 64-bit keys, denoted as k_1 for key 1, k_2 for key 2, and k_3 for key 3.

The algorithm for 3DES encryption is given in Equation 5.2. To encrypt a 64-bit block of data, a DES encryption with k_1 is first applied to the data block. Next, a DES decryption is applied with k_2 to the result of the previous encryption. A final 3DES encryption with k_3 is then applied to produce the final 3DES encrypted data block.

$$ciphertext = E_{K_3}(D_{K_2}(E_{K_1}(plaintext))) \quad (5.2)$$

For decryption, the algorithm shown in Equation 5.3 is used. To decrypt a 64-bit block of data, a DES decryption with k_3 is first applied to the encrypted data block. Next, a DES encryption with k_2 is applied to the result of the previous decryption. Finally, this result is then DES decrypted with k_1 to obtain the decrypted data block.

$$plaintext = D_{K_1}(E_{K_2}(D_{K_3}(ciphertext))) \quad (5.3)$$

3DES was chosen as an application since it possesses characteristics that we can use to evaluate the flexibility design requirement. First, 3DES requires more than one input. Second, 3DES operates on data blocks that are larger than 32 bits. Third, the input data provided to 3DES could contain any sort of combination of bytes.

Choosing 3DES as an application ended up affecting how message passing was implemented.

Since the data that 3DES could be applied to could be any sequence of bytes, this eliminated the possibility of using a delimiter-based method for reading and parsing data.

5.3.2 3DES Hardware Core

Since the focus of this work was not to design and build hardware cores, but instead to build and analyze a system *using* hardware cores, a pre-built hardware core was used. The 3DES core chosen for inclusion in our system was from OpenCores, an open-source repository of hardware components. As of April 2011, the site hosts 800 different IP-blocks [24]. The project page for the particular 3DES core used in our system can be found at [25].

5.3.2.1 Interfacing with the 3DES Core

To read and write data to/from the hardware core with the hardware core manager, a custom data structure for 3DES was created. This data structure allowed for easier access to the software accessible slave registers that are used to read/write data to/from the hardware core since the address offsets of the different slave registers is taken care of by the definition of the custom data structure. Code segment 1 shows the 3DES data structure.

```
typedef struct {  
    long key1_in_A;  
    long key1_in_B;  
    long key2_in_A;  
    long key2_in_B;  
    long key3_in_A;  
    long key3_in_B;  
    long function_select;  
    long data_in_A;  
    long data_in_B;  
    long data_out_A;  
    long data_out_B;  
} tripleDES;
```

Code Segment 1: Data structure for the 3DES core.

After initializing a pointer to the hardware core, data can be written to and read from the hardware core using pointers. An abbreviated example of using pointers to access the hardware core is shown in Code Segment 2. The full code can be found in Appendix C.

```

volatile tripleDES *hw_core = (tripleDES*)(baseAddr);

long key1_in_A;
long key1_in_B;
long data_out_A;
long data_out_B;

// Writing Key 1 data to the core
hw_core->key1_in_A = key1_in_A;
hw_core->key1_in_B = key1_in_B;

// Reading output data from the core
data_out_A = hw_core->data_out_A;
data_out_B = hw_core->data_out_B;

```

Code Segment 2: Using pointers to access the 3DES core.

5.3.2.2 Verification

While only the runtimes for 3DES encryption were recorded, decryption was still used to verify that 3DES had been applied correctly to a file. The following lines show how we used `hexdump` to verify that the data had been processed correctly:

```

$ hexdump decryptedFile.txt > decryptedFileHexDump.txt
$ diff decryptedFileHexDump.txt originalFileHexDump.txt
$

```

Since outputs were in binary format, after a file was encrypted, then decrypted, its contents

were written in hexadecimal format to another file using the `hexdump` command. The hexdump of the output file was then compared to the hexdump file of the original input file using the `diff` command. If `diff` did not produce any output, then the file was successfully encrypted and decrypted.

5.4 Test Scenarios

In order to observe what effect varying the number and configuration of hardware processing elements (hardware cores and FPGA boards) has on the system, the following test scenarios were used:

- Single core, single board
- Multiple cores, single board
- Single core per board, multiple boards
- Multiple cores per board, multiple boards

Tests were run a total of ten times, after which the processing times were averaged and the throughput calculated. Recorded processing times reflect times for an application conducting 3DES encryption, starting from system initialization and ending once all data has been returned to the software application. These times were generated using the Unix `time` command. Variables that were changed between tests include: input data file size (total amount of data to be processed), number of cores, and number of boards.

In addition to these tests scenarios, a software implementation of 3DES was also tested in order to compare the performances of a hardware/software solution versus a software-only solution.

5.4.1 Single Core, Single Board Configuration

The single core, single board configuration serves as the base case for the performance analysis. The calculated throughputs of configurations which incorporated additional hardware are compared to this particular configuration in order to determine whether a gain in performance was achieved or not. Table 5.1 indicates that the baseline throughput is 128 bytes/second for a 1 kB input data file size and 135 bytes/second for a 10 kB input data file size.

Number of Cores	Average Processing Time (seconds)		Throughput (Bytes/Second)	
	1 kB	10 kB	1 kB	10 kB
1	8.0	76.0	128	135

Table 5.1: Single core, single board configuration throughput results.

5.4.2 Multiple Cores, Single Board Configuration

Following the single core, single board configuration, we performed a series of tests to observe what effect incorporating additional hardware into the system would have on performance. Additional hardware can be incorporated into the system in the form of additional hardware cores or additional FPGA boards. Our first approach involved integrating more hardware cores.

For the multiple cores, single board configuration, a total of six separate test runs were conducted. The first test was a two core configuration. Each subsequent test added another core to the board until a maximum of seven cores were present in the system. Only seven cores could be configured into the system due to the fact that there is a limit to how many peripherals can be attached to the PLB bus. As defined in the data sheet for the PLB,

the maximum supported allowable value for the number of PLB slaves is 16 [26]. After subtracting the number of required peripherals that are attached to the PLB bus, such as those listed in Section 5.2.1.3, only room for seven additional peripherals is available.

Before conducting the tests for this configuration, we hypothesized that each additional hardware core would increase performance by some given amount since adding more cores would increase the number of processing elements in the system; however, the results proved this hypothesis to be incorrect. Table 5.2, which combines the results for the single core, single board and multiple cores, single board configurations, reveals that adding additional cores to the system did not result in any gain in performance. In fact, throughput remained at relatively the same rate for all configurations, with the percentage change from integrating an additional core being as high as ± 2.2 percent and as low as 0 percent. This low percentage change is reflected in Figure 5.2, which graphically depicts the data from Table 5.2.

Number of Cores	Average Processing Time (seconds)		Throughput (Bytes/Second)		Percentage Change of Throughput (%)	
	1 kB	10 kB	1 kB	10 kB	1 kB	10 kB
1	8.0	76.0	128	135	0	0
2	8.0	74.2	128	138	0	2.2
3	8.0	74.4	128	138	0	0
4	8.0	74.4	128	138	0	0
5	8.1	74.3	126	138	-1.5	0
6	8.1	75.9	126	135	0	-2.2
7	8.2	74.4	125	138	-0.8	2.2

Table 5.2: Multiple cores, single board configuration throughput results.

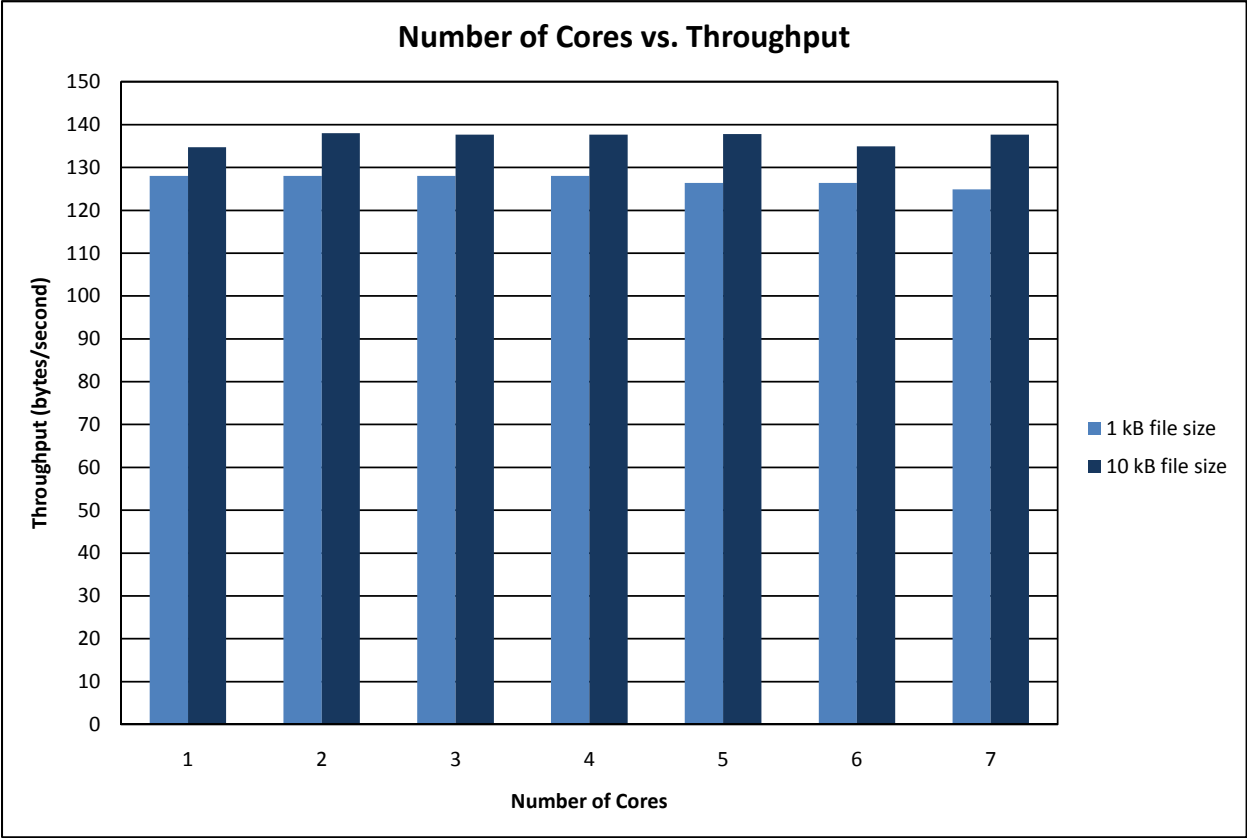


Figure 5.2: Graph of number of cores versus throughput.

5.4.3 Single Core Per Board, Multiple Boards Configuration

After collecting data for all the multiple cores, single board configurations, we next tested what effect incorporating additional hardware into the system would have on performance using our second approach of incorporating additional boards. For this data set, tests were conducted up to three boards since we only had three boards at our disposal.

In contrast to all previously-tested configurations, the single core per board, multiple board configuration showed increased performance when additional hardware was incorporated into the system. Table 5.3 shows that adding additional boards to the system increased throughput by roughly between 1.5 to 2 times the single board configuration.

It should be noted that seeing improved performance with the single core per board, multiple boards configuration is dependent on thread scheduling. Initial test runs for a three board configuration revealed that a single core thread was starving the two other core threads, causing data to only be sent to a single board. Since the two other boards in the configuration were not receiving any data, they were not utilized, and the system emulated a single core, single board configuration. To fix the starvation issue and ensure that all boards were utilized, `sched_yield()` was incorporated into the code. `sched_yield()` forced threads to give up control of the CPU once they were finished sending data to a particular board.

Number of cores	Average Processing Time (seconds)			Throughput (Bytes/Second)			Percentage Change of Throughput (%)		
	1 kB	10 kB	100 kB	1 kB	10 kB	100 kB	1 kB	10 kB	100 kB
1	8.0	76.0	756.3	128	135	135	0	0	0
2	5.3	39.3	378.6	193	261	270	50.8	93.3	100
3	3.7	26.5	256.6	277	386	399	43.5	47.9	47.8

Table 5.3: Single core per board, multiple boards throughput results.

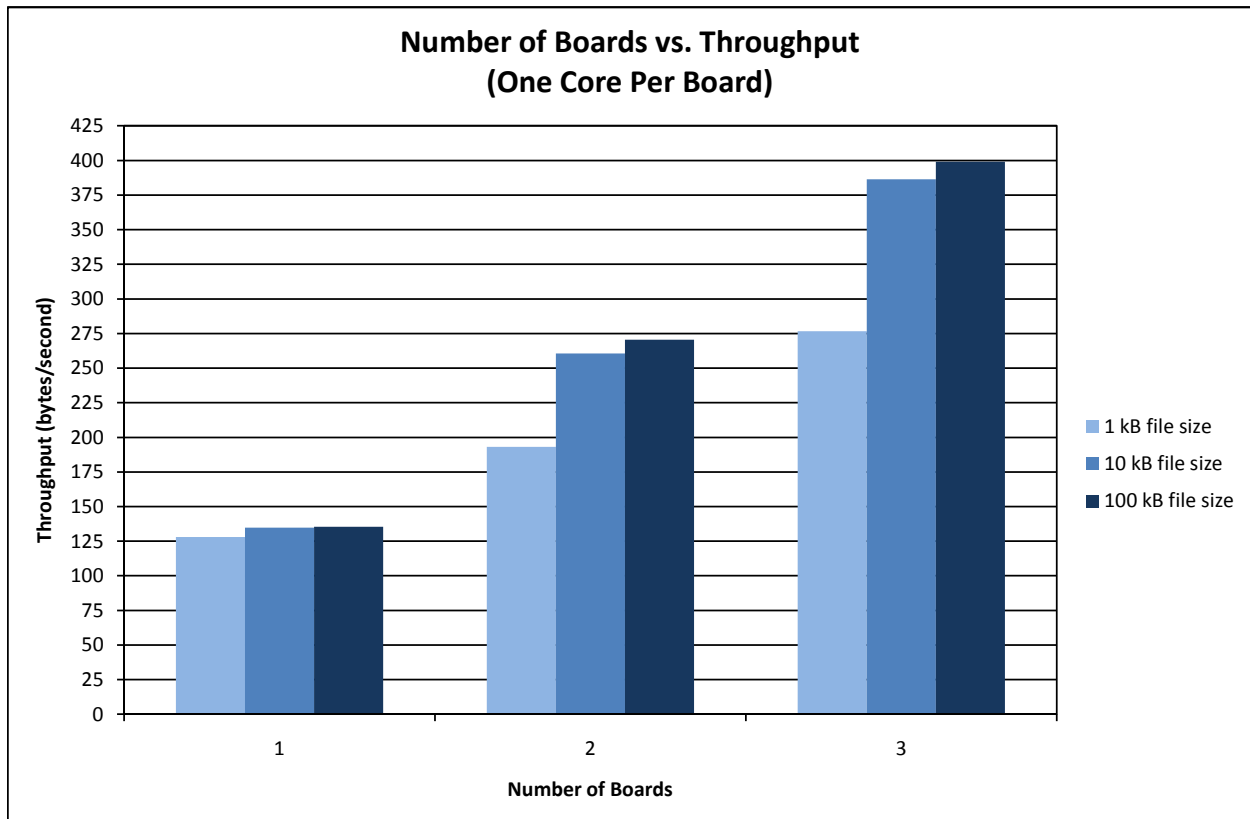


Figure 5.3: Graph of number of boards versus throughput (one core per board).

5.4.4 Multiple Cores Per Board, Multiple Boards Configuration

Based on the results of the multiple cores per board, single board configuration, the conclusion was that multiple cores per board, multiple boards configurations would not yield increased performance. In fact, the multiple cores per board, single board configuration results suggested that multiple cores per board, multiple boards configurations would perform similarly to a single core per board, multiple boards configuration. While it was decided that it was unnecessary to iterate through all the possible multiple cores per board, multiple boards configurations since they would not generate any significant data, two tests were conducted to verify the theory that multiple cores per board, multiple boards configurations would produce similar results to a single core per board, multiple boards configuration.

These two tests included a test for the median number of cores per board (four cores per board for a total of 12 cores in the system), as well as the maximum number of cores per board (seven cores per board for a total of 21 cores in the system). In choosing the median and maximum number of cores per board, we can assume that throughputs for the configurations that were not tested would not vary significantly from the calculated throughputs of the median and maximum number of cores per board configurations.

As shown in Table 5.4, having four or seven cores per board in a three board configuration for a total of 12 and 21 cores in the system, respectively, did not noticeably impact throughput. At most, a -2.9 percentage change in throughput going from a single core per board to seven cores per board was observed. This data supports our theory that multiple cores per board, multiple boards configurations would not yield increased performance.

Number of Cores (Per Board)	Average Processing Time (seconds)		Throughput (Bytes/Second)		Percentage Change of Throughput From Single Core (%)	
	1 kB	10 kB	1 kB	10 kB	1 kB	10 kB
1	3.7	26.5	277	386	0	0
4	3.6	26.6	284	385	2.5	-0.3
7	3.8	26.7	269	384	-2.9	-0.5

Table 5.4: Multiple cores per board, multiple boards configuration throughput results comparison: one core per board vs. four cores per board vs. seven cores per board.

5.5 Software Implementation

To evaluate the performance of the various FPGA board and hardware core configurations, a software-only implementation of 3DES was written using MATLAB for comparison.

The MATLAB code for 3DES was created with the help of code from [27]. Only MATLAB

source code for DES encryption was provided, so code for DES decryption was created by modifying the DES encryption code. DES encryption and decryption were then used to form the 3DES algorithm. Source code for the MATLAB implementation is provided in Appendix D.

Recorded processing times reflect times for 3DES encryption; however 3DES decryption was used to verify the output files. The output was verified using the “Compare Selected Files” feature in MATLAB. The profiler feature was used to record runtime. The same input data files and input keys used for the FPGA-based configurations were used for the MATLAB implementation. Similarly, tests were run a total of ten times.

Results showed that the MATLAB implementation of 3DES achieved a better throughput than the best platform configuration. Even the best throughput of the single core per board, multiple boards configuration of just under 400 bytes/second, shown in Table 5.3, could not compare to the performance of the MATLAB implementation, whose throughput leveled off at around 600 bytes/second, as shown in Table 5.5.

File Size	Average Processing Time (seconds)	Throughput (bytes/second)
1 kB	2.4	427
10 kB	16.9	606
100 kB	170.0	602
1 MB	1752.7	598

Table 5.5: 3DES MATLAB performance results.

5.6 Preliminary Conclusions

Looking over the data collected for all the various system configurations, we can draw some preliminary conclusions. First, of the two approaches- additional cores and additional boards-

used to evaluate the effect of adding additional hardware into the system, only the addition of more boards with a single core on each board demonstrated an improvement in performance. However, even the best single core, multiple board configuration, could not achieve the same or greater performance than the MATLAB software implementation. Since hardware typically achieves higher performance than software, particularly in the case of 3DES [9], the fact that the software implementation of 3DES performed better than our hardware/software solution using FPGAs suggests that a performance bottleneck exists within the system. In the next section, we explore possible sources of the bottleneck.

5.7 Performance Bottleneck

Based on how data flows through the system, we can identify five potential locations for the bottleneck:

- Hardware core
- File I/O
- Network transmission
- Software framework
- Hardware framework

We provide qualitative data on the hardware core, file I/O, and network transmission since we have the means to isolate those components and observe their contribution to the average processing time. For the software and hardware frameworks, we offer theories on the possibility of an existing bottleneck.

5.7.1 Hardware Core

To observe the impact of the 3DES computation on overall performance, we remove the core from the system and replace it with a dummy core. The structure of the dummy core resembles that of a 3DES core in that it takes in the same number and size of input data, as well as output data. The difference between the cores is that the dummy core does not perform the 3DES computation, instead it simply assigns input data to the output. By using a dummy core, the time it takes for the 3DES computation to be executed in hardware is removed from the overall processing time.

Table 5.6 provides the 1 kB input data file and 10 kB input data file average processing times and throughputs for the dummy core and 3DES core. Comparing these values, we notice that the difference between the throughputs is small- only a 0.7 percentage change of throughput was observed between the dummy core and 3DES core in the case of a 10 kB input data file size while no change was observed with the 1 kB input data file size. Based on these values, we can eliminate the hardware core as the source of the bottleneck.

File Size	Average Processing Time (seconds)		Throughput (Bytes/Second)		Percentage Change of Throughput (%)
	Dummy Core	3DES	Dummy Core	3DES	
1 kB	8.2	8.2	125	125	0
10 kB	75.0	74.4	137	138	0.7

Table 5.6: Hardware core performance comparison.

5.7.2 File I/O

After determining that the 3DES hardware core was not the source of the bottleneck, we looked to see if file I/O was limiting performance. Removing the amount of time it takes to conduct file I/O involved using `gettimeofday()` to measure elapsed time starting from the time that all input data has been read into an input data buffer and ending when all data has been received back from the FPGA boards.

As with the case with the hardware core, no significant differences in performance were observed; a -7.4 percentage change was calculated between the throughput without file I/O and the throughput with file I/O for the 1 kB input data file size while only a 0.7 percentage change was calculated with the 10 kB input data file size. This data suggests that the bottleneck is not a result of file I/O.

File Size	Average Processing Time (Seconds)		Throughput (Bytes/Second)		Percentage Change of Throughput (%)
	No File I/O	With File I/O	No File I/O	With File I/O	
1 kB	7.6	8.2	135	125	-7.4
10 kB	74.5	74.4	137	138	0.7

Table 5.7: Processing times excluding time for file I/O (1 board, 7 cores per board configuration)

5.7.3 Network Transmission

Finally, to estimate the time that data spends traveling through the network, we refer back to Section 4.4, which describes the messages that are exchanged between the host PC and FPGA boards.

While there are four types of messages that are exchanged, the bulk of the messages are data requests and data responses. Therefore, we generalize the amount of data that flows through the network by only considering data requests and data responses. While core requests and core responses do contribute to network traffic, since there is only one core request and one core response required per FPGA board, the contribution, in comparison to that of the total number of data requests and responses is rather small.

A data request for 3DES requires a core type (8 bytes), job ID (4 bytes), and input data (36 bytes) components, amounting to a total of 52 bytes. A data response for 3DES requires a core type (8 bytes), job ID (4 bytes), and output data (8 bytes) components for total of 24 bytes. A message header (4 bytes), which stores the total length of a message, must precede a data request/response, so that the the number of bytes to be read is known. Since the system uses Ethernet and TCP/IP, an additional 54 bytes (20 bytes for a typical IP header, 20 bytes for TCP header, and 14 bytes for Ethernet header) must be added to the data request and response. In total, a data request requires 106 bytes and a data response requires 78 bytes. Summing these up, every 8 bytes of input data to be processed by a 3DES hardware core requires 184 bytes to be transmitted over the network.

We can calculate the total amount of bytes transmitted over the network using Equation 5.4, where n represents the total number of bytes transmitted over the network, f equals the input data file size in bytes, d is the input data size in bytes, q is the bytes required for a data request, and r is the bytes required for a data response.

$$n = \frac{f}{d} \times (q + r) \quad (5.4)$$

Using a 10 kB input data file as an example, we can calculate the total number of bytes transmitted over the network as follows:

$$\frac{10240 \text{ bytes}}{8 \text{ bytes}} \times (106 \text{ bytes} + 78 \text{ bytes}) = 235520 \text{ bytes}$$

Since our Ethernet configuration has the capability of sending 100 Mbps, we can estimate that the total time that data spends on the network is:

$$\frac{\text{total bytes transmitted over the network}}{\text{network throughput}} = \frac{235520 \text{ bytes}}{100 \text{ megabits per second}} = 18.8 \text{ ms}$$

Comparing 18.8 ms to the processing times provided in Tables 5.2 and 5.3, which show the average processing times for a 10 kB input data file ranging from 26.5 to 74.2 seconds, reveals that time that data spends traveling through the network only amounts to a small portion of the overall processing time. Applying this same method of measuring network transmission time to the 1 kB input data file case, we come to the conclusion that the bottleneck is not related to the time that data spends traveling through the network.

5.7.4 Analysis

Of the five potential causes of a performance bottleneck, quantitative data was collected for the hardware core, file I/O, and network transmission. This data indicated that the bottleneck is not caused by the hardware core, file I/O, or network transmission, suggesting that the problem exists in either the software or hardware core manager frameworks.

When evaluating the effect of incorporating additional hardware into the system on performance, we observed improved performance in the single core per board, multiple boards configuration, but not in the multiple core, single board configuration. Based on this observation, we theorize that a bottleneck exists in the hardware core manager framework as a result of the system convention of there only being one input data queue per core type.

Queues must be protected from being accessed by multiple threads at the same time in order to ensure that data is read and written properly. As a result of this protection scheme, only one thread is allowed to access the queue at a time; therefore, there is a lack of parallelism since other threads must wait their turn to pull data from the queue. We speculate that the performance gain in the single core per board, multiple boards configuration was due to the fact that, by adding additional boards to the system, additional hardware core managers were made available for processing. The presence of additional hardware core managers meant that there were more input data queues present in the system. Since each of these input data queues was on a separate board and independent of each other, it would, in theory, be possible for each of these queues to be operated on at the same time, thus achieving parallelism. In the next chapter, we suggest potential methods of eliminating the performance bottleneck that can be explored as future work.

5.8 Summary

This chapter discussed the results gathered from the performance analysis of the system. The next chapter summarizes this research and concludes with a discussion of potential future work.

Chapter 6

Conclusion

6.1 Summary

This work involved designing and developing a software framework that provides users with an API to develop a software application that interfaces with multiple FPGA boards and a hardware core manager framework that gives users the ability to configure and interact with multiple FPGA boards and/or hardware cores. We demonstrated that the system is flexible, in that it could accommodate various application requirements, such as multiple inputs and inputs larger than 32 bits. We also showed that the system is scalable, in that it could accommodate multiple hardware cores and FPGA boards. Using an application developed using the frameworks, we performed an analysis of various system configurations to observe the effects of incorporating additional hardware components (FPGA boards and hardware cores) on performance. While the results of our single core per board, multiple board configuration test scenario showed an increase in performance, the results of the other test configurations and the software implementation revealed that a performance bottleneck exists in the system. With a series of tests meant to probe the system for the bottleneck, we

were able to eliminate the hardware core, file I/O, and network transmission as sources of the bottleneck. For the two remaining possible bottleneck locations, the software framework and the hardware core manager framework, we offered theories on what could potentially be causing the bottleneck. Suggestions for fixing the potential bottleneck in these areas, are described in the following section, which discusses future work.

6.2 Future Work

One area of potential future research is exploring methods to improve the performance of multiple core configurations. The results presented in Chapter 5 revealed that there was no performance gain when multiple cores were added to the system. In Section 5.7.4, we offered a theory that the one input data queue per core type system convention, in combination with the need to protect input data queues from multiple access, caused there to be a lack of parallelism in the hardware core manager framework. Noting that the single core per board, multiple boards configuration achieved improved performance, exploring methods of incorporating multiple hardware core managers on a single board appears to be a viable path to increasing parallelism in the hardware core manager framework. This solution could potentially be achieved by configuring an FPGA to support more than one MicroBlaze processor, therefore allowing more than one hardware core manager to operate on the FPGA. Another option to increase parallelism would be to redesign the hardware core manager as a hardware component.

In addition to exploring methods of increasing parallelism, more work could be done in terms of application development using the frameworks. This work only covered one application, 3DES. Future work could involve developing applications of a different category, such as digital signal processing or bioinformatics.

Bibliography

- [1] D. M. Harris and S. L. Harris, *Digital Design and Computer Architecture*. Elsevier, 2007.
- [2] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2007.
- [3] L. Agarwal, M. Wazlowski, and S. Ghosh, “An asynchronous approach to efficient execution of programs on adaptive architectures utilizing FPGAs,” in *IEEE Workshop on FPGAs for Custom Computing Machines*, 1994, pp. 101–110.
- [4] SLAAC, “SLAAC.” [Online]. Available: <http://slaac.east.isi.edu/>
- [5] K. Yao, “Implementing an application programming interface for distributed adaptive computing systems,” Master’s thesis, Virginia Polytechnic Institute and State University, 2000.
- [6] M. Jones, L. Scharf, J. Scott, C. Twaddle, M. Yaconis, K. Yao, P. Athanas, and B. Schott, “Implementing an API for distributed adaptive computing systems,” in *Proceedings of the Seventh Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 1999, pp. 222–230.

- [7] K. Puttegowda, W. Worek, N. Pappas, A. Dandapani, P. Athanas, and A. Dickerman, “A run-time reconfigurable system for gene-sequence searching,” in *VLSID '03: Proceedings of the 16th International Conference on VLSI Design*. Washington, DC, USA: IEEE Computer Society, 2003, p. 561.
- [8] Z. S. Nakad, “High performance applications on reconfigurable clusters,” Ph.D. dissertation, Virginia Polytechnic Institute and State University, 2000.
- [9] W. S. Troy, “Construction and validation of a reconfigurable computer cluster,” Master’s thesis, Baylor University, 2009.
- [10] R. Sass, W. V. Kritikos, A. G. Schmidt, S. Beeravolu, and P. Beeraka, “Reconfigurable computing cluster (RCC) project: Investigating the feasibility of FPGA-based petascale computing,” in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 127–140.
- [11] A. G. Schmidt, W. V. Kritikos, S. Datta, and R. Sass, “Reconfigurable computing cluster project: Phase I brief,” in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2008, pp. 300–301.
- [12] N. Shirazi, P. M. Athanas, and A. L. Abbott, “Implementation of a 2-D fast fourier transform on a FPGA-based custom computing machine,” in *Proceedings of the 5th International Workshop on Field-Programmable Logic and Applications*, 1995, pp. 282–292.
- [13] R. J. Petersen and B. L. Hutchings, “An assessment of the suitability of FPGA-based systems for use in digital signal processing,” in *in Digital Signal Processing. In 5th International Workshop on Field-Programmable Logic and Applications*, 1995, pp. 293–302.

- [14] National Center for Biotechnology Information GenBank Statistics, “GenBank Growth.” [Online]. Available: <http://slaac.east.isi.edu/>
- [15] L. Hasan, Z. Al-Ars, and S. Vassiliadis, “Hardware acceleration of sequence alignment algorithms - an overview,” in *International Conference on Design and Technology of Integrated Systems in Nanoscale Era*, 2007, pp. 92–97.
- [16] M. B. Gokhale and P. S. Graham, *Reconfigurable Computing: Accelerating Computation with Field-Programmable Gate Arrays*. Springer, 2005.
- [17] T. Blum, “Modular exponentiation on reconfigurable hardware,” Master’s thesis, Worcester Polytechnic Institute, 1999.
- [18] Xilinx, “Virtex-5 family overview,” 2009. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds100.pdf
- [19] —, “Microblaze processor reference guide,” 2008. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf
- [20] “Light-weight IP.” [Online]. Available: <http://savannah.nongnu.org/projects/lwip/>
- [21] “Light-weight IP Documentation Wiki- LwIP and multithreading.” [Online]. Available: http://lwip.wikia.com/wiki/LwIP_and_multithreading
- [22] “Eclipse.” [Online]. Available: <http://www.eclipse.org/>
- [23] “Wireshark.” [Online]. Available: <http://www.wireshark.org/>
- [24] OpenCores, “OpenCores Statistics.” [Online]. Available: <http://opencores.org/numbers>
- [25] “3DES (Triple DES) / DES (VHDL) :: Overview.” [Online]. Available: <http://opencores.org/project,3des.vhdl>

- [26] “Processor Local Bus (PLB) v4.6 (v1.04a).” [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ds531.pdf
- [27] “Alexander Stanoyevitch’s Cryptography Web Page.” [Online]. Available: <http://www.csudh.edu/math/astanoyevitch/cryptography.html>

Appendix A

Software Framework

A.1 wrapper.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include "wrapper.h"

// printf mutex
pthread_mutex_t stdoutMutex = PTHREAD_MUTEX_INITIALIZER;

// socket mutex
pthread_mutex_t sendMutex = PTHREAD_MUTEX_INITIALIZER;

// thread ID mutex
pthread_mutex_t threadIDMutex = PTHREAD_MUTEX_INITIALIZER;

// thread IDs
pthread_t threadID[NUM.BOARDS];
int threadIDCount = 0;

int fileSize ;
int coreCount = 0;

/*****

Function: setupSocket

Usage: sets up sockets and establishes connect to FPGA boards

Parameter Definition:
- fpgaIP: IP address of FPGA board
```

– *fpgaPort*: known port

Return value: socket descriptor

NOTE: must use this function twice for every connection to an FPGA board since we need one read socket and one write socket per FPGA board

```
*****/
int setupSocket(char *fpgaIP, unsigned short fpgaPort) {

    int sock;
    struct sockaddr_in fpgaAddr;

    if ((sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0) {
        fprintf(stderr, "Error_creating_socket!\n\r");
        exit(1);
    }

    memset(&fpgaAddr, 0, sizeof(fpgaAddr));
    fpgaAddr.sin_family = AF_INET;
    fpgaAddr.sin_addr.s_addr = inet_addr(fpgaIP);
    fpgaAddr.sin_port = htons(fpgaPort);

    if (connect(sock, (struct sockaddr *) &fpgaAddr, sizeof(fpgaAddr)) < 0) {
        fprintf(stderr, "Error_connecting!\n\r");
        exit(1);
    }

    return sock;
}

```

*****/

Function: setUpQueues

Usage: sets up input data queues

```
*****/
void setUpQueues() {
    int queueIndex;

    for (queueIndex = 0; queueIndex < NUM_CORE_TYPES; queueIndex++) {
        int msgidOut;
        int msgidIn;

        strcpy(coreQueue[queueIndex].coreType, coreMap[queueIndex].coreType);

        // Message key needs to be unique– if you add NUM_CORE_TYPES to the
        // index to create the key value for the result queues, you will
        // guarantee that you won't get the same key value
        msgidOut = msgget((key_t) queueIndex, (0666 | IPC_CREAT));

        if ((msgidOut < 0)) {
            fprintf(stderr, "Error_creating_queues!!!\n\r");
            exit(1);
        }

        // Initialize core queue information
        coreQueue[queueIndex].msgid = msgidOut;
        coreQueue[queueIndex].numReceived = 0;
        pthread_mutex_init(&coreQueue[queueIndex].numReceivedMutex, NULL);
        coreQueue[queueIndex].numProcessed = 0;
        pthread_mutex_init(&coreQueue[queueIndex].numProcessedMutex, NULL);
    }
}

```

```

    }
}

/*****

Function: setUpCoreInfo

Usage: Requests core information from FPGA board and builds
core list based on this information

Parameter Definition:
- writeSock: write socket
- readSock: read socket
- cores: list of cores in system
- numCores: number of cores in system

Return value: number of cores set up

*****/
int setUpCoreInfo(int writeSock, int readSock, ipCore *cores, int numCores) {
    int bytesRcvd;

    uint8_t coreMsgChar[MSG_TYPE_SIZE];

    char coreType = CORE_MSG_TYPE;

    memcpy(coreMsgChar, &coreType, MSG_TYPE_SIZE);

    //Send the string to the server
    if (sendMsg(writeSock, coreMsgChar, sizeof(coreMsgChar))
        != sizeof(coreMsgChar) + sizeof(int)) {
        fprintf(stderr, "Error:_sendRequest_sent_an_unexpected_number_of_bytes!\n\r");
        return -1;
    }

    int coreInfoSize = sizeof(int) + (CORE_TYPE_SIZE * numCores);
    uint8_t coreInfo[coreInfoSize];
    uint8_t *coreInfoPtr = coreInfo;

    // Receive core information
    //
    // Use < for single board configurations
    // Use > for multiple board configurations
    if ((bytesRcvd = recvMsg(readSock, coreInfo, coreInfoSize)) > coreInfoSize) {
        fprintf(stderr, "Error:_received_unexpected_number_of_bytes!\n\r");
        exit(1);
    }

    int coresRcvd = 0;

    memcpy(&coresRcvd, coreInfoPtr, sizeof(int));

    coresRcvd = ntohl(coresRcvd);

    coreInfoPtr += sizeof(int);

    //Receive core information
    int i;
    for (i = 0; i < coresRcvd; i++) {

        memcpy(cores[coreCount].coreType, coreInfoPtr, CORE_TYPE_SIZE);

        cores[coreCount].location = writeSock;

```

```

        coreCount++;

        coreInfoPtr += CORE_TYPE_SIZE;
    }

    return coresRcvd;
}

/*****

Function: readFile

Usage: reads input data from file

Parameter Definition:
- arg: pointer to FileThrdInfo structure

*****/
void *readFile(void *arg) {

    FileThrdInfo *fileData;
    fileData = (FileThrdInfo *) arg;

    FILE * inFilePtr;

    msgStruct msg;

    // Open file
    if ((inFilePtr = fopen(fileData->fileName, "rb")) == NULL) {
        fprintf(stderr, "Error_opening_file_%s\n\r", fileData->fileName);
        removeAllQueues();
        exit(1);
    }

    int j = 0;

    int bytesRead;
    int leftoverBytes;
    int numBytesToRead;
    leftoverBytes = fileSize % DATA_SIZE;
    numBytesToRead = fileSize - leftoverBytes;

    while (j < (numBytesToRead / DATA_SIZE)) {

        if ((bytesRead = fread(&(msg.msgBuffer), DATA_SIZE, 1, inFilePtr)) < 1) {
            fprintf (
                stderr,
                "Error:_Read_unexpected_number_of_bytes_Expected_%d_read_%d\n\r",
                DATA_SIZE, bytesRead);
            removeAllQueues();
            exit(1);
        }

        // Set job ID
        // For the message queue, the msg type must be a positive number,
        // so increment numReceived before using it as the job id
        pthread_mutex_lock(&((fileData->queue)->numReceivedMutex));
        ((fileData->queue)->numReceived)++;
        memcpy(&(msg.msgCount), &((fileData->queue)->numReceived), JOB_ID_SIZE);
        pthread_mutex_unlock(&((fileData->queue)->numReceivedMutex));
    }
}

```

```

        // Place data into message queue
        if ((msgsnd((fileData->queue)->msgid, &msg,
                    (sizeof(msg) - sizeof(long)), 0)) == -1) {
            fprintf(stderr, "read_file_msgsnd_error!\n\n");
            removeAllQueues();
            exit(1);
        }

        j++;
    }

    if (leftoverBytes > 0) {

        uint8_t *msgPtr = msg.msgBuffer;

        if ((bytesRead = fread(msgPtr, leftoverBytes, 1, inFilePtr)) < 1) {
            fprintf (
                stderr,
                "Error:Read_unexpected_number_of_bytes_Expected_%d_read_%d\n",
                leftoverBytes, bytesRead);
            removeAllQueues();
            exit(1);
        }

        msgPtr += bytesRead;
        memset(msgPtr, 0, (DATA_SIZE - leftoverBytes));

        // Place data into message queue
        if ((msgsnd((fileData->queue)->msgid, &msg,
                    (sizeof(msg) - sizeof(long)), 0)) == -1) {
            fprintf (stderr, "read_file_msgsnd_error!\n\n");
            removeAllQueues();
            exit(1);
        }
    }

    // Close data file
    fclose (inFilePtr);

    // Terminate thread
    pthread_exit(NULL);
}

```

/******

Function: tripleDES

Usage: core-specific function that forms 3DES data request messages

Parameter Definition:

- arg: pointer to ThrdInfo structure

*****/

// Core-specific function (one per core type)

```

void *tripleDES(void *arg) {
    // Grab input data from the queue (make sure to have a mutex)
    // Parse the input string (this will be different for each string)
    // Format the request -> make sure to include the core type and job ID
    // There should be a separate job ID counter for each queue that
    // will need to be protected by a mutex.

```

```

    ThrdInfo *my_data;

```

```

my_data = (ThrdInfo *) arg;

char dataType = DATA_MSG_TYPE;

uint8_t msgType[MSG_TYPE_SIZE];

memcpy(msgType, &dataType, MSG_TYPE_SIZE);

uint8_t msg[MAX_MSG_SIZE];
uint8_t *msgPtr = msg;

msgStruct msgBuffer;

uint8_t keys[TOTAL_KEY_SIZE]; // 3DES input keys
uint8_t *keysPtr = keys;

uint8_t coreType[CORE_TYPE_SIZE];
memcpy(coreType, (my_data->core)->coreType, CORE_TYPE_SIZE);

FILE * keysFile;

int outBytes = 0;

// Open the file containing the keys
if ((keysFile = fopen(KEY_FILE_NAME, "rb")) == NULL) {
    fprintf(stderr, "ERROR: could not open %s\n", KEY_FILE_NAME);
    removeAllQueues();
    exit(1);
}

// Get keys
while (fscanf(keysFile, "%2x", keysPtr) == 1) {
    keysPtr++;
}

// Close the keys file
fclose(keysFile);

int leftoverBytes;
int numBytesToRead;
leftoverBytes = fileSize % DATA_SIZE;
numBytesToRead = fileSize - leftoverBytes;
int numJobsProcessed = 0;

while (numJobsProcessed < ((numBytesToRead / DATA_SIZE) + leftoverBytes)) {
    if (msgrcv((my_data->queue)->msgid, &msgBuffer, (sizeof(msgBuffer)
        - sizeof(long)), 0, 0) == -1) {
        fprintf(stderr, "tripleDES_msgrcv_error!\n");
        removeAllQueues();
        exit(1);
    } else {

        msgPtr = msg;

        // Form the outgoing message

        // Message type
        memcpy(msgPtr, msgType, MSG_TYPE_SIZE);
        msgPtr += MSG_TYPE_SIZE;

        // Core type
        memcpy(msgPtr, (my_data->core)->coreType, CORE_TYPE_SIZE);
        msgPtr += CORE_TYPE_SIZE;
    }
}

```



```

// Job ID
memcpy(msgPtr, &(msgBuffer.msgCount), JOB_ID_SIZE);
msgPtr += JOB_ID_SIZE;

// Keys
memcpy(msgPtr, keys, TOTAL_KEY_SIZE);
msgPtr += TOTAL_KEY_SIZE;

// Function select
memcpy(msgPtr, &functionSelect, FUNCT_SELECT_SIZE);
msgPtr += FUNCT_SELECT_SIZE;

// Input data
memcpy(msgPtr, msgBuffer.msgBuffer, DATA_SIZE);

//Send the message to the FPGA board
pthread_mutex_lock(&sendMutex);
if ((outBytes
    = sendMsg((my_data->core)->location, msg, sizeof(msg))
    != sizeof(msg) + sizeof(int)) {
    fprintf(stderr, "Error_sending_string_to_server!\n\n");
    removeAllQueues();
    exit(1);
}

// Check how many jobs have been processed
pthread_mutex_lock(&((my_data->queue)->numProcessedMutex));
memcpy(&(msgBuffer.msgCount), &((my_data->queue)->numProcessed),
    JOB_ID_SIZE);
((my_data->queue)->numProcessed)++;
numJobsProcessed = ((my_data->queue)->numProcessed);
pthread_mutex_unlock(&((my_data->queue)->numProcessedMutex));
pthread_mutex_unlock(&sendMutex);
}

// UNCOMMENT FOR MULTIPLE BOARD CONFIGURATIONS
// Need one of these per board in the multiple board configuration,
// so that no thread starvation will occur
//
// sched_yield ();
// sched_yield ();
// sched_yield ();
}

// Terminate thread
pthread_exit(NULL);
}

/*****

Function: matchToMap

Usage: maps core type to core map entry

Parameter Definition:
- inCoreType: core type

Return value: index of core map entry (if successful)
or -1 (if not successful)

*****/

```

```

int matchToMap(char *inCoreType) {
    // Need to compare the core type with the core types in the core map entries.
    // If there is a match, then this is a valid core type.
    //
    // If there is a mismatch, then this core type has not been defined with a
    // core map entry. The user will need to make an entry, in this case.
    int i;

    for (i = 0; i < sizeof(coreMap); i++) {
        if (strcmp(inCoreType, coreMap[i].coreType) == 0) {
            return i;
        }
    }

    return -1; // ERROR
}

```

/******

Function: mapToQueue

Usage: maps core type to input data queue

Parameter Definition:

- *inCoreType: core type*
- *queueList: input data queue*

Return value: index of queue (if successful)
or -1 (if not successful)

```

*****/
int mapToQueue(char *inCoreType, queueInfo queueList[]) {
    // If the core type matches the core type of the queue, then place that
    // result in that queue.
    //
    // If there is a mismatch, then something went wrong.
    //
    int i;

    for (i = 0; i < NUM_CORE_TYPES; i++) {
        if (strcmp(inCoreType, queueList[i].coreType) == 0) {
            return i;
        }
    }

    return -1; // ERROR
}

```

/******

Function: recvResults

Usage: receives data results from FPGA board

Parameter Definition:

- *arg: pointer to socket to receive results from*

```

*****/
void *recvResults(void *arg) {
    int *sockPtr = (int *) arg;
    int sock = *sockPtr;

    uint8_t inMsg[RESULT_SIZE];
    uint8_t *inMsgPtr = inMsg;
}

```

```

int bytesRcvd = 0;
int rListIndex;
int check;

msgStruct msg;
Result r;

pthread_mutex_lock(&threadIDMutex);
threadID[threadIDCount] = pthread_self();
threadIDCount++;
pthread_mutex_unlock(&threadIDMutex);

char coreType[CORE_TYPE_SIZE];
int numResults = 0;

int numToRecv = fileSize / DATA_SIZE;

if (( fileSize % DATA_SIZE) != 0) {
    numToRecv += (fileSize % DATA_SIZE);
}

while ((bytesRcvd = recvMsg(sock, inMsg, RESULT_SIZE)) > 0) {
    // Reset pointer
    inMsgPtr = inMsg;

    memcpy(coreType, inMsgPtr, CORE_TYPE_SIZE);
    inMsgPtr += CORE_TYPE_SIZE; // Skip over the core type

    if ((rListIndex = mapToResultList(coreType, rList)) >= 0) {
        // Set msgCount = inMsg jobID
        memcpy(&(r.seqNum), inMsgPtr, sizeof(r.seqNum));

        // Only place result into queue, so skip over job id
        inMsgPtr += JOB_ID_SIZE;
        memcpy(r.data, inMsgPtr, DATA_SIZE);

        pthread_mutex_lock(&(rList[rListIndex].mutex));
        addResult(&r, rList);
        numResults = rList[rListIndex].numItems;
        pthread_mutex_unlock(&(rList[rListIndex].mutex));

        if (numResults == numToRecv) {
            break;
        }
    } else {
        fprintf(stderr,
                "Error:_Received_message_core_type_does_not_match_a_queue_core_type!\n\r");
        exit(1);
    }
}

int i = 0;
for (i = 0; i < NUM_BOARDS; i++) {
    pthread_cancel(threadID[i]);
}

// Terminate thread
pthread_exit(NULL);
}

/*****

```

Function: writeResults

Usage: writes results to file

Parameter Definition:

– *arg: pointer to WFileThrdInfo data structure*

```
*****/
void *writeResults(void *arg) {

    WFileThrdInfo *fThrdInfo;
    fThrdInfo = (WFileThrdInfo *) arg;

    msgStruct msg;

    FILE * outFilePtr;

    int leftoverBytes;
    int numBytesToRead;
    leftoverBytes = fileSize % DATA_SIZE;
    numBytesToRead = fileSize - leftoverBytes;
    int numResultsWritten = 0;

    if ((outFilePtr = fopen(fThrdInfo->fileName, "wb")) == NULL) {
        fprintf(stderr, "Error_opening_file_%s_\n\r", fThrdInfo->fileName);
        exit(1);
    }

    // Write results to file
    while (!(resultListIsEmpty(fThrdInfo->rList))) {

        Result r;

        removeResult(&r, fThrdInfo->rList);
        fwrite(&r.data, DATA_SIZE, 1, outFilePtr);

        numResultsWritten++;
    }

    // Close file
    fclose(outFilePtr);

    // Terminate thread
    pthread_exit(NULL);
}

```

```
*****
```

Function: removeAllQueues

Usage: removes all input data queues

```
*****/
void removeAllQueues() {
    int msgqIndex = 0;

    for (msgqIndex = 0; msgqIndex < NUM_CORE_TYPES; msgqIndex++) {
        if ((msgctl(coreQueue[msgqIndex].msgid, IPC_RMID, NULL) < 0)) {
            fprintf(stderr, "Error_removing_queues!!!_\n\r");
            exit(1);
        }
    }
}

```

```

}

/*****

Function: recvMsg

Usage: receives message from FPGA board

Parameter Definition:
- sock: socket to read message from
- inMsg: message to read
- maxMsgSize: length of message to receive

Return value: number of bytes received

*****/
int recvMsg(int sock, uint8_t *inMsg, int maxMsgSize) {
    int msgHeaderSize = sizeof(int);
    int msgTotalSize = 0;
    int bytesRcvd = 0;

    // Receive the size of the incoming message
    if ((bytesRcvd = readn(sock, &msgTotalSize, msgHeaderSize))
        != msgHeaderSize) {
        fprintf(stderr,
            "Error:_expected_to_receive_%d_bytes,_received_%d!\n\r",
            msgHeaderSize, bytesRcvd);

        return -1;
    }

    // Set the number of bytes to expect
    msgTotalSize = ntohl(msgTotalSize);

    // Check the size of the incoming message
    if (msgTotalSize > maxMsgSize) {
        fprintf(
            stderr,
            "Incoming_message_too_large_for_receiving_buffer!_msgTotalSize:%d_maxMsgSize:%d\n\r",
            msgTotalSize, maxMsgSize);

        return -1;
    }

    // Clear out the message buffer
    memset(inMsg, 0, sizeof(inMsg));

    if ((bytesRcvd = readn(sock, inMsg, msgTotalSize)) != msgTotalSize) {
        fprintf(stderr,
            "Error:_expected_to_receive_%d_bytes,_received_%d!\n\r",
            msgTotalSize, bytesRcvd);

        return -1;
    }

    return bytesRcvd;
}

/*****

Function: sendMsg

Usage: sends message to FPGA board

Parameter Definition:
- sock: socket to sent message through
- outMsg: message to write

```

– *msgSize*: length of message to write

Return value: number of bytes sent

```
*****/
int sendMsg(int sock, uint8_t *outMsg, int msgSize) {
    int bytesSent = 0;
    int msgHeaderSize = sizeof(int);
    int msgTotalSize = htonl(msgSize);

    int msgBufferSize = msgHeaderSize + msgSize;

    uint8_t msgBuffer[msgBufferSize];

    uint8_t *msgBufferPtr = msgBuffer;

    memcpy(msgBufferPtr, &msgTotalSize, msgHeaderSize);

    msgBufferPtr += msgHeaderSize;

    memcpy(msgBufferPtr, outMsg, msgSize);

    if ((bytesSent = write(sock, msgBuffer, msgBufferSize)) != msgBufferSize) {
        fprintf(stderr, "Error: expected to send %d bytes, sent %d!\n\r",
                msgBufferSize, bytesSent);
        return -1;
    }

    return bytesSent;
}

```

```
/******
```

Function: readn

Usage: reads bytes from socket

Parameter Definition:

– *sock*: socket to read data from

– *inMsg*: message to read from socket

– *numBytesToRead*: number of bytes to read

Adapted from:

Unix Network Programming – The Sockets Networking API

Volume 1, Third Edition

by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff

(Page 89)

Return value: number of bytes read

```
*****/
int readn(int sock, void *inMsg, int numBytesToRead) {
    int numBytesLeft;
    int numBytesRead;
    void *inMsgPtr = inMsg;

    numBytesLeft = numBytesToRead;

    while (numBytesLeft > 0) {
        if ((numBytesRead = read(sock, inMsgPtr, numBytesLeft, 0)) < 0) {
            if (errno == EINTR) {
                printf("Calling read again ... \n\r");
            }
        }
    }
}

```

```

        numBytesRead = 0; // Call recv again
    } else {
        return -1;
    }
} else if (numBytesRead == 0) {
    printf("No more bytes...\n\r");
    break; // No more bytes
}

numBytesLeft -= numBytesRead;
inMsgPtr += numBytesRead;
}

return (numBytesToRead - numBytesLeft); // Return >= 0
}

```

/******

Function: writen

Usage: writes bytes to socket

Parameter Definition:

- sock: socket to write data to
- outMsg: message to write to socket
- numBytesToWrite: number of bytes to write

*Unix Network Programming – The Sockets Networking API
Volume 1, Third Edition
by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff
(Page 89)*

Return value: number of bytes written

```

*****/
int writen(int sock, void *outMsg, int numBytesToWrite) {
    int numBytesLeft;
    int numBytesWritten;
    void *outMsgPtr = outMsg;

    numBytesLeft = numBytesToWrite;

    while (numBytesLeft > 0) {
        if ((numBytesWritten = send(sock, outMsgPtr, numBytesLeft, 0)) <= 0) {
            if (numBytesWritten < 0 && errno == EINTR) {
                numBytesWritten = 0; // Call send again
            } else {
                return -1; // Error
            }
        }

        numBytesLeft -= numBytesWritten;
        outMsgPtr += numBytesWritten;
    }
    return numBytesToWrite;
}

```

/******

INPUT DATA QUEUE FUNCTIONS

Adapted from:

C Primer Plus by Stephen Prata

```
*****  
*****/  
  
/*****
```

Function: mapToResultList

Usage: place output data into the appropriate data results queue

Parameter Definition:
– *inCoreType*: core type
– *rl*: data results queue

Return value: index of data results queue

```
*****/  
  
int mapToResultList(char *inCoreType, ResultList rl[]) {
```

```
    int i;  
  
    for (i = 0; i < NUM_CORE_TYPES; i++) {  
        if (strncmp(inCoreType, rl[i].coreType, strlen(rl[i].coreType)) == 0) {  
            return i;  
        }  
    }  
  
    return -1; // ERROR  
}
```

```
/*****  
copyToNode and copyToItem function definitions  
*****/  
static void copyToNode(Result result, Node *pn);  
static void copyToItem(Node *pn, Result *result);
```

```
/*****  
  
Function: copyToNode
```

Usage: copies results

Parameter Definition:
– *result*: result to copy
– *pn*: node to copy result to

```
*****/  
static void copyToNode(Result result, Node *pn) {  
    pn->result = result;  
}
```

```
/*****  
  
Function: copyToItem
```

Usage: copies result to new item

Parameter Definition:
– *pn*: node to copy result to
– *result*: result to copy

```
*****/  

```



```

static void copyToItem(Node *pn, Result *result) {
    *result = pn->result;
}

/*****

Function: initResultList

Usage: initialize the data results queue

Parameter Definition:
- rl: data results queue
- inCoreType: core type

*****/
void initResultList(ResultList *rl, char *inCoreType) {
    memcpy(rl->coreType, inCoreType, CORE_TYPE_SIZE);
    pthread_mutex_init(&(rl->mutex), NULL);
    rl->front = rl->rear = NULL;
    rl->numItems = 0;
}

/*****

Function: resultListIsEmpty

Usage: checks if the data results queue is empty

Parameter Definition:
- rl: data results queue

Return value: true (if the data results queue is empty)
              false (if the data results queue is not empty)

*****/
bool resultListIsEmpty(const ResultList *rl) {
    return rl->numItems == 0;
}

/*****

Function: resultListItemCount

Usage: returns number of results in the data results queue

Parameter Definition:
- rl: data result queue

Return value: number of results in the data results queue

*****/
int resultListItemCount(const ResultList *rl) {
    return rl->numItems;
}

/*****

Function: addResult

Usage: adds result to data results queue in order by job ID

Parameter Definition:
- r: result to add
- rl: data results queue to add result to

```

*Return value: true (if successful)
false (if unsuccessful)*

```
*****/
bool addResult(Result *r, ResultList *rl) {
    Node *pnew;
    Result temp;

    Node *current, *last;

    pnew = (Node *) malloc(sizeof(Node));

    if (pnew == NULL) {
        fprintf(stderr, "Unable to allocate memory!\n");
        exit(1);
    }

    memcpy(&temp, r, sizeof(Result));
    copyToNode(temp, pnew);
    pnew->next = NULL;

    if (resultListIsEmpty(rl)) {
        rl->front = pnew; //Item goes to front
    } else if ((pnew->result).seqNum < (rl->front->result).seqNum) {
        pnew->next = rl->front;
        rl->front = pnew; // Item goes to front
    } else {
        current = rl->front->next;
        last = rl->front;

        int keepGoing = 1;

        // Keep going through list until sequence number is < next number in list
        while (keepGoing && current != NULL) {
            if ((pnew->result).seqNum > ((current->result).seqNum)) {
                last = current;
                current = current->next;
            } else {
                keepGoing = 0;
            }
        }

        last->next = pnew;
        pnew->next = current;
    }

    rl->numItems++; //One more item in queue

    return true;
}

```

Function: removeResult

Usage: removes results from the front of the data results queue

Parameter Definition:

- r: result to remove*
- rl: data results queue to remove result from*

*Return value: true (if successful)
false (if unsuccessful)*

```

*****/
bool removeResult(Result *result, ResultList *rl) {
    Node *pt;

    if (resultListIsEmpty(rl))
        return false;
    copyToItem(rl->front, result);
    pt = rl->front;
    rl->front = rl->front->next;
    free(pt);
    rl->numItems--;
    if (rl->numItems == 0)
        rl->rear = NULL;

    return true;
}

/*****

Function: emptyResults

Usage: empties data results queue

Parameter Definition:
- rl: data results queue to empty

*****/
void emptyResults(ResultList *rl) {
    Result dummy;
    while (!resultListIsEmpty(rl))
        removeResult(&dummy, rl);
}

```

A.2 wrapper.h

```

#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <stdbool.h>
#include <sys/time.h>
#include <sys/msg.h>
#include <sched.h>

// Number of different core types (i.e. tripleDES)
#define NUM_CORE_TYPES 1

// Number of boards in the system
// Must manually change this for different configurations
#define NUM_BOARDS 1

// Maximum length for a file name
#define MAX_FILE_NAME_SIZE 20

// Size of the different components of the message in terms of bytes
#define MSG_TYPE_SIZE 1 // Message type
#define CORE_TYPE_SIZE 8 // Core type
#define JOB_ID_SIZE 4 // Job ID

```

```

#define KEY_SIZE 8 // Input data
#define NUM_KEYS 3 // Input data
#define TOTAL_KEY_SIZE (KEY_SIZE * 3) // Input data
#define FUNCT_SELECT_SIZE 4 // Input data
#define DATA_SIZE 8 // Input data
#define RESULT_SIZE (CORE_TYPE_SIZE + JOB_ID_SIZE + DATA_SIZE) // Output data
#define MAX_MSG_SIZE (MSG_TYPE_SIZE + CORE_TYPE_SIZE + JOB_ID_SIZE + \
    (KEY_SIZE * NUM_KEYS) + FUNCT_SELECT_SIZE + DATA_SIZE)

// File that contains 3DES keys
#define KEY_FILE_NAME "keys.txt"

// File that contains IP addresses
#define IP_ADDR_FILE_NAME "ipAdrs.txt"

#define IP_ADDR_SIZE 15 //(###.###.###.###)
// Request message definitions
#define CORE_MSG_TYPE 'c' // Core request
#define DATA_MSG_TYPE 'd' // Data request
// Known port definitions
#define READ_PORT 9600
#define SEND_PORT 9601

//
// ipCore
//
// Represents a hardware core
//
typedef struct {
    char coreType[CORE_TYPE_SIZE]; // core type; i.e. multiplier
    int location; // location aka socket descriptor of board it's located on
} ipCore;

//
// board
//
// Represents an FPGA board
//
typedef struct {
    char ipAddr[IP_ADDR_SIZE]; //IP address of board
    int sock; // write socket
} board;

//
// coreMapEntry
//
// Represents a core map entry
//
// User must use this to associate core types with their core-specific functions
//
typedef struct {
    char coreType[CORE_TYPE_SIZE]; // Core type, i.e. "tripleDES, "FIR"...
    char fileName[MAX_FILE_NAME_SIZE]; // Input data file associated with core type
    void (*funct)(char *dataIn, char* dataOut); // Function pointer
} coreMapEntry;

//
// queueInfo
//
// Represents an input data queue
//
typedef struct {
    char coreType[CORE_TYPE_SIZE]; // core type
    int msgid; // message id of queue

```

```

        int numReceived; // number of messages received
        pthread_mutex_t numReceivedMutex; // mutex to protect numReceived
        int numProcessed; // number of messages processed
        pthread_mutex_t numProcessedMutex; // mutex to protect numProcessed
    } queueInfo;

//
// Result
//
// Represents a result from a hardware core
//
typedef struct {
    uint8_t data[DATA_SIZE]; // result from hardware core
    int seqNum; // job ID
} Result;

//
// Node
//
// Represents a node in the data results queue
//
typedef struct node {
    Result result; // data result
    struct node *next; // pointer to next result
} Node;

//
// ResultList
//
// Represents a data results queue
//
typedef struct {
    char coreType[CORE_TYPE_SIZE]; //core type
    pthread_mutex_t mutex; // mutex
    Node *front; //pointer to the front of the queue
    Node *rear; //pointer to the rear of the queue
    int numItems; //number of items in the queue
} ResultList;

//
// ThrdInfo
//
// Information needed by a core-specific thread
//
typedef struct {
    ipCore *core; // core associated with thread
    queueInfo *queue; // queue associated with thread
} ThrdInfo;

//
// FileThrdInfo
//
// Information needed by a read file thread
//
typedef struct {
    char fileName[MAX_FILE_NAME_SIZE]; // file to read from
    queueInfo *queue; // queue to put data from file into
} FileThrdInfo;

//
// WFileThrdInfo
//
// Information needed by a write results file thread
//

```

```

typedef struct {
    char fileName[MAX_FILE_NAME_SIZE]; // file to write to
    ResultList *rList; // data results queue to pull data from
} WFileThrdInfo;

//
// msgStructure
//
// Represents a message
//
typedef struct {
    long msgCount; // job ID
    uint8_t msgBuffer[MAX_MSG_SIZE]; // message buffer
} msgStruct;

//
// EXTERNAL VARIABLES
//
extern ResultList rList[NUM_CORE_TYPES];
extern coreMapEntry coreMap[NUM_CORE_TYPES];
extern queueInfo coreQueue[NUM_CORE_TYPES];
extern coreMapEntry coreMap[NUM_CORE_TYPES];
extern msgStruct msgStructure;
extern int fileSize;
extern int functionSelect;
extern pthread_t threadID[NUM_BOARDS];
extern int threadCount;
extern struct timeval start, end;
extern int coreCount;

//
// EXTERNAL FUNCTIONS
//

// Communication-related
extern int setupSocket(char *fpgaIP, unsigned short fpgaPort);
extern int setUpCoreInfo(int writeSock, int readSock, ipCore *cores,
    int numCores);
extern int recvMsg(int sock, uint8_t *inMsg, int maxMsgSize);
extern int sendMsg(int sock, uint8_t *outMsg, int sizeOfMsg);
extern int readn(int sock, void *inMsg, int numBytesToRead);
extern int writen(int sock, void *outMsg, int numBytesToWrite);

// Queue-related
extern bool resultListIsEmpty(const ResultList *rl);
extern int resultListItemCount(const ResultList *rl);
extern bool addResult(Result *r, ResultList *rl);
extern bool removeResult(Result *result, ResultList *rl);
extern void emptyResults(ResultList *rl);
extern int mapToResultList(char *inCoreType, ResultList rl[]);
extern void initResultList(ResultList *rl, char *inCoreType);
extern void setUpQueues();
extern void removeAllQueues();

// Data formatting-related
extern void *writeResults(void *arg);
extern void *readFile(void *arg);
extern void *recvResults(void *arg);

// Mapping functions
extern int matchToMap(char *inCoreType);
extern int mapToQueue(char *inCoreType, queueInfo queueList[]);

// Core-specific (3DES)

```

```
extern void *tripleDES(void *arg);  
  
// Other  
extern ipCore *getCore(char *coreType, ipCore cores[]);
```

Appendix B

Example Software Application

B.1 FPGA.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>
#include "wrapper.h"

// File names
FILE *inFilePtr; // input data file
FILE *outFilePtr; // output data file

// Variables for IP address file
FILE *ipAddrFile;
char ipAddr[IP_ADDR_SIZE]; //+1 to leave room for the null-terminator

FILE *keysFile;

// Core map
coreMapEntry coreMap[NUM_CORE_TYPES] = { { "3DES", "3DES.txt",
      (void *) tripleDES } };

// Read and write sockets
int readSock, writeSock;

// Function pointer to the core-specific parser function
void (*functPtr)(char *dataIn, char* dataOut);

// THREADS
//
```



```

// One thread per input data file to read input data into the queues
// One thread per result data queue to read result data to file
pthread_t readFileThrds[NUM_CORE_TYPES];
pthread_t writeResultThrds[NUM_CORE_TYPES];

//Input data queue information
queueInfo coreQueue[NUM_CORE_TYPES];

//Result queue information
ResultList rList [NUM_CORE_TYPES];

// Thread information
FileThrdInfo readFileThrdInfo[sizeof(coreMap)];
WFileThrdInfo writeFileThrdInfo[sizeof(coreMap)];

// Message structure
msgStruct msgStructure = { 1 };

// MESSAGE COMPONENTS

// Actual message
uint8_t msg[MAX_MSG_SIZE];
uint8_t *msgPtr = msg;

// 3DES input keys
uint8_t keys[TOTAL_KEY_SIZE];
uint8_t *keysPtr = keys;

// 3DES function select
uint8_t functSelect [FUNCT_SELECT_SIZE];
int functionSelect ;

// 3DES input data buffer
uint8_t dataBuffer[DATA_SIZE];
uint8_t *dataBufferPtr = dataBuffer;

// 3DES result buffer
uint8_t result [RESULT_SIZE];
uint8_t *resultPtr = result;

// File size (used to calculate number of data requests)
int fileSize ;

// Variables to calculate elapsed time
struct timeval start ;
struct timeval end;

// MAIN
int main(int argc, char *argv[]) {

    // COMMAND-LINE ARGUMENTS

    // core type to process data with
    char *inCoreType = argv[1];

    // number of boards in system
    int numBoards = atoi(argv[2]);

    // number of cores in system
    int numCores = atoi(argv[3]);

    // function select (0 = decryption; 1 = encryption)
    int fSelect = atoi(argv[4]);

```

```

// input data file name
char *inFileName = argv[5];

// output data file name
char *outFileName = argv[6];

// Core setup
int coresSetUp = 0;
int totalCoresSetUp = 0;

// Boards
board *boards;
boards = (board *) malloc(numBoards * sizeof(board));

// Cores
ipCore *cores;
cores = (ipCore *) malloc(numCores * sizeof(ipCore));

// MUTEXES

// One mutex per results queue
pthread_mutex_t *resultMutex;
resultMutex = (pthread_mutex_t *) malloc(numBoards
    * sizeof(pthread_mutex_t));

// One mutex per input data queue
pthread_mutex_t *coreMutex;
coreMutex = (pthread_mutex_t *) malloc(numCores * sizeof(pthread_mutex_t));

// THREADS

// One thread per socket for receiving results
pthread_t *recvResultThrds;
recvResultThrds = (pthread_t *) malloc(numBoards * sizeof(pthread_t));

// One thread per core for sending requests
pthread_t *coreThrds;
coreThrds = (pthread_t *) malloc(numCores * sizeof(pthread_t));

// Thread ID
//
// Used to kill threads in multiple board configurations, so that the
// application can exit
pthread_t * threadID;
threadID = (pthread_t *) malloc(numBoards * sizeof(pthread_t));

// Thread information
ThrdInfo *inThrdInfo;
inThrdInfo = (ThrdInfo *) malloc(numCores * sizeof(ThrdInfo));

// Check number of command-line arguments
if (argc < 7) {
    fprintf (
        stderr ,
        "Usage:_%s_<core_type>_<number_of_boards>_<number_of_cores>_\  

-----<function_select>_<input_file_name>_<output_file_name>_\\n\\r",
        argv [0]);
    exit (1);
}

// Check that the given core type is valid (of the right length)
if (strlen (inCoreType) > CORE_TYPE_SIZE) {
    fprintf (
        stderr ,

```

```

                "ERROR:_%s_has_too_many_characters_Max_number_of_characters_is_%d!!!\n\r",
                inCoreType, CORE_TYPE_SIZE);
        exit (1);
    }

    // Check that the given core type is defined in the core map
    int i;
    for (i = 0; i < sizeof(coreMap); i++) {
        if (strcmp(coreMap[i].coreType, inCoreType) == 0) {
            strcpy(coreMap[i].fileName, inFileFileName);
            strcpy(coreQueue[i].coreType, coreMap[i].coreType);
            initResultList (rList, coreMap[i].coreType);
            break;
        }

        // ERROR: Core type does not match a core type in the core map!
        fprintf (stderr,
                "ERROR:_%s_is_NOT_defined_in_the_program_core_map!!!\n\r",
                inCoreType);
        exit (1);
    }

    // Check that the function select is a valid value (0 or 1)
    // Function select = 0 for decryption
    // Function select = 1 for encryption
    if (fSelect == 0) {
        functionSelect = 0x00000000;
    } else if (fSelect == 1) {
        functionSelect = 0xFFFFFFFF;
    } else {
        fprintf (
            stderr,
            "ERROR:_invalid_function_select!_Must_be_0_(for_decryption)_or_1_(for_encryption).\n\r");
        exit (1);
    }

    // Make sure the file exists
    if ((ipAddrFile = fopen(IP_ADDR_FILE_NAME, "rb")) == NULL) { // File name should be a variable/macro
        fprintf (stderr, "Cannot_open_%s\n\r", IP_ADDR_FILE_NAME);
        exit (1);
    }

    i = 0;

    // Get the IP addresses
    while (fscanf(ipAddrFile, "%s", ipAddr) != EOF) {

        // Build the sockets for this IP address
        readSock = setupSocket(ipAddr, READ_PORT);
        writeSock = setupSocket(ipAddr, SEND_PORT);

        // Set board information
        memcpy(boards[i].ipAddr, ipAddr, sizeof(ipAddr));
        boards[i].sock = writeSock;

        // Create core list
        if ((coresSetUp = setUpCoreInfo(readSock, writeSock, cores, numCores))
            < 0) {
            fprintf (stderr, "Error_setting_up_core_information!\n\r");
            exit (1);
        }

        totalCoresSetUp += coresSetUp;
    }

```

```

        i++;
    }

    // Close IP address file
    fclose(ipAddrFile);

    // Verify that correct number of cores was set up
    if (totalCoresSetUp != numCores) {
        fprintf (
            stderr,
            "Error:~set~up~incorrect~amount~of~core~information!~Expected:~%d~Set~up:~%d~\n\r",
            numCores, totalCoresSetUp);
        exit (1);
    }

    // Set up the input data queues
    setUpQueues();

    // SET FILE SIZE BASED ON INPUT FILE
    // Open file
    if ((inFilePtr = fopen(inFileName, "rb")) == NULL) { // File name should be a variable/macro
        fprintf (stderr, "Error~opening~file~%s~\n\r", inFileName);
        exit (1);
    }

    // Get size of data file
    fseek (inFilePtr, 0, SEEK_END);
    fileSize = ftell (inFilePtr);

    // Close file
    fclose (inFilePtr);

    // Create one thread per input data file
    // There is one input data file per core type.
    for (i = 0; i < NUM_CORE_TYPES; i++) {

        strcpy(readFileThrdInfo[i].fileName, inFileName);

        int index;
        index = mapToQueue(coreMap[i].coreType, coreQueue);

        readFileThrdInfo[i].queue = &coreQueue[index];

        if (pthread_create(&readFileThrds[i], NULL, readFile,
            (void *) &readFileThrdInfo[i]))
            printf ("Thread~creation~failed!~\n\r");
    }

    int j;

    // Go through core list and create a thread for each core
    for (j = 0; j < numCores; j++) {

        int index, index2;

        // Does this core have a core map entry?
        if ((index = matchToMap(cores[j].coreType)) < 0) {
            // ERROR!
            printf ("index:~%d~\n\r", index);
            printf ("cores~type:~%s~\n\r", cores[j].coreType);
            printf (
                "ERROR:~This~core~type~does~not~exist~in~the~core~map!!!~\n\r");
        }
    }

```

```

// Does this core have an input data queue?
if ((index2 = mapToQueue(cores[j].coreType, coreQueue)) < 0) {
    // ERROR!
    printf("index2:_%d_\n\r", index2);
    printf("cores_type:_%s_\n\r", cores[j].coreType);
    printf(
        "ERROR:_This_core_type_does_not_exist_in_the_queue_list!!!_\n\r");
}

// The function that will construct the request
functPtr = coreMap[index].funct;

inThrdInfo[j].core = &cores[j];
inThrdInfo[j].queue = &coreQueue[index2];

if (pthread_create(&coreThrds[j], NULL, tripleDES,
    (void *) &inThrdInfo[j]))
    printf("Thread_creation_failed!_\n\r");
}

// Create thread to received results from the FPGA boards
for (i = 0; i < numBoards; i++) {
    if (pthread_create(&recvResultThrds[i], NULL, recvResults,
        (void *) &boards[i].sock))
        printf("Thread_creation_failed!_\n\r");
}

// Wait for all results to be received before continuing
for (i = 0; i < numBoards; i++) {
    pthread_join(recvResultThrds[i], NULL);
}

// Create write results threads
for (i = 0; i < NUM_CORE_TYPES; i++) {
    strcpy(writeFileThrdInfo[i].fileName, outFileNames);

    int index;
    index = mapToResultList(coreMap[i].coreType, rList);

    writeFileThrdInfo[i].rList = &rList[index];

    if (pthread_create(&writeResultThrds[i], NULL, writeResults,
        (void *) &writeFileThrdInfo[i]))
        printf("Thread_creation_failed!_\n\r");
}

// Wait for all results to be written
for (i = 0; i < NUM_CORE_TYPES; i++) {
    pthread_join(writeResultThrds[i], NULL);
}

// Clean up!

// Remove the message queues
removeAllQueues();

// Close the read sockets
for (i = 0; i < numCores; i++) {
    close(cores[i].location);
}

// Close the send sockets

```

```

    for (i = 0; i < numBoards; i++) {
        close(boards[i].sock);
    }

    return 0;
}

```

B.2 Script to Run Software Application

```

#!/bin/sh
echo -n "Enter core type: "
read -e CORETYPE
echo -n "Enter total number of boards in the system: "
read -e NUMBOARDS
echo -n "Enter total number of cores in the system: "
read -e NUMCORES
echo -n "Enter input filename: "
read -e INFILENAME
# UNCOMMENT TO VERIFY 3DES USING DECRYPTION
#echo -n "Enter input file hexdump file: "
#read -e INFILEHEXDUMP
OUTFILENAME=outFile.txt
# UNCOMMENT TO VERIFY 3DES USING DECRYPTION
#OUTFILENAME2=outFile2.txt
#OUTFILENAME2HEXDUMP=outFile2HexDump.txt

for i in 1 2 3 4 5 6 7 8 9 10
do
    time ./FPGA $CORETYPE $NUMBOARDS $NUMCORES 1 $INFILENAME $OUTFILENAME
# UNCOMMENT TO VERIFY 3DES USING DECRYPTION
# ./FPGA $CORETYPE $NUMBOARDS $NUMCORES 0 $OUTFILENAME $OUTFILENAME2
# hexdump $OUTFILENAME2 > $OUTFILENAME2HEXDUMP
# diff $OUTFILENAME2HEXDUMP $INFILEHEXDUMP
done

```

B.3 Makefile for Software Application

```

CC = gcc
HDRS = wrapper.h
OBJS = wrapper.o
CFLAGS = -Wall -g -lpthread
EXECS = FPGA

all: $(EXECS)

%.o: %.c $(HDRS)
    $(CC) -c $(CFLAGS) $< -o $@

FPGA: FPGA.c wrapper.o
    $(CC) $(CFLAGS) $< $(OBJS) -o $@ $(LFLAGS)

```

clean: /bin/rm -f \$(OBJS) \$(EXECS) core* *~ semantic.cache

Appendix C

Hardware Core Manager

C.1 main.c

```
/*
 * Copyright (c) 2008 Xilinx, Inc. All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

//
// NOTE TO USER:
//
// Portions of this code were adapted from the lwip demo software application from
// the Xilinx EDK Standard IP Design with Pcores Addition reference design,
// which can be found at: http://www.xilinx.com/univ/xupv5-lx110t-bsb.htm.
//

#include "xmk.h" // Must be first header file listed in order to use Xilkernel
#include <stdio.h>
#include "xenv_standalone.h"
#include "xparameters.h"
#include "netif/xadapter.h"
#include "memory_map.h"
#include "wrapper.h"
```



```

#include "xgpio.h"

/***** Constant Definitions *****/

#define ESCAPE      0x1b

// Specify the base address of the MAC we are using
#ifndef XPAR_ETHERNET_MAC_BASEADDR
#define EMAC_BASEADDR XPAR_ETHERNET_MAC_BASEADDR
#elif XPAR_LLTEMAC_0_BASEADDR
#define EMAC_BASEADDR XPAR_LLTEMAC_0_BASEADDR
#else
#error "Design needs to have at least one MAC"
#endif

/*****

Function: print_ip

Usage: prints out the IP address

Parameter Definition:
- msg: message to print
- ip: IP address to print

*****/
void print_ip(char *msg, struct ip_addr *ip) {
    print(msg);
    xil_printf ("%d.%d.%d.%d\n\r", ip4_addr1(ip), ip4_addr2(ip), ip4_addr3(ip),
                ip4_addr4(ip));
}

/*****

Function: print_ip_settings

Usage: prints out IP setting details

Parameter Definition:
- ip: IP address
- mask: netmask address
- gw: gateway address

*****/
void print_ip_settings(struct ip_addr *ip, struct ip_addr *mask,
                      struct ip_addr *gw) {

    print_ip("Board_IP: ", ip);
    print_ip("Netmask: ", mask);
    print_ip("Gateway: ", gw);
}

/*****

Function: main

Usage: starts Xilkernel

*****/
int main() {
#ifdef _MICROBLAZE_
    microblaze_init_icache_range(0, XPAR_MICROBLAZE_0_CACHE_BYTE_SIZE);
    microblaze_init_dcach_range(0, XPAR_MICROBLAZE_0_DCACHE_BYTE_SIZE);
#endif
}

```

```

        microblaze_enable_exceptions();
#endif

        // Enable caches
        XCACHE_ENABLE_ICACHE();
        XCACHE_ENABLE_DCACHE();

        // Starts main_thread()
        xilkernel_main ();
    }

struct netif server_netif ;

/*****

Function: startNetwork

Usage: sets up the IP address based on the DIP switch
settings ; sets up the core list ; sets up the network
interface ; starts the echo_application_thread

*****/
int startNetwork() {
    struct netif *netif ;
    struct ip_addr ipaddr, netmask, gw;

    // Every board has a unique MAC address.
    unsigned char
    mac_ethernet_address[] = { 0x00, 0x18, 0x3E, 0x00, 0xa3, 0x47 }; // Board #1
    //unsigned char mac_ethernet_address[] = { 0x00, 0x18, 0x3E, 0x00, 0x8a, 0x02 }; // Board #2
    //unsigned char mac_ethernet_address[] = { 0x00, 0x18, 0x3E, 0x00, 0xa2, 0xfa }; // Board #3

    netif = &server_netif;

    // Set last digit of IP address using DIP switches
    //
    // This DIP switch code is adapted from BoardConnect code written by
    // Dave Palframan (Bucknell EE '09)
    u8 data;
    XGpio gpio;
    XGpio_Initialize(&gpio, XPAR_DIP_SWITCHES_8BIT_DEVICE_ID);
    XGpio_SetDataDirection(&gpio, 1, 0xFFFFFFFF);
    data = (u8) XGpio_DiscreteRead(&gpio, 1);

    // Initialize IP addresses to be used
    IP4_ADDR(&ipaddr, 192, 168, 1, data);
    IP4_ADDR(&netmask, 255, 255, 255, 0);
    IP4_ADDR(&gw, 192, 168, 1, 1);

    // Print out IP settings of the board
    xil_printf ("\n\r");
    print_ip_settings (&ipaddr, &netmask, &gw);
    xil_printf ("\n\r");

    // Print all application headers
    print_echo_app_header();

    xil_printf ("\n\r\n\r");
    xil_printf ("xemac_add_\n\r");

    // Add network interface to the netif_list , and set it as default
    if (!xemac_add(netif, &ipaddr, &netmask, &gw, mac_ethernet_address,
        EMAC_BASEADDR)) {
        xil_printf ("Error_adding_N/W_interface\n\r");
    }
}

```

```

        return -1;
    }

    xil_printf (" netif_set_default _\n\r");
    netif_set_default (netif);

    // Specify that the network if is up
    xil_printf (" netif_set_up _\n\r");
    netif_set_up (netif);

    // Start packet receive thread – required for lwIP operation
    xil_printf ("sys_thread_new_\n\r");
    sys_thread_new(xemacif_input_thread, netif, DEFAULT_THREAD_PRIO);

    // Start receiving requests
    xil_printf ("Ready_to_receive_requests...\n\r");
    rcvRequests();

    return 0;
}

/*****

Function: main_thread

Usage: initializes lwIP; starts the network thread

*****/
int main_thread() {

    // Initialize lwIP before calling sys_thread_new
    lwip_init ();

    // Any thread using lwIP should be created using sys_thread_new
    xil_printf ("Starting_network...\n\r");
    sys_thread_new(startNetwork, NULL, DEFAULT_THREAD_PRIO);

}

```

C.2 HCM.c

```

/*
 * Copyright (c) 2008 Xilinx, Inc. All rights reserved.
 *
 * Xilinx, Inc.
 * XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS" AS A
 * COURTESY TO YOU. BY PROVIDING THIS DESIGN, CODE, OR INFORMATION AS
 * ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE, APPLICATION OR
 * STANDARD, XILINX IS MAKING NO REPRESENTATION THAT THIS IMPLEMENTATION
 * IS FREE FROM ANY CLAIMS OF INFRINGEMENT, AND YOU ARE RESPONSIBLE
 * FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE FOR YOUR IMPLEMENTATION.
 * XILINX EXPRESSLY DISCLAIMS ANY WARRANTY WHATSOEVER WITH RESPECT TO
 * THE ADEQUACY OF THE IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO
 * ANY WARRANTIES OR REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE
 * FROM CLAIMS OF INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY
 * AND FITNESS FOR A PARTICULAR PURPOSE.
 *
 */

//
// NOTE TO USER:
//

```

```

// Portions of this code were adapted from the lwip demo software application from
// the Xilinx EDK Standard IP Design with Pcores Addition reference design,
// which can be found at: http://www.xilinx.com/univ/xupv5-lx110t-bsb.htm.
//

```

```

#include "xmk.h" // Must be first header file listed in order to use Xilkernel
#include <stdio.h>
#include <string.h>

```

```

#include <stdlib.h>
#include <stdint.h>

```

```

#include <sysace_stdio.h>

```

```

#include "lwip/inet.h"
#include "lwip/sockets.h"
#include "lwipopts.h"

```

```

#include "xparameters.h"
#include "xbasic_types.h"
#include "xstatus.h"
#include "wrapper.h"

```

```

#include "sys/timer.h"

```

```

ipCore cores[NUM_CORES]; // Core list
queueInfo queueList[NUM_CORE_TYPES]; // Queue list

```

```

pthread_mutex_t sockMutex = PTHREAD_MUTEX_INITIALIZER;

```

```

int doneProcessing = 0;

```

```

pid_t threadID[NUM_CORES];

```

```

u16_t READ_PORT = 9600;
u16_t WRITE_PORT = 9601;

```

```

/*****

```

Function: checkPeripheral

Usage: checks that the peripheral exists

Parameter Definition:

– core: hardware core

```

*****/

```

```

void checkPeripheral(ipCore core) {

```

```

    Xuint32 *baseaddr_p = (Xuint32 *) (core.baseAddr);

```

```

    // Check that the peripheral exists

```

```

    XASSERT_NONVOID(baseaddr_p != XNULL);

```

```

}

```

```

/*****

```

Function: print_echo_app_header

Usage: prints a header for the application

```

*****/

```

```

void print_echo_app_header() {
    xil_printf ("\n\r-----FPGA_Processing_Platform-----\n\r");
    xil_printf ("You can connect to read port_%d and send port_%d\n\r",
                READ_PORT, WRITE_PORT);
}

/*****

Function: processCoreRequest

Usage: processes a core request from the host PC; sends
core information to the host PC

Parameter Definition:
- sock: socket to send core information to

*****/
void processCoreRequest(int sock) {
    int bytesRcvd, bytesSent;
    int i;

    // Core type buffer
    uint8_t coreType[CORE_TYPE_SIZE];

    // Number of cores on this FPGA
    int numCores = NUM_CORES;

    // Length of the core response message buffer
    int coreInfoBufferSize = sizeof(int) + (sizeof(coreType) * NUM_CORES);

    // Core response message buffer
    uint8_t coreInfoBuffer[coreInfoBufferSize];
    uint8_t *coreInfoBufferPtr = coreInfoBuffer;

    // Format value to network byte order
    numCores = htonl(numCores);

    // Copy the number of cores on this FPGA to the core response
    // message buffer
    memcpy(coreInfoBufferPtr, &numCores, sizeof(int));

    coreInfoBufferPtr += sizeof(int);

    for (i = 0; i < NUM_CORES; i++) {
        // Copy the core types into the core response message buffer
        memset(coreType, '\0', CORE_TYPE_SIZE);
        memcpy(coreType, cores[i].type, CORE_TYPE_SIZE);
        memcpy(coreInfoBufferPtr, coreType, CORE_TYPE_SIZE);
        coreInfoBufferPtr += CORE_TYPE_SIZE;
    }

    // Lock the socket, so that outgoing data does not get corrupted
    pthread_mutex_lock(&sockMutex);

    // Send core information
    if ((bytesSent = sendMsg(sock, coreInfoBuffer, coreInfoBufferSize))
        != coreInfoBufferSize + sizeof(int)) {
        printf("Error! Sent unexpected number of bytes!\n\r");
        exit(1);
    }

    // Done sending data, so unlock the socket

```

```

        pthread_mutex_unlock(&sockMutex);
    }

/*****

Function: processDataRequest

Usage: places incoming data request messages into the
appropriate data queue

Parameter Definition:
- inMsg: data request message
- inMsgSize: length of data request message

*****/
void processDataRequest(uint8_t *inMsg, int inMsgSize) {

    int i;

    // Temporary buffer
    uint8_t tmpBuffer[inMsgSize];
    uint8_t *tmpBufferPtr = tmpBuffer;

    // Message structure
    msgStruct msg = { 1 };

    // Core type buffer
    uint8_t coreType[CORE_TYPE_SIZE];

    memset(tmpBuffer, 0, sizeof(tmpBuffer));
    memset(coreType, 0, sizeof(coreType));

    // Copy the data request message to the temporary buffer
    memcpy(tmpBuffer, inMsg, inMsgSize);

    // Copy the core type from the data request
    memcpy(coreType, tmpBuffer, sizeof(coreType));

    // Copy the data request message into the message buffer
    memcpy(msg.msgBuffer, tmpBuffer, sizeof(tmpBuffer));

    // Place the data request message into the appropriate queue
    for (i = 0; i < NUM_CORE_TYPES; i++) {
        if (strcmp(coreType, queueList[i].coreType) == 0) {
            if ((msgsnd(queueList[i].msgId, &msg, (sizeof(msg) - sizeof(long)),
                0)) == -1)
                xil_printf ("msgsnd_error!_\n\r");
            break;
        }

        // No match!
        xil_printf ("ERROR: _data_queue_for_this_core_type_does_not_exist!_\n\r");
    }
}

/*****

Function: processRequest

Usage: receives messages from the host PC and processes them
based on the message type

Parameter Definition:

```

```

- read_sd: socket to read requests from
- write_sd: socket to write responses to

*****/
void processRequest(int read_sd, int write_sd) {

    // Request message
    uint8_t inMsg[MAX_MSG_SIZE];
    uint8_t *inMsgPtr = inMsg;

    int bytesRcvd;
    char msgType;
    int i = 0;

    // While there are still incoming messages...
    while ((bytesRcvd = recvMsg(read_sd, inMsg, MAX_MSG_SIZE)) > 0) {

        inMsgPtr = inMsg;

        // Copy the message type
        memcpy(&msgType, inMsgPtr, MSG_TYPE_SIZE);

        // Is it a core request?
        if (msgType == CORE_MSG_TYPE) {

            // Set up the queues
            setUpQueues();

            // Set up core list
            setUpCoreList(CORE_INFO_FILE_NAME, write_sd);

            // Process the core request message
            processCoreRequest(write_sd);

            // Is it a data request?
        } else if (msgType == DATA_MSG_TYPE) {

            // Skip over the message type portion of the data request message
            inMsgPtr += MSG_TYPE_SIZE;

            // Process the data request message
            processDataRequest(inMsgPtr, (bytesRcvd - MSG_TYPE_SIZE));

        } else {
            xil_printf ("Invalid request!");
        }
    }

    // Kill processData threads, remove message queues, close sockets, and
    // re-initialize threadIDCount, so that we can perform multiple test
    // runs of a particular configuration without having to restart the
    // FPGA board.

    // Kills processData threads
    for (i = 0; i < NUM_CORES; i++) {
        kill (threadID[i]);
    }

    // Removes message queues
    for (i = 0; i < NUM_CORE_TYPES; i++) {
        msgctl(queueList[i].msgid, IPC_RMID, (struct msqid_ds *) NULL);
    }
}

```

```

    // Close read and write sockets
    close(read_sd);
    close(write_sd);

    // Re-initialize threadIDCount
    threadIDCount = 0;
}

/*****

Function: recvRequests

Usage: Establishes the connection to the host PC. Sets up
the read and write sockets.

*****/
void recvRequests() {

    // Since the sockets API of the Xilinx-provided implementation of lwip
    // is not thread-safe, it is necessary to create two separate sockets -
    // one for reading and one for writing.

    int sock, sock2, read_sd, write_sd;
    struct sockaddr_in address, address2, remote, remote2;
    int size, size2;

    if ((sock = lwip_socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        return;

    if ((sock2 = lwip_socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)) < 0)
        return;

    address.sin_family = AF_INET;
    address.sin_port = htons(READ_PORT);
    address.sin_addr.s_addr = INADDR_ANY;

    address2.sin_family = AF_INET;
    address2.sin_port = htons(WRITE_PORT);
    address2.sin_addr.s_addr = INADDR_ANY;

    if (lwip_bind(sock, (struct sockaddr *) &address, sizeof(address)) < 0)
        return;

    if (lwip_bind(sock2, (struct sockaddr *) &address2, sizeof(address2)) < 0)
        return;

    lwip_listen(sock, MAX_SOCKETS);

    lwip_listen(sock2, MAX_SOCKETS);

    size = sizeof(remote);
    size2 = sizeof(remote2);

    while (1) {
        read_sd = lwip_accept(sock, (struct sockaddr *) &remote, &size);
        write_sd = lwip_accept(sock2, (struct sockaddr *) &remote2, &size2);
        processRequest(read_sd, write_sd);
    }
}

```


C.3 wrapper.c

```
#include "xmk.h" // Must be first header file listed

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

#include "lwip/inet.h"
#include "lwip/sockets.h"
#include "lwipopts.h"
#include "wrapper.h"

/*****

    VARIABLE DECLARATIONS

*****/
const char NEWLINE = '\n';
const char STOP = '!';

// User-defined core type map
const coreMapEntry entry[NUM_CORE_TYPES] = { { "3DES", &tripleDESFunction } };

queueInfo queueList[NUM_CORE_TYPES];
threadInfo tInfo [NUM_CORES];

// MUTEXES
pthread_mutex_t uartMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t totalSentMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t threadIDMutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t sockMutex;

int doneProcessing;
int threadIDCount = 0;

/*****

    Function: matchToMap

    Usage: Check if the given core type is a valid
    core type, as defined by the user in the core map.

    Parameter Definition:
    - inCoreType: input core type

*****/
int matchToMap(char *inCoreType) {
    int i;
    for (i = 0; i < NUM_CORE_TYPES; i++) {
        // Does this core type match a core type defined by the user
        // in the core map?
        if (strcmp(inCoreType, entry[i].coreType) == 0) {
            return i; // Yes, there is a match!
        }
    }
    return -1; // No, there is not a match!
}
```

```

}

/*****

Function: mapToQueue

Usage: Place data of a given core type into
its respective queue.

Parameter Definition:
- inCoreType: input core type

*****/

int mapToQueue(char *inCoreType) {
    int i;

    for (i = 0; i < NUM_CORE_TYPES; i++) {
        // Does this core type match one of the queues' core type?
        if (strcmp(inCoreType, queueList[i].coreType) == 0) {
            return i; // Yes, there is a match!
        }
    }

    return -1; // No, there is no match!
}

/*****

Function: setUpQueues

Usage: Sets up the queues using a message queue.

*****/

void setUpQueues() {
    int queueIndex;

    for (queueIndex = 0; queueIndex < NUM_CORE_TYPES; queueIndex++) {
        int msgid;

        memset(queueList[queueIndex].coreType, '\0',
            sizeof(queueList[queueIndex].coreType));

        // Use the core types defined in the core map to label the queues
        strcpy(queueList[queueIndex].coreType, entry[queueIndex].coreType);

        // Create a unique message id for the queue
        msgid = msgget((key_t) queueIndex, IPC_CREAT);

        // Set this queue's message id to the message id that was just
        // created
        queueList[queueIndex].msgid = msgid;
    }
}

/*****

Function: processData

Usage: Grabs input data from message queue, processes
the input data, and send the result back to the

```

host PC.

Parameter Definition:

– *arg*: Pointer to a *threadInfo* data structure

```
*****/
void *processData(void *arg) {
    threadInfo *tInfoPtr = (threadInfo *) arg;
    threadInfo tInfo = *tInfoPtr;

    msgStruct msg = { 1 };

    int bytesSent;

    pthread_mutex_lock(&threadIDMutex);
    threadID[threadIDCount] = get_currentPID();
    threadIDCount++;
    pthread_mutex_unlock(&threadIDMutex);

    int tmpDataSize = CORE_TYPE_SIZE + JOB_ID_SIZE + tInfo.inputSize;
    int inDataSize = tInfo.inputSize;
    int resultDataSize = tInfo.outputSize;
    int outDataSize = CORE_TYPE_SIZE + JOB_ID_SIZE + tInfo.outputSize;

    uint8_t tmpData[tmpDataSize];
    uint8_t *tmpDataPtr = tmpData;
    uint8_t inData[inDataSize];
    uint8_t resultData[resultDataSize];
    uint8_t outData[outDataSize];
    uint8_t *outDataPtr = outData;
    uint8_t coreType[CORE_TYPE_SIZE];
    uint32_t jobID;

    // Function pointer to the function used to process the data for this
    // thread's core type
    void (*functPtr)(uint8_t *dataIn, uint8_t *dataOut, Xuint32 baseAddr) =
        tInfo.functPtr;

    while (1) {
        // Clear data structures
        memset(tmpData, 0, tmpDataSize);
        memset(resultData, 0, resultDataSize);
        memset(outData, 0, outDataSize);
        memset(coreType, 0, CORE_TYPE_SIZE);
        memset(&jobID, 0, JOB_ID_SIZE);
        memset(&msg, 0, sizeof(msg));

        // Reset bytesSent
        bytesSent = 0;

        // Reset pointers
        tmpDataPtr = tmpData;
        outDataPtr = outData;

        // Grab data from the queue
        if (msgrcv(tInfo.msgid, &msg, (sizeof(msg) - sizeof(long)), 0, 0) == -1) {
            xil_printf("msgrcv_error!\n\r");
            exit(1);
        } else {
            // Copy the data message into the temporary buffer

```

```

memcpy(tmpData, msg.msgBuffer, tmpDataSize);

// Move the pointer to the start of the input data
tmpDataPtr += (CORE_TYPE_SIZE + JOB_ID_SIZE);

// Copy the input data
memcpy(inData, tmpDataPtr, inDataSize);

// Process the data using the core-specific function
funcPtr(inData, resultData, tInfo.baseAddr);

// Copy the core type and job ID into the data response message
memcpy(outDataPtr, msg.msgBuffer, (CORE_TYPE_SIZE + JOB_ID_SIZE));

// Move the pointer to where the output data should be copied to
outDataPtr += (CORE_TYPE_SIZE + JOB_ID_SIZE);

// Copy the output data into the data response message
memcpy(outDataPtr, resultData, resultDataSize);

// Lock the socket that data is going to be sent through,
// so the data response doesn't get corrupted
pthread_mutex_lock(&sockMutex);

// Send the data response to the host PC
if ((bytesSent = sendMsg(tInfo.sock, outData, outDataSize))
    != outDataSize + sizeof(int)) {
    pthread_mutex_lock(&uartMutex);
    xil_printf ("Error: _expected_to_send_%d_bytes,_sent_%d!\n\r",
                outDataSize, bytesSent);
    pthread_mutex_unlock(&uartMutex);
}

// Done sending, so unlock the socket
pthread_mutex_unlock(&sockMutex);
}
}

pthread_exit(NULL);
}

```

/******

Function: setUpCoreList

*Usage: Reads core information from a file on the CF card
and builds the core list based on this information.
A thread is created for each core.*

Parameter Definitions:

– *fileName*: name of the file located on the Flash card
that contains the board's core information
– *sd*: socket that data will be sent to

*****/

```

void setUpCoreList(const char *fileName, int sd) {

    int fd; // File descriptor for core information file
    char infoBuffer[MAX_RECV_BUF]; // Char. read from the core info. file
    char string[MAX_RECV_BUF] = ""; // Core information string

    int stop = 0;
    int coreIndex = 0;

```

```

int i = 0;

ipCore inCore;

// Open the file on the CF card that contains the core information
if ((fd = sysace_fopen(fileName, "r")) == 0) {
    xil_printf ("Cannot open input file: %s\n", fileName);
    exit (1);
}

// While there's still data to be read from the file ...
while (!stop) {

    // Read from file
    if (sysace_fread(infoBuffer, 1, 1, fd) == -1) {
        xil_printf ("Error with reading!\n");
        exit (1);
    }

    // Is the character not a newline and also not a stop character?
    if ((*infoBuffer != NEWLINE) && (*infoBuffer != STOP)) {
        // Yes, so concatenate it to the core information string
        strcat(string, infoBuffer);

        // Is the character is a new line character?
    } else if (*infoBuffer == NEWLINE) {

        int index, index2;

        memset(&inCore, 0, sizeof(inCore));

        // Parse the core information string into the necessary data
        // structures (e.g. core type, base address, core input data
        // size, and core output data size)
        sscanf(string, "%[^,]%lx,%d,%d\n", &inCore.type, &inCore.baseAddr,
            &inCore.inputSize, &inCore.outputSize);

        cores[coreIndex] = inCore;

        // Index of the queue that matches this core type
        index = mapToQueue(inCore.type);

        // Index of the core map entry that matches this core type
        index2 = matchToMap(inCore.type);

        // SET UP THE CORE INFORMATION

        // Base address of the core that this thread will be responsible
        // for sending input data to and receiving data from
        tInfo[i].baseAddr = cores[coreIndex].baseAddr;

        // Message id for this thread's core queue
        tInfo[i].msgid = queueList[index].msgid;

        // Pointer to the function that processes data for this core type
        tInfo[i].functPtr = entry[index2].functPtr;

        // Socket that this thread will need to send data through
        tInfo[i].sock = sd;

        // Combined length of the core's input data (in bytes)
        tInfo[i].inputSize = cores[coreIndex].inputSize;

        // Combined length of the core's output data (in bytes)

```

```

        tInfo[i].outputSize = cores[coreIndex].outputSize;

        // Create the core-specific thread
        sys_thread_new(processData, &tInfo[i], DEFAULT_THREAD_PRIO);

        coreIndex++;
        i++;

        // Reset the string
        memset(string, '\0', sizeof(string));

        // Has the end of the file been reached?
    } else if (*infoBuffer == STOP) {
        // Yes
        stop = 1;
    }
}

// Close the file
if (sysace_fclose(fd) == -1) {
    xil_printf ("Cannot close input file: %s\r\n", fileName);
    exit (1);
}
}

```

Function: mapCoreToFunc

Usage: Match the given core type to the function that is responsible for processing said core type.

Parameter Definitions:

- *inCoreType*: input core type
- *inFunc*: pointer to a user-defined function that processes data for a given core type

*****/

```

int mapCoreToFunc(char *inCoreType, void(*inFunc)(uint8_t *inData,
        uint8_t *outData, Xuint32 baseAddr)) {
    int i;

    for (i = 0; i < NUM_CORE_TYPES; i++) {
        // Does this core type match one of functions defined in the core map?
        if (strcmp(inCoreType, entry[i].coreType) == 0) {
            inFunc = entry[i].functPtr; // Yes, there is a match!
            return i;
        }
    }

    return -1; // No, there is no match!
}

```

Function: tripleDESFunction

Usage: This is a user-defined function that formats and sends input data to a triple DES hardware core.

Parameter Definitions:

- *dataIn*: input data
- *dataOut*: result of the tripleDES computation

– baseAddr: baseAddr of the tripleDES core

*****/

```
void tripleDESFunction(uint8_t *dataIn, uint8_t *dataOut, Xuint32 baseAddr) {
    long key1_in_A;
    long key1_in_B;
    long key2_in_A;
    long key2_in_B;
    long key3_in_A;
    long key3_in_B;
    long funct_select; // FFFFFFFF for encryption; 00000000 for decryption
    long data_in_A;
    long data_in_B;
    long data_out_A;
    long data_out_B;
    long encrypted_A;
    long encrypted_B;

    // Pointer to the hardware core
    volatile tripleDES *hw_core = (tripleDES*) (baseAddr);

    // Input data
    uint8_t inputData[TRIPLEDES_DATA_SIZE];
    uint8_t *inputDataPtr = inputData;

    // Output data
    uint8_t resultData[DATA_SIZE];
    uint8_t *resultDataPtr = resultData;

    // Copy the the input data
    memcpy(inputData, dataIn, TRIPLEDES_DATA_SIZE);

    // Copy the first 32 bits of key 1
    memcpy(&key1_in_A, inputDataPtr, KEY_SIZE / 2);
    inputDataPtr += KEY_SIZE / 2;

    // Copy the last 32 bits of key 1
    memcpy(&key1_in_B, inputDataPtr, KEY_SIZE / 2);
    inputDataPtr += KEY_SIZE / 2;

    // Copy the first 32 bits of key 2
    memcpy(&key2_in_A, inputDataPtr, KEY_SIZE / 2);
    inputDataPtr += KEY_SIZE / 2;

    // Copy the last 32 bits of key 2
    memcpy(&key2_in_B, inputDataPtr, KEY_SIZE / 2);
    inputDataPtr += KEY_SIZE / 2;

    // Copy the first 32 bits of key 3
    memcpy(&key3_in_A, inputDataPtr, KEY_SIZE / 2);
    inputDataPtr += KEY_SIZE / 2;

    // Copy the last 32 bits of key 3
    memcpy(&key3_in_B, inputDataPtr, KEY_SIZE / 2);
    inputDataPtr += KEY_SIZE / 2;

    // Copy the function select
    memcpy(&funct_select, inputDataPtr, FUNCT_SELECT_SIZE);
    inputDataPtr += FUNCT_SELECT_SIZE;

    // Copy the first 32 bits of input data
    memcpy(&data_in_A, inputDataPtr, DATA_SIZE / 2);
    inputDataPtr += DATA_SIZE / 2;
```

```

// Copy the last 32 bits of input data
memcpy(&data_in_B, inputDataPtr, DATA_SIZE / 2);

// WRITE DATA TO THE HARDWARE CORE

// Function select
hw_core->function_select = funct_select;

// Key 1
hw_core->key1_in_A = key1_in_A;
hw_core->key1_in_B = key1_in_B;

// Key 2
hw_core->key2_in_A = key2_in_A;
hw_core->key2_in_B = key2_in_B;

// Key 3
hw_core->key3_in_A = key3_in_A;
hw_core->key3_in_B = key3_in_B;

// Input data
hw_core->data_in_A = data_in_A;
hw_core->data_in_B = data_in_B;

// Need to wait before grabbing data from core
sleep(0);

// READ RESULT FROM THE HARDWARE CORE
data_out_A = hw_core->data_out_A;
data_out_B = hw_core->data_out_B;

// Copy the first 32 bits of the result
memcpy(resultDataPtr, &(hw_core->data_out_A), (DATA_SIZE / 2));
resultDataPtr += (DATA_SIZE / 2);

// Copy the last 32 bits of the result
memcpy(resultDataPtr, &(hw_core->data_out_B), (DATA_SIZE / 2));

// Copy the result to the output data buffer
memcpy(dataOut, resultData, TRIPLEDES_RESULT_SIZE);
}

/*****

Function: recvMsg

Usage: Receives messages

Parameter Definitions:
- sock: socket to read messages from
- inMsg: incoming message buffer
- maxMsgSize: maximum number of bytes to read

Return value: the number of bytes received

*****/
int recvMsg(int sock, uint8_t *inMsg, int maxMsgSize) {
    int msgHeaderSize = sizeof(int); // Use 4 bytes to store the message size
    int msgTotalSize = 0; // Message size (in bytes)
    int bytesRcvd = 0; // Number of bytes received

    // Read the size of the message
    if ((bytesRcvd = readn(sock, &msgTotalSize, msgHeaderSize))

```



```

        != msgHeaderSize) {
pthread_mutex_lock(&uartMutex);
xil_printf ("Error: _expected_to_receive_%d_bytes,_received_%d!\n\r",
            msgHeaderSize, bytesRcvd);
pthread_mutex_unlock(&uartMutex);
return -1;
}

// Set the number of bytes to expect
msgTotalSize = ntohl(msgTotalSize);

// Check the size of the incoming message
if (msgTotalSize > maxMsgSize) {
pthread_mutex_lock(&uartMutex);
xil_printf ("Incoming_message_too_large_for_receiving_buffer!\n\r");
pthread_mutex_unlock(&uartMutex);
return -1;
}

// Clear out the message buffer
memset(inMsg, '\0', sizeof(inMsg));

// Read the message
if ((bytesRcvd = readn(sock, inMsg, msgTotalSize)) != msgTotalSize) {
pthread_mutex_lock(&uartMutex);
xil_printf ("Error: _expected_to_receive_%d_bytes,_received_%d!\n\r",
            msgTotalSize, bytesRcvd);
pthread_mutex_unlock(&uartMutex);
return -1;
}

return bytesRcvd;
}

```

/******

Function: sendMsg

Usage: Sends messages

Parameter Definitions:

- sock: socket to send messages through
- outMsg: outgoing message buffer
- msgSize: number of bytes to send

Return value: the number of bytes sent

```

*****/
int sendMsg(int sock, uint8_t *outMsg, int msgSize) {
    int msgHeaderSize = sizeof(int); // Use 4 bytes to store the message size
    int bytesSent = 0;

    int msgTotalSize = htonl(msgSize);

    // Length of the outgoing message
    int msgBufferSize = msgHeaderSize + msgSize;

    // Outgoing message buffer
    uint8_t msgBuffer[msgBufferSize];
    uint8_t *msgBufferPtr = msgBuffer;

    // Copy the length of the outgoing message
    memcpy(msgBufferPtr, &msgTotalSize, msgHeaderSize);
    msgBufferPtr += msgHeaderSize;
}

```

```

// Copy the outgoing message
memcpy(msgBufferPtr, outMsg, msgSize);

// Send the message
if ((bytesSent = writen(sock, msgBuffer, msgBufferSize)) != msgBufferSize) {
    pthread_mutex_lock(&uartMutex);
    xil_printf ("Error: _expected_to_send_%d_bytes,_sent_%d!\n\r",
                msgHeaderSize, bytesSent);
    pthread_mutex_unlock(&uartMutex);
    return -1;
}

return bytesSent;
}

```

/******

Function: readn

Usage: reads bytes from a socket

Parameter Definitions:

- sock: socket to read bytes from
- inMsg: incoming message buffer
- numBytesToRead: number of bytes to read

Return value: the number of bytes read

--

Adapted from:

*Unix Network Programming – The Sockets Networking API
Volume 1, Third Edition
by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff
(Page 89)*

*****/

```

int readn(int sock, void *inMsg, int numBytesToRead) {
    int numBytesLeft;
    int numBytesRead;
    void *inMsgPtr = inMsg;

    numBytesLeft = numBytesToRead;

    // While there are still bytes to read...
    while (numBytesLeft > 0) {
        // Read bytes
        if ((numBytesRead = recv(sock, inMsgPtr, numBytesLeft, 0)) < 0) {
            // Handle errors
            if (errno == EINTR) {
                pthread_mutex_lock(&uartMutex);
                xil_printf ("Calling_recv_again ... \n\r");
                pthread_mutex_unlock(&uartMutex);
                numBytesRead = 0; // Call recv again
            } else {
                return -1;
            }
        } else if (numBytesRead == 0) {
            pthread_mutex_lock(&uartMutex);
            xil_printf ("No_more_bytes_for_recv...\n\r");
            pthread_mutex_unlock(&uartMutex);
            break; // No more bytes
        }
    }
}

```

```

    }

    numBytesLeft -= numBytesRead;
    inMsgPtr += numBytesRead;
}

return (numBytesToRead - numBytesLeft); // Return >= 0
}

```

/******

Function: writen

Usage: write bytes to a socket

Parameter Definitions:

- sock: socket to write bytes to
- outMsg: outgoing message buffer
- numBytesToWrite: number of bytes to write

Return value: the number of bytes written

--

Adapted from:

*Unix Network Programming – The Sockets Networking API
Volume 1, Third Edition*

by W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff

(Page 89)

```

*****/
int writen(int sock, void *outMsg, int numBytesToWrite) {
    int numBytesLeft;
    int numBytesWritten;
    void *outMsgPtr = outMsg;

    numBytesLeft = numBytesToWrite;

    void *outMsgPtrTest = outMsg;

    // While there are still bytes to write ...
    while (numBytesLeft > 0) {
        // Send bytes
        if ((numBytesWritten = send(sock, outMsgPtr, numBytesLeft, 0)) <= 0) {
            // Handle errors
            if (numBytesWritten < 0 && errno == EINTR) {
                pthread_mutex_lock(&uartMutex);
                xil_printf ("Call_send_again ... \n\r");
                pthread_mutex_unlock(&uartMutex);
                numBytesWritten = 0; // Call send again
            } else {
                return -1; // Error
            }
        }

        numBytesLeft -= numBytesWritten;
        outMsgPtr += numBytesWritten;
    }

    return numBytesToWrite;
}

```

C.4 wrapper.h

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#include <pthread.h>
#include <errno.h>
#include <stdint.h>
#include <semaphore.h>
#include "lwip/inet.h"
#include "lwipopts.h"
#include "xbasic_types.h"
#include "sys/msg.h"
#include "sys/ipc.h"
#include "sys/timer.h"
#include "sys/process.h"

/*****

MACROS

*****/

#define NUM_BOARDS 1
#define NUM_CORES 3 // Must manually change this value when configuration is changed

#define CORE_TYPE_CHARS 4 // Number of characters in the core type string
#define IP_ADDR_CHARS 15 // Number of characters in the IP address string (###.###.###.###)
#define NUM_CORE_TYPES 1 // Number of different core types in system

#define MAX_SOCKETS 5 // Maximum number of sockets to listen on
#define CORE_INFO_FILE_NAME "cores.txt" // Name of file on CF card containing core info.
#define CORE_MSG_TYPE 'c' // Core request definition
#define DATA_MSG_TYPE 'd' // Data request definition
#define MSG_TYPE_SIZE 1 // Message type size in bytes
#define CORE_TYPE_SIZE 8 // Core type size in bytes
#define JOB_ID_SIZE 4 // Job ID size in bytes
#define KEY_SIZE 8 // 3DES key input data size in bytes
#define NUM_KEYS 3 // Number of keys
#define FUNCT_SELECT_SIZE 4 // Function select input data size in bytes
#define DATA_SIZE 8 // Output data size in bytes
#define TRIPLEDES_DATA_SIZE ((KEY_SIZE * NUM_KEYS) + FUNCT_SELECT_SIZE + DATA_SIZE)
#define MAX_DATA_SIZE ((KEY_SIZE * NUM_KEYS) + FUNCT_SELECT_SIZE + DATA_SIZE)
#define MAX_MSG_SIZE (MSG_TYPE_SIZE + CORE_TYPE_SIZE + JOB_ID_SIZE + MAX_DATA_SIZE)
#define TRIPLEDES_RESULT_SIZE 8 // Output data size in bytes
#define MAX_RECV_BUF 2048

/*****

STRUCTURES

*****/

//
// Core information
//
// Represents a hardware core
//
typedef struct {
    char type[CORE_TYPE_SIZE]; // core type; i.e. multiplier
    uint32 baseAddr; // base address
    int inputSize; // input size (in bytes)
    int outputSize; // output size (in bytes)
} ipCore;
```

```

//
// Message structure
//
// Represents a message that gets placed into an input data queue
//
typedef struct {
    long msgCount; // Message number
    uint8_t msgBuffer[MAX_MSG_SIZE]; // Message buffer
} msgStruct;

//
// Core map entry
//
// A data structure used by the user to assigned a core-specific function
// to a particular core type
//
typedef struct {
    char coreType[CORE_TYPE_SIZE]; // Core type, i.e. "tripleDES", "FIR"...
    void (*functPtr)(uint8_t *dataIn, uint8_t *dataOut, Xuint32 baseAddr); // Pointer to function
} coreMapEntry;
// associated with the core type

//
// Queue information
//
// Represent an input data queue
//
typedef struct {
    char coreType[CORE_TYPE_SIZE]; // Core type
    int msgid; // Message id
} queueInfo;

//
// Thread information
//
// Represents a core-specific thread
//
typedef struct {
    int msgid; // Input data queue message id
    Xuint32 baseAddr; // Hardware core base address
    int sock; // Socket to send data to
    int inputSize; // Core input data size (in bytes)
    int outputSize; // Core output data size (in bytes)
    void (*functPtr)(uint8_t *dataIn, uint8_t *dataOut, Xuint32 baseAddr); // Pointer to core-specific function
} threadInfo;

/*****

EXTERNAL VARIABLES

*****/
// Character definitions
extern const char STOP;
extern const char NEWLINE;

// Core information
extern ipCore cores[NUM_CORES];

// Core map entries
extern const coreMapEntry entry[NUM_CORE_TYPES];

// Queue information
extern queueInfo queueList[NUM_CORE_TYPES];

```

```

// Thread information
extern threadInfo tInfo[NUM.CORES];

// Mutexes
extern pthread_mutex_t uartMutex;
extern pthread_mutex_t sockMutex;
extern pthread_mutex_t threadIDMutex;

// Thread ID
extern pid_t threadID[NUM.CORES];
extern int threadIDCount;

/*****

EXTERNAL FUNCTIONS

*****/

// Initialization functions
extern void makeQueues();
extern void makeThread();
extern void setUpCoreList(const char *fileName, int sd);
extern void setUpQueues();
extern int setupSocket(char *serverIP, unsigned short serverPort);

// Message reading/writing/passing functions
extern int readn(int sock, void *inMsg, int numBytesToRead);
extern int recvData(int sock, char *dataBuffer);
extern int recvMsg(int sock, uint8_t *inMsg, int maxMsgSize);
extern void recvRequests();
extern int sendData(int sock, int jobID, char *data);
extern int sendMsg(int sock, uint8_t *outMsg, int msgSize);
extern int writen(int sock, void *outMsg, int numBytesToWrite);

// Mapping functions
extern int mapToMap(char *inCoreType);
extern int matchToQueue(char *inCoreType);

// Other useful functions
extern ipCore *getCore(char *coreType, ipCore cores[]);
extern void *processData(void *arg);

/*****

CORE-SPECIFIC INFORMATION

*****/
//
// tripleDES
//
// Stores all the necessary input and output data for a 3DES hardware core
//
typedef struct {
    long key1_in_A; // First 32 bits of key 1
    long key1_in_B; // Last 32 bits of key 1
    long key2_in_A; // First 32 bits of key 2
    long key2_in_B; // Last 32 bits of key 2
    long key3_in_A; // First 32 bits of key 3
    long key3_in_B; // Last 32 bits of key 3
    long function_select; // 0xFFFFFFFF for encryption; 0 for decryption
    long data_in_A; // First 32 bits of input data
    long data_in_B; // Last 32 bits of input data
    long data_out_A; // First 32 bits of output data

```

```

    long data_out_B; // Last 32 bits of output data
} tripleDES;

// Core-specific function for 3DES
extern void tripleDESFunction(uint8_t *dataIn, uint8_t *dataOut,
                             Xuint32 baseAddr);

```

C.5 memory_map.h

```

/*****
*
* XILINX IS PROVIDING THIS DESIGN, CODE, OR INFORMATION "AS IS"
* AS A COURTESY TO YOU, SOLELY FOR USE IN DEVELOPING PROGRAMS AND
* SOLUTIONS FOR XILINX DEVICES. BY PROVIDING THIS DESIGN, CODE,
* OR INFORMATION AS ONE POSSIBLE IMPLEMENTATION OF THIS FEATURE,
* APPLICATION OR STANDARD, XILINX IS MAKING NO REPRESENTATION
* THAT THIS IMPLEMENTATION IS FREE FROM ANY CLAIMS OF INFRINGEMENT,
* AND YOU ARE RESPONSIBLE FOR OBTAINING ANY RIGHTS YOU MAY REQUIRE
* FOR YOUR IMPLEMENTATION. XILINX EXPRESSLY DISCLAIMS ANY
* WARRANTY WHATSOEVER WITH RESPECT TO THE ADEQUACY OF THE
* IMPLEMENTATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTIES OR
* REPRESENTATIONS THAT THIS IMPLEMENTATION IS FREE FROM CLAIMS OF
* INFRINGEMENT, IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
* FOR A PARTICULAR PURPOSE.
*
* (c) Copyright 2007 Xilinx Inc.
* All rights reserved.
*
*****/

/*****
* Header file that translates existing constants in xparameters.h
* into constants that are used by the software applications.
* Note: xparameters.h must be included before this file.
*****/

#ifndef MEMORY_MAP_H
#define MEMORY_MAP_H

#define UART_BASEADDR      XPAR_RS232_UART_1_BASEADDR
#define UART2_BASEADDR     XPAR_RS232_UART_2_BASEADDR
#define UART_BAUDRATE      9600
#define UART_CLOCK         XPAR_XUARTNS550_CLOCK_HZ
#define TMRCTR_BASEADDR    XPAR_XPS_TIMER_1_BASEADDR

#define SYSACE_BASEADDR    XPAR_SYSACE_COMPACTFLASH_BASEADDR
#define SRAM_BASEADDR      XPAR_SRAM_MEM0_BASEADDR
#define FLASH_BASEADDR     XPAR_SRAM_MEM1_BASEADDR

#define PUSHB_CWSEN_BASEADDR XPAR_PUSH_BUTTONS_5BIT_BASEADDR
#define DIPSW_BASEADDR     XPAR_DIP_SWITCHES_8BIT_BASEADDR

#define LEDS_CWSEN_BASEADDR XPAR_LEDS_POSITIONS_BASEADDR
#define GPIO_LEDS_BASEADDR XPAR_LEDS_8BIT_BASEADDR

#define ERR_LEDS_BASEADDR  XPAR_XPS_GPIO_0_BASEADDR
#define PIEZO_BASEADDR     XPAR_XPS_GPIO_1_BASEADDR
#define ROTARY_ENCODER_BASEADDR XPAR_XPS_GPIO_2_BASEADDR
#define LCD_BASEADDR       XPAR_XPS_GPIO_4_BASEADDR

#define IIC_0_BASE_ADDRESS  XPAR_XPS_IIC_0_BASEADDR
#define IIC_1_BASE_ADDRESS  XPAR_XPS_IIC_1_BASEADDR

```

```

#define IIC_2_BASE_ADDRESS XPAR_XPS_IIC_2_BASEADDR

#define IIC_BASE_ADDRESS XPAR_IIC_EEPROM_BASEADDR
#define TFT_DEVICE_ID XPAR_PLBV46_DVI_CNTRLR_0_DEVICE_ID
#define TFT_BASEADDR XPAR_PLBV46_DVI_CNTRLR_0_DCR_BASEADDR
#define ETHERNET_BASEADDR XPAR_LLTEMAC_0_BASEADDR

//#define PPC440

#ifndef PPC440
#define DDR_BASEADDR XPAR_DDR2_SDRAM_MEM_BASEADDR
#define CPU_CORE_FREQUENCY XPAR_CPU_PPC440_CORE_CLOCK_FREQ_HZ
#else
#define DDR_BASEADDR XPAR_DDR2_SDRAM_MPMC_BASEADDR
#define CPU_CORE_FREQUENCY XPAR_MICROBLAZE_CORE_CLOCK_FREQ_HZ
#endif

//#define PPC440CACHE
#ifndef PPC440CACHE
#define PPC440_ICACHE 0xC0000000
#define PPC440_DCACHE 0xC0000000
#endif

#endif

```


Appendix D

MATLAB Implementation

D.1 Script to Run 3DES

```
% Prompt user for inputs
% Note: keys and file names must be entered as strings, so they must have
% single quote marks around them
k1 = input('Enter_key_1:');
k2 = input('Enter_key_2:');
k3 = input('Enter_key_3:');
numTestRuns = input('Enter_number_of_test_runs:');
inFileName = input('Enter_input_file_name:');
outFileName = input('Enter_output_file_name:');
outResultsFileName = input('Enter_results_file_name:');

results = zeros(1,numTestRuns);

fid = fopen(outResultsFileName, 'w');

for i=1:numTestRuns
    profile on;
    TripleDES(inFileName, k1, k2, k3, 1, outFileName);
    p = profile('info');
    disp([p.FunctionTable(p.FunctionHistory(2,1)).TotalTime]);
    profile off
    results(i) = [p.FunctionTable(p.FunctionHistory(2,1)).TotalTime];
    fprintf(fid, 'Test_Run_#%d:_%10.3f_seconds\n', i, results(i));
end

average = sum(results)/numTestRuns

fprintf(fid, 'Average_Time:_%10.3f_seconds\n', average);

fclose(fid);
```

D.2 3DES MATLAB Code

The source code for helper functions, `InitPerm()`, `InvInitPerm()`, `DESRoundKeys`, `SBox()`, `DESRoundKeyFunction()`, and `DES()` can be found at [27].

```
function y = TripleDES(inFileName,inKey1,inKey2,inKey3,inFunct, outFileName)
%
% Triple DES Function
%
% Input Parameters:
%
% inFileName: input file (e.g. inFile.jpg, inFile.txt)
% inKey1: key 1 in hexadecimal format
% inKey2: key 2 in hexadecimal format
% inKey3: key 3 in hexadecimal format
% inFunct: function selection - 1 for encryption, 0 for decryption
% outFileName: output file (e.g. outFile.jpg, outFile.txt)
%
% Return value:
%
% y: binary vector of encrypted/decrypted data
%
% Overview of Triple DES:
%
% E_Key# = DES encryption with Key #
% D_Key# = DES decryption with Key #
%
% Encryption: E_Key3(D_Key2(E_Key1(Message))) = Output
% Decryption: D_Key1(E_Key2(D_Key3(Output))) = Message
%
% Example Usage:
%
% >> TripleDES('file.jpg', '0123456789ABCDEF', '0123456789ABCDEF', '0123456789ABCDEF', 1, 'outFile.jpg')
%
%
% Array of 32 zeros, for the case when there isn't an even number of 32-bit
% data chunks
extraData = 0;

% Open file
fid = fopen(inFileName);

% Read file in unsigned 32-bit chunks
[data, amountOfData] = fread(fid, inf, 'ubit32');

% Close file
fclose(fid);

fileInfo = dir(inFileName);
fileInfoBits = fileInfo.bytes*8;
extraBits = mod(fileInfoBits,32);

% Can we evenly divide all the 32-bit chunks of data into pairs to form 64-bit chunks of data?
if (mod(amountOfData, 2) == 0) % Divisible by 2
    counter = amountOfData;
else % Not divisible by 2
    data = [data ; extraData]; % Add 32 zeros
    counter = amountOfData + 1; % Increment amount of data since we added data
end
```

```

% Vector to hold output data
outData = [];

% Convert Key1 from hexadecimal format to binary vector
key1 = hex2bin(char(inKey1));

% Convert Key2 from hexadecimal format to binary vector
key2 = hex2bin(char(inKey2));

% Convert Key3 from hexadecimal format to binary vector
key3 = hex2bin(char(inKey3));

for i=1:2:counter

    % Convert from decimal to binary in 32-bit chunks since converting
    % in 64-bit chunks isn't accurate
    inMsg_A = de2bi(data(i), 32);
    inMsg_B = de2bi(data(i+1), 32);

    % Concatenate 32-bit chunks to form 64-bit chunk
    inMsg = [inMsg_A inMsg_B];

    % 3DES
    if (inFunct == 1)
        % Encryption
        outEncrypt = DES(DESDecrypt (DES (inMsg, key1), key2), key3);
        outData = [outData outEncrypt];

    elseif (inFunct == 0)
        % Decryption
        outDecrypt = DESDecrypt(DES(DESDecrypt(inMsg, key3), key2), key1);
        outData = [outData outDecrypt];

    else
        % Invalid value for function
        fprintf('ERROR! Invalid value for function! Valid values: 1 for encryption, 0 for decryption.\n\r');
    end

end

end

% Create the file
fid2 = fopen(outFileName,'w');

% Write data to file
fwrite(fid2, outData, 'ubit1');

% Close the file
fclose(fid2);

fprintf('Size of outData: %d bits \n\r', size(outData,2));

y = outData;

```

D.3 DESDecrypt MATLAB Code

```
function C = DESDecrypt(P,Key)
% C = DES(P,Key)
% Inputs: P = 64 bit (plaintext) vector P, Key = a 64 bit vector
% that serves as an admissible DES key.
% Output: C = the 64 bit (ciphertext) vector that corresponds to the
% output of the DES encryption algorithm.
% If the key is not admissible (i.e., if the parity check bits do not
% satisfy the required properties) the program will produce an error message.

% UNCOMMENT TO ENFORCE PARITY BIT REQUIREMENT OF DES
%First we check if the parity check bits are correct.
% for i = 8:8:64
%     if mod(Key(i)+sum(Key(i-7:i-1)),2)==0
%         error('Key fails parity bit requirement of DES, use another key and try again. '), return
%     end
% end

%We generate the 16 round keys; the ith row of the following matrix is the
%ith round key (of 48 bits)
RoundKeys = DESRoundKeys(Key);

%Step 1: Initial Permutation
PIP = InitPerm(P);
L = PIP(1:32); R = PIP(33:64);

%Step 2: 16 Round Feistel Cipher
for round = 1:16
    RoundKey = RoundKeys(17-round,:);
    Lnew = R;
    Rnew = xor(L,DESRoundKeyFunction(R,RoundKey));
    L = Lnew; R = Rnew;
end

%Step 3: Left/Right switch, and then apply inverse of initial permutation
%to get the ciphertext
C = InvInitPerm([R L]);
```